Algoritmos e Programação

Prof. Cléverson Tambosi

UNIASSELVI
2013

Elaboração:
Prof. Cléverson Tambosi

Revisão, Diagramação e Produção:
Centro Universitário Leonardo da Vinci – UNIASSELVI

Ficha catalográfica elaborada na fonte pela Biblioteca Dante Alighieri
UNIASSELVI – Indaial.

001.642
T155a Tambosi, Cléverson
Algoritmos e programação / Cléverson Tambosi. Indaial : Uniasselvi, 2013.
239 p. : il

ISBN 978-85-7830- 720-2

1. Programação de computadores. 2. Algoritmos.
I. Centro Universitário Leonardo da Vinci.

Impresso por:

Apresentação

Caro(a) acadêmico(a)!

Estamos iniciando o estudo da disciplina de Algoritmos e Programação. Este assunto é uma peça chave para quem pretende entrar profissionalmente na informática.

O objetivo deste estudo é aprender e exercitar a lógica, principalmente dentro da área computacional. O fortalecimento da lógica, por sua vez, é crucial para o aprendizado da programação de computadores.

Este livro é dividido em três unidades. A primeira contém os princípios básicos sobre lógica e como praticá-la voltada para a área de programação através do conceito dos algoritmos.

A segunda parte continua com estes conceitos, apresentando estruturas avançadas. Por fim, a terceira unidade conduz os estudos à prática de uma linguagem de programação.

Bons estudos!

Prof. Cléverson Tambosi

Você já me conhece das outras disciplinas? Não? É calouro? Enfim, tanto para você que está chegando agora à UNIASSELVI quanto para você que já é veterano, há novidades em nosso material.

Na Educação a Distância, o livro impresso, entregue a todos os acadêmicos desde 2005, é o material base da disciplina. A partir de 2017, nossos livros estão de visual novo, com um formato mais prático, que cabe na bolsa e facilita a leitura.

O conteúdo continua na íntegra, mas a estrutura interna foi aperfeiçoada com nova diagramação no texto, aproveitando ao máximo o espaço da página, o que também contribui para diminuir a extração de árvores para produção de folhas de papel, por exemplo.

Assim, a UNIASSELVI, preocupando-se com o impacto de nossas ações sobre o ambiente, apresenta também este livro no formato digital. Assim, você, acadêmico, tem a possibilidade de estudá-lo com versatilidade nas telas do celular, *tablet* ou computador.

Eu mesmo, UNI, ganhei um novo *layout*, você me verá frequentemente e surgirei para apresentar dicas de vídeos e outras fontes de conhecimento que complementam o assunto em questão.

Todos esses ajustes foram pensados a partir de relatos que recebemos nas pesquisas institucionais sobre os materiais impressos, para que você, nossa maior prioridade, possa continuar seus estudos com um material de qualidade.

Aproveito o momento para convidá-lo para um bate-papo sobre o Exame Nacional de Desempenho de Estudantes – ENADE.

Bons estudos!

Olá acadêmico! Para melhorar a qualidade dos materiais ofertados a você e dinamizar ainda mais os seus estudos, a Uniasselvi disponibiliza materiais que possuem o código *QR Code*, que é um código que permite que você acesse um conteúdo interativo relacionado ao tema que você está estudando. Para utilizar essa ferramenta, acesse as lojas de aplicativos e baixe um leitor de *QR Code*. Depois, é só aproveitar mais essa facilidade para aprimorar seus estudos!

BATE PAPO SOBRE O ENADE!

Olá, acadêmico!

Você já ouviu falar sobre o **ENADE**?

Se ainda não ouviu falar nada sobre o ENADE, agora você receberá algumas informações sobre o tema.

Ouviu falar? Ótimo, este informativo reforçará o que você já sabe e poderá lhe trazer novidades.

Vamos lá!

Qual é o significado da expressão ENADE?

EXAME NACIONAL DE DESEMPENHO DOS ESTUDANTES

Em algum momento de sua vida acadêmica você precisará fazer a prova ENADE.

Que prova é essa?

É **obrigatória**, organizada pelo INEP – Instituto Nacional de Estudos e Pesquisas Educacionais Anísio Teixeira.

Quem determina que esta prova é obrigatória... O **MEC – Ministério da Educação**.

O objetivo do MEC com esta prova é o de avaliar seu desempenho acadêmico assim como a qualidade do seu curso.

Fique atento! Quem não participa da prova fica impedido de se formar e não pode retirar o diploma de conclusão do curso até regularizar sua situação junto ao MEC.

Não se preocupe porque a partir de hoje nós estaremos auxiliando você nesta caminhada.

Você receberá outros informativos como este, complementando as orientações e esclarecendo suas dúvidas.

Você tem uma trilha de aprendizagem do ENADE, receberá e-mails, SMS, seu tutor e os profissionais do polo também estarão orientados.

Participará de webconferências entre outras tantas atividades para que esteja preparado para #mandar bem na prova ENADE.

Nós aqui no NEAD e também a equipe no polo estamos com você para vencermos este desafio.

Conte sempre com a gente, para juntos mandarmos bem no ENADE!

UNIASSELVI

SUMÁRIO

UNIDADE 1

INTRODUÇÃO À LÓGICA E AOS ALGORITMOS

OBJETIVOS DE APRENDIZAGEM

A partir desta unidade, você será capaz de:

- entender os princípios básicos para compreender a lógica de programação;
- entender o que é uma linguagem de programação;
- conhecer a pseudolinguagem portugol para a prática de algoritmos;
- entender e conhecer as estruturas e técnicas básicas para resolver problemas práticos através de passos a serem executados, conhecidos como algoritmos.

PLANO DE ESTUDOS

Esta unidade está dividida em cinco tópicos. Ao final de cada tópico, você encontrará atividades que auxiliarão a fixar os conceitos básicos, além de praticar a lógica através de problemas práticos.

TÓPICO 1 – ORIGEM E CONCEITOS INICIAIS

TÓPICO 2 – PROBLEMAS COMPUTACIONAIS E CONCEITOS DA ESTRUTURA DE UM ALGORITMO

TÓPICO 3 – OPERADORES

TÓPICO 4 – PRIMEIROS COMANDOS

TÓPICO 5 – CONSTRUINDO O PRIMEIRO ALGORITMO COMPLETO

TÓPICO 1

ORIGEM E CONCEITOS INICIAIS

1 INTRODUÇÃO

Antes de mais nada, vamos conhecer a origem e o que significa a palavra "algoritmo".

Essa palavra tem origem no apelido de um matemático árabe do século IX, Al-Khwarizmi (veja figura), cujos conhecimentos abriram uma nova era da matemática. Seu nome, na forma latina, é *algoritmi*, dando origem à palavra que conhecemos hoje.

FIGURA 1 – AL-KHWARIZMI

FONTE: Disponível em: <http://www.essaseoutras.xpg.com.br/os-10-matematicos-mais-importantes-da-historia-lista-completa-veja/>. Acesso em: 25 mar. 2013.

Mas afinal, **o que é algoritmo**?

Um algoritmo pode ser entendido como uma sequência de passos ou instruções, que têm por objetivo resolver um determinado problema.

Com base nisso, se pararmos para analisar, podemos perceber que os algoritmos fazem parte do dia a dia de todos nós. Por exemplo, uma receita para fazer um bolo.

O objetivo (ou "problema a ser resolvido") é fazer o bolo. Para solucionar este objetivo, há uma série de passos que devem ser executados em uma sequência definida, para atingir este objetivo (ou seja, "resolver o problema" ou, neste exemplo, fazer o bolo).

Qualquer passo não executado da forma correta, ou mesmo fora da sequência definida, pode fazer com que não obtenhamos o bolo exatamente como se deseja ou, até mesmo, pode fazer com que estrague todo o resultado.

Voltando nosso foco para a área da informática, um algoritmo é um conjunto de instruções passadas ao computador para que ele consiga resolver um problema específico.

Veremos mais adiante que um programa de computador é um algoritmo, porém escrito em uma determinada Linguagem de Programação.

Veja o exemplo de um algoritmo para fazer uma ligação a partir de um telefone fixo:

1. tirar o telefone do gancho;
2. verificar se há linha;
3. discar o número;
4. aguardar até 30 segundos ou até atenderem a ligação;
5. se atenderem, iniciar conversa;
6. se não atenderem após 30 segundos, colocar o telefone no gancho.

2 CONCEITOS

Quando trabalhamos com algoritmos (e, consequentemente, com linguagens de programação), existem vários termos e conceitos que são falados e usados. Veremos alguns destes conceitos a seguir.

2.1 DIFERENÇA ENTRE "DADO" E "INFORMAÇÃO"

É fácil confundir o que é um dado e o que é uma informação. Geralmente, quando se pergunta a muitas pessoas o que é um dado, a resposta é "um dado é uma informação". E o que é uma informação? "Uma informação é um dado".

Mas vamos desmistificar isto agora.

O **dado** é basicamente um valor, porém sem significado. Por exemplo: o valor "6,5" ou a palavra "nome".

Estes dois valores acima apresentados (e da forma como foram apresentados) são apenas dados. Se eu lhe perguntar "O que é este 6,5?", você saberá responder? Pode ser uma nota, um peso, o comprimento de algo etc.

Da mesma forma, a palavra nome, por si só, não contém informação. Esta palavra, sozinha, não informa "nome do que" estamos falando. Pode ser o nome de uma pessoa, uma cidade, um objeto etc.

Na **informação**, sim, há um significado. Por exemplo, "Nome da rua" ou "6,5 kg". Estes dois são demonstrações de informação, pois sabe-se que o nome é de uma rua e 6,5 é o peso, em quilos, de algo.

2.2 LÓGICA E LÓGICA DE PROGRAMAÇÃO

Seguindo o que foi dito anteriormente, para resolver um problema buscando utilizar o melhor caminho, é preciso ter uma boa lógica.

A lógica, de acordo com Popkin e Stroll (1993, p. 280), "[...] é o estudo do pensamento válido". Em complemento, Santos diz que "[...] a lógica é a ciência das leis ideais do pensamento e a arte de aplicá-las à pesquisa e à demonstração da verdade". (SANTOS, 1965, p. 21).

A lógica então, associada aos algoritmos, pode ser entendida como um encadeamento de pensamentos ideais para resolver um determinado problema e, consequentemente, **Lógica de Programação** é um encadeamento de instruções para resolver um problema através do computador.

Para criarmos algoritmos, é essencial trabalharmos muito com o desenvolvimento da lógica. Sem lógica, não conseguiremos resolver muitos problemas da melhor maneira.

2.3 LINGUAGEM E LINGUAGEM DE PROGRAMAÇÃO

Para escrever um algoritmo, precisamos de uma linguagem. Mas como linguagens para escrever algoritmos fogem do que estamos acostumados a falar ou ouvir, vamos entender o que significa "linguagem". Se formos analisar o conceito da palavra, podemos ver que linguagem, segundo Houaiss (1991, p. 44), pode se referir "[...] tanto à capacidade especificamente humana para aquisição e utilização de sistemas complexos de comunicação, quanto a uma instância específica de um sistema de comunicação complexo".

Em outras palavras, uma linguagem pode ser entendida como uma forma de comunicação. Por exemplo, há a "linguagem dos sinais (libras)", que é uma forma de comunicação utilizando-se gestos, uma linguagem por cores: o semáforo. Através das cores, sabe-se se é para seguir ou parar.

Seguindo este conceito, uma Linguagem de Programação é uma forma de se comunicar ou, mais especificamente, passar instruções para um computador.

Um algoritmo, ou seja, instruções para resolver um problema, pode ser escrito em basicamente qualquer linguagem de programação. Porém, quando se deseja definir um conjunto de instruções necessárias para resolver um problema, mas sem a utilização de uma linguagem de programação específica, podemos utilizar a chamada pseudolinguagem.

2.4 PSEUDOLINGUAGEM

É uma linguagem utilizada especificamente para construção de algoritmos. Também conhecida como "portugol", junção de "**portug**uês" com "al**gol**" (algol é uma antiga linguagem de programação utilizada principalmente em aplicações científicas).

Por que fazer um algoritmo? Por que não começar logo a escrever um programa, direto em uma linguagem de programação?
Com um algoritmo é possível definir os passos para resolver um problema sem precisar se preocupar com as regras complexas que uma linguagem de programação exige. Definidos os passos, estes podem ser passados para qualquer linguagem.

Você poderá encontrar vários termos para um algoritmo, escritos em uma linguagem de programação: "Código-fonte", somente "Código" ou até mesmo "Rotina". Veja exemplos de frases que você pode ouvir ou ler:
"Veja no código-fonte como está sendo calculado [...]"
"Eu fiz um código que soma dois números".
"Eu escrevi uma rotina que faz a soma de dois números".

RESUMO DO TÓPICO 1

Neste tópico, você viu que:

- A palavra algoritmo tem origem no apelido do matemático árabe Al-Khwarizmi, cujos conhecimentos abriram uma nova era da matemática.

- Um algoritmo pode ser entendido como uma sequência de passos ou instruções, que têm por objetivo resolver um determinado problema.

- Uma receita de bolo, por exemplo, com seus ingredientes e passos, pode ser considerada uma espécie de algoritmo.

- Quando se trabalha com algoritmos, muito se fala em "dados" e "informações". Um dado é basicamente um valor, porém sem significado. Por exemplo, o número "2" sozinho. Uma informação possui dados com significados. Por exemplo, "2 metros".

- Ao construirmos algoritmos, precisamos exercitar a lógica, que pode ser entendida como uma ciência das leis ideais do pensamento e a arte de aplicá-las à pesquisa e à demonstração da verdade.

- A lógica, dentro do algoritmo, pode ser vista como "lógica de programação". A lógica de programação é um encadeamento de instruções para resolver um problema através do computador.

- Os algoritmos são escritos geralmente em uma pseudolinguagem chamada portugol. Já os programas de computador são escritos em uma linguagem de programação.

AUTOATIVIDADE

1 Qual é a origem da palavra “algoritmo”?

2 O que é um algoritmo?

3 Qual é a diferença básica entre “dado” e “informação”?

4 O que é lógica de programação?

5 O que é uma pseudolinguagem?

6 Faça um algoritmo para escovar os dentes.

7 Faça um algoritmo para escrever e enviar um *e-mail*.

Depois que você desenvolver qualquer um destes algoritmos, experimente solicitar a outra pessoa que os faça também. Em seguida, compare o algoritmo feito por você com o algoritmo feito pela outra pessoa.

Você verá que, muito provavelmente, os passos (ou seja, as instruções) não são exatamente os mesmos. Você pode ter feito passos a menos ou a mais, bem como um mesmo passo pode ter sido escrito de maneira diferente. Porém, o objetivo, que era "escovar os dentes" ou "escrever e enviar um e-mail", foi atingido.

Percebe-se, com isto, que um mesmo problema pode ser resolvido de várias formas. Cada uma destas formas pode ser considerada um algoritmo diferente. Em outras palavras, um algoritmo é uma maneira (um caminho) para resolver um problema. Este caminho pode ser melhor ou pior que outro para tal resolução.

TÓPICO 2

PROBLEMAS COMPUTACIONAIS E CONCEITOS DA ESTRUTURA DE UM ALGORITMO

1 INTRODUÇÃO

Nos exemplos anteriores, vimos algoritmos para resolver problemas cotidianos, pois é uma forma fácil de demonstrar a presença dos algoritmos em nossas vidas. Porém, não é este tipo de problemas que comumente resolvemos em um computador.

Portanto, a partir de agora, os exemplos serão voltados a problemas computacionais.

Todo algoritmo é composto basicamente das seguintes etapas:

Isto quer dizer que, para resolver praticamente qualquer problema, teremos uma entrada de informações, um processamento sobre estas informações e uma saída, ou seja, uma "resposta" à pessoa que está utilizando o computador para resolver tal problema.

Na informática, é muito comum chamar a pessoa que utiliza os nossos programas de "usuários", pois, como dito, é a pessoa que usa o programa. Muitas vezes, precisaremos mencionar essa pessoa, ou seja, o usuário, que é quem informa dados para um programa esperando que o mesmo lhe devolva uma solução.
Usuário, então, é como passaremos a identificar essa pessoa a partir de agora.

Vamos analisar um problema computacional prático. Imagine a seguinte situação:

João nunca teve oportunidade de aprender matemática na vida. Você acaba de pagar R$ 15,00 a João por um serviço que ele prestou a você. João também recebeu R$ 10,00 de um outro conhecido dele.

Como João não sabe fazer cálculos, ele tem um problema a resolver: somar dois números.

João solicita a você que resolva este problema. Quais os passos que você precisa executar para resolver o problema de João?

Num primeiro momento, pensamos da seguinte forma: Ah! É só "fazer" 15 + 10, que resulta em 25.

Porém, como nosso objetivo é construir os passos (ou, a partir de agora, chamaremos sempre estes passos de **instruções**) que, futuramente, passaremos a um computador, não é tão simples assim.

Precisamos ter em mente que o computador não tem inteligência para resolver nenhum problema. Somos nós que devemos passar as instruções específicas, para que o computador consiga resolver cada um dos problemas que queremos que ele (o computador) resolva.

Quais as instruções, então, necessárias para resolver o problema do João?

Digamos que você não sabe quais são os valores que João precisa somar. Ele simplesmente encontra você e diz: "Preciso somar dois números! Resolve isso para mim?".

Qual é a primeira coisa que você tem que fazer? Precisa saber quais são os números a serem somados.

Assim, você terá que pedir estes números para João. Portanto, uma primeira instrução pode ser:

"Informe o primeiro número". João irá lhe informar.

Antes de solicitar o segundo número a João, há uma instrução muito importante, que inicialmente não nos damos conta, mas que, para que um computador possa fazer posteriormente o processamento, é crucial: **armazenar este número**.

Esquecendo a parte computacional, onde **você** armazena o número ao tentar resolver um problema como este?

Em um papel, na sua cabeça, qualquer uma dessas formas funcionaria, certo? Porém, o computador armazena esta informação na "memória do computador".

Veremos, mais adiante, como armazenar informações na memória do computador.

Suponhamos, então, que você tenha armazenado o primeiro número (15) em um papel.

Qual é a próxima "instrução a ser executada"? Seria solicitar o segundo número e armazená-lo também (ou seja, escrever no papel, memorizar etc.). João lhe informa o valor 10 e você escreve no papel.

Se pararmos para analisar, não é difícil identificar qual etapa básica de um algoritmo acabamos de executar. Essa foi a etapa de **Entrada** (e consequentemente, **armazenamento**) de dados.

Agora, para continuar a resolver o problema do João, teremos que executar a próxima etapa que, em nosso caso, é a soma dos valores.

Você pode somar os valores direto "de cabeça" ou ir fazendo o cálculo no papel. Independente da forma que você faça, estará fazendo um **Processamento** sobre os valores "armazenados". Após processar (calcular) a soma e obter o resultado, este resultado precisa ser armazenado. Mais uma vez, podemos não nos dar conta desta necessidade (do armazenamento do resultado), porém ela acontece. Mais uma vez, em sua cabeça (em sua memória) ou no papel.

Depois de obtido e armazenado o resultado, você fará a última etapa, a **Saída** de dados. Para você, a saída de dados é apresentar o resultado a João. Em outras palavras, devolver uma resposta a quem solicitou a resolução do problema.

Tudo isto parece bastante coisa, se comparado ao simples pensamento inicial de "somar os dois números e falar para João", não é mesmo?

Mas todas estas instruções são necessárias quando se pretende construir um algoritmo para resolver este problema.

Vamos ver agora, como ficaria o algoritmo?

O algoritmo a seguir ainda não está escrito na pseudolinguagem portugol, mas não é por isso que ele deixaria de ser um algoritmo, afinal, são instruções para resolver um problema.

Vamos lá:

Algoritmo para somar dois números e devolver o resultado

```
1. solicitar e armazenar o primeiro número;
2. solicitar e armazenar o segundo número;
3. somar o primeiro número com o segundo número e armazenar o
   resultado;
4. exibir o resultado a quem solicitou.
```

Viu só? Nem é tão complicado quanto parecia, não é mesmo?

Na maioria dos casos, sempre que trabalharmos com algum valor, este deverá ser armazenado em algum lugar.

Antes de começar a estudar como construir algoritmos utilizando a pseudolinguagem portugol, precisaremos estudar alguns conceitos utilizados na escrita de um algoritmo.

2 CONCEITOS DA ESTRUTURA DE UM ALGORITMO

Como já dissemos, nossos algoritmos serão escritos na pseudolinguagem portugol. Esta linguagem exige várias regras de escrita, fazendo com que um algoritmo tenha vários conceitos e partes (estruturas) distintas. Vamos vê-los a partir de agora.

2.1 IDENTIFICADORES

Vamos fazer uma analogia com nosso mundo real. Seu nome, por exemplo, é um identificador. Ele identifica você. Em um algoritmo, um identificador é um nome que pode identificar vários tipos de elementos. O próprio algoritmo em si tem um identificador, o qual representa o nome do algoritmo. Por exemplo, "**Algoritmo SomaNumeros**". Neste caso, o identificador do algoritmo chama-se SomaNumeros.

Você deve ter percebido que as palavras "Soma" e "Numeros" estão juntas, e a palavra "Numeros" está sem acento. Isto se deve ao fato de que não se pode dar qualquer nome a um identificador.

Existe uma regra para definir estes nomes. A regra é a seguinte:

1. Deve iniciar com uma letra ou com um "caractere de sublinhado" (também conhecido como *underline* ou *underscore*).

2. Depois disto, pode ser seguido por letras, números ou "caracteres de sublinhado".

Exemplos:

Nome do identificador	**Conferindo a regra**
idade	começa com uma letra e é seguida por mais letras
Salario_Bruto	começa com uma letra e é seguida por letras e caractere(s) de sublinhado
_valor	começa com um sublinhado e é seguido por letras
x1	começa por uma letra e é seguido por um número
t1000	começa por uma letra e é seguido por números

Podemos considerar, como um terceiro item desta regra, não utilizar os chamados "caracteres especiais". Enquadram-se nesta categoria as letras acentuadas ou cedilha (ç) e símbolos como !, @, #, $, %, ¨, &.

Porém, seguindo sempre o item 1 e, se necessário, o item 2 em seguida, sempre conseguiremos criar nomes permitidos para os identificadores.

2.2 VARIÁVEIS

Lembra que uma instrução importante a ser executada na etapa de **entrada** de dados é o armazenamento destes dados?

As variáveis são o "local" mais utilizado para tal armazenamento.

Para entender melhor o que é uma variável, podemos fazer a seguinte analogia: uma variável é como se fosse uma caixinha, para a qual você:

a) **deve** dar um nome; ou seja, **identificá-la**, utilizando a regra para nomes de identificadores;

b) **pode** colocar um elemento dentro dela.

Mas o que é esse elemento? É um valor qualquer que você precisa armazenar. Um número (lembra o caso do João, onde você precisava armazenar um número?), um nome etc.

Você pode armazenar **apenas um** valor por vez dentro dessa "caixinha". Isto significa que, se você quiser colocar outro valor, terá que substituir o valor anterior pelo novo.

Sempre que precisar utilizar o valor contido na "caixinha", basta "chamar o nome desta caixinha" ou "utilizar o nome da caixinha". Mas para melhor entender o que eu quis dizer com esta afirmação, vamos continuar a leitura.

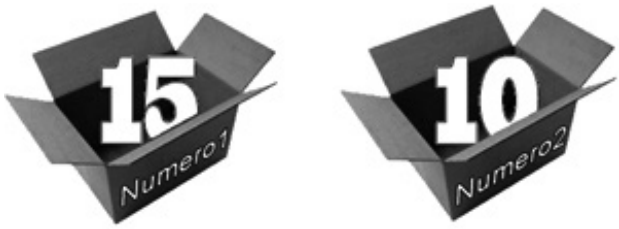
Vamos voltar ao exemplo do João. Você precisa armazenar os dois valores a serem somados. Onde vamos armazenar? Em duas variáveis.

Por que duas? Porque precisamos armazenar **dois** números e, como dito anteriormente, não podemos armazenar dois valores ao mesmo tempo em uma variável. Também não podemos armazenar um valor, depois o outro (na mesma variável), pois precisaremos dos dois valores armazenados ao mesmo tempo, para efetuar a soma.

Vamos, então, nomear a primeira caixinha de **Numero1** e a segunda caixinha de **Numero2**.

Você coloca o 15 na caixinha chamada **Numero1** e o 10 na caixinha chamada **Numero2**.

FIGURA 2 – VARIÁVEIS



FONTE: O autor

Desta forma, conseguiremos desenvolver um algoritmo "genérico" para este problema, ou seja, desenvolveremos uma instrução que some o conteúdo da caixinha chamada **Numero1** com o conteúdo da caixinha chamada **Numero2**, não importando quais são esses valores.

Se definirmos a seguinte instrução:

Numero1 + Numero2, como mencionado há pouco, não importa o que contém cada uma das caixinhas, o que nos importa, a partir de agora, é que nós definimos uma instrução que resolve um problema (o problema do João, que era somar dois números) independente de quais dados o problema envolva.

Talvez agora fique mais claro o porquê deste identificador receber o nome de "variável", não é mesmo? É porque o seu conteúdo (ou seja, o valor que ela armazena) pode variar quantas vezes forem necessárias, dependendo do que precisamos manter armazenado nelas para resolver um problema.

Entendido o que é uma variável, vamos parar de chamá-la de "caixinha" a partir de agora, certo?

2.3 CONSTANTES

Uma constante também é um local para armazenar valores. Porém, como o próprio nome diz, seu valor é constante, ou seja, não pode mudar durante a execução de um algoritmo. Ele será sempre o mesmo desde o início de um algoritmo até o final.

Para que é utilizada uma constante? Vamos ver um exemplo, mas antes leia a nota a seguir, que relembrará um importante conceito matemático, o qual será utilizado no exemplo:

O valor de Π (Pi): na matemática, PI é uma proporção numérica originada da relação entre as grandezas do perímetro (medida do contorno) de uma circunferência e seu diâmetro. É representada pela letra grega Π (cuja pronúncia é realmente "pi"), e seu valor é, aproximadamente, 3,141592653589.

Vamos lá, então: Você precisa utilizar o valor do Π (**Pi**) – 3,141592653589 – em várias instruções de um algoritmo. Imagine se, em todas as vezes que precisar utilizar o valor do Pi, você precisasse escrevê-lo novamente! Além de ter que, ou decorar o número, ou consultá-lo em algum lugar toda vez que fosse escrever, a chance de errar um número na digitação poderia ser maior.

Assim, você pode definir que uma constante chamada **pi** contenha o valor 3,141592653589 e, em todas as vezes que precisar do valor, apenas utilize o nome constante.

Veja um exemplo:

Numero1 + **pi**

Na instrução acima, estamos somando o valor da variável **Numero1** com o valor da constante **pi**. Se, nesse momento, o valor da variável **Numero1** fosse "2,4", o resultado da instrução seria 5,541592653589, ou seja, 3,141592653589 + 2,4.

2.4 TIPOS DE DADOS

Cada informação a ser armazenada ou utilizada em um algoritmo pertence a um tipo de dado definido. Para começar a entender o conceito de tipos de dados, vamos comparar com a matemática: o número 5,2 é de que tipo? Da forma mais abrangente, 5,2 pode ser um número **real**, certo?

Já o número 8, por exemplo, pertence diretamente ao conjunto dos números **inteiros**, apesar de entrar, também, no conjunto dos reais.

Reforçando: cada valor armazenado em um algoritmo terá um tipo bem definido. Isto quer dizer que, sempre que utilizar, por exemplo, uma variável em um algoritmo, para cada uma delas você precisará definir qual é o tipo de valor que ela poderá armazenar.

Porém, falamos somente em valores numéricos. Mas podemos armazenar, por exemplo, o seu nome em um algoritmo. Quando se deseja armazenar um

conjunto de caracteres (letras, símbolos ou até mesmo números que não precisarão ser utilizados para cálculos), utilizamos um tipo de dados, chamado **caractere**.

Nos algoritmos, são utilizados somente quatro **tipos de dados** primários, a saber:

- **inteiro**, **real, caractere** e **lógico**.

Os três primeiros, nós acabamos de saber para que usar. Vamos revisar:

a) **Inteiro** – armazena os valores correspondentes ao conjunto de números inteiros especificados pela matemática. Exemplos: 1; 35; 9.500; 43; 51.

b) **Real** – armazena os números representados pelo conjunto real na matemática. Exemplos: 3,5; 2,88; 32; 84,5. Como se pode perceber, assim como na matemática, este tipo de dado também consegue armazenar os valores inteiros.

c) **Caractere** – armazena valores não numéricos. Exemplos: "José da Silva"; "Rua Içara"; "001.983.994-32"; "128". Você deve estar se perguntando "mas 128, por exemplo, não é numérico?". Sim, para **nós**, são números. Mas quando números são armazenados em uma variável do tipo **caractere**, para o algoritmo não é considerado número. É apenas um conjunto de caracteres.

Mas, e o tipo **lógico**?

Um identificador definido com o tipo **lógico** poderá apenas armazenar os valores **verdadeiro** ou **falso**, nada diferente disto. Exemplos de variáveis:

```
Acabou ← verdadeiro;
Continua ← falso;
```

2.5 OPERADOR DE ATRIBUIÇÃO

Falamos bastante sobre o fato de variáveis ou constantes armazenarem valores. Mas, em um algoritmo, como colocar (**atribuir**) um valor em uma delas? É bem fácil! Basta utilizar o operador de atribuição, que é representado pelo símbolo ← (uma seta apontando para a esquerda).

Vamos ver uns exemplos práticos:

`Numero1 ← 10`	`Variável Numero1 recebe o valor 10`
`Peso ← 75.6`	`Variável Peso recebe o valor 75.6`
`Nome ← 'José'`	`Variável Nome recebe o valor José`
`Sobrenome ← 'da Silva'`	`Variável Sobrenome recebe o valor 'da Silva'`
`MaiorDeIdade ← verdadeiro`	`Variável MaiorDeIdade recebe o valor verdadeiro`

Algumas considerações sobre estes exemplos:

- Como podemos perceber, o símbolo de atribuição, em uma instrução, é lido como "recebe", ou seja, "variável recebe valor".
- Quando vamos atribuir valores a uma variável do tipo de dado caractere, temos que usar o apóstrofo (') para envolver o valor.

2.6 DECLARAÇÕES

Como mencionado anteriormente, as estruturas e identificadores (como variáveis e constantes) que forem utilizadas em um algoritmo, precisarão estar **declaradas**. O que quer dizer isso?

Declarar um determinado identificador significa informar, no início do algoritmo, que esse identificador **existe** e qual é o tipo de dado que ele poderá suportar.

2.6.1 Declaração de variáveis

A sintaxe do portugol para a declaração de variáveis é a seguinte:

```
variavel : tipo_de_dado;
```

Suponhamos uma variável chamada **Idade**, que armazenará somente valores inteiros:

```
Idade : Inteiro;
```

Podem-se declarar mais variáveis ao mesmo tempo, para cada tipo, com esta sintaxe:

```
variavel1, variavel2, variavel3, variavelN : Inteiro;
```

Vamos a mais um exemplo:

```
Idade, Quantidade, Contador, Sequencia : Inteiro;
```

E assim, com os demais tipos de dados. Exemplos:

```
Salario, Peso, Percentual : Real;
Nome_Pessoa, Nome_Automovel : Caractere;
```

2.6.2 Declaração de constantes

A declaração de constantes é um pouco diferente. Como já vimos, uma constante armazenará um valor do início ao fim do algoritmo. Quando declaramos uma constante, já atribuímos a ela o seu valor. Não precisa especificar o tipo de dado. Vejamos a sintaxe:

```
constante = valor;
```

Exemplos práticos:

```
pi = 3,141592;
percentual = 12,8;
```

Você percebeu que ao final de cada instrução há um ponto e vírgula? É mais um conceito a sabermos. Sempre, ao final de uma instrução, devemos colocar um ponto e vírgula indicando que aquela instrução acabou. Mais adiante, veremos que, se não colocarmos o ponto e vírgula, poderemos não identificar onde termina uma instrução e começa outra.

Nós acabamos de ver dois tipos de declaração (**variáveis** e **constantes**). Para cada uma delas, há um espaço específico na área de declarações de um algoritmo. Para definir a área de declaração de variáveis, deve-se utilizar a palavra "Variáveis" ou "var". Para definir a área de declaração de constantes, deve-se utilizar a palavra "Constantes" ou "const". Vamos ver como fica:

```
Algoritmo Declaracoes;
Constantes
    taxa = 10;
    raiz_de_2 = 1.41;
Variáveis
    Idade : inteiro;
    Nome : caractere;
    Salario, Valor : Real;

Início
      <Instruções>
Fim.
```

Desta forma, descrevemos, no início do algoritmo, todas as variáveis e constantes que serão utilizadas no algoritmo.

As instruções do algoritmo utilizadas para resolver o problema ao qual se destina ficarão entre os comandos **Início** e **Fim.** que ficam logo após as áreas de declarações. Isto estudaremos detalhadamente mais tarde.

RESUMO DO TÓPICO 2

Neste tópico, você viu que:

- Conhecemos um termo muito utilizado para designar a pessoa que utiliza e interage com um programa de computador: o usuário.

- Para resolver problemas computacionais, existe uma sequência comum: a entrada de dados, o processamento sobre estes dados e uma saída.

- A entrada, geralmente, são informações que o usuário deverá fornecer. O processamento é a ação a ser realizada sobre os dados, que depende do problema a ser resolvido sobre eles. Por fim, a saída é a resposta que o algoritmo (ou programa na prática) deverá fornecer ao usuário.

- Em um algoritmo existem vários tipos de identificadores; são palavras que "identificam" diversos tipos de elementos.

- Não podemos dar qualquer nome aos identificadores. Há uma regra para tal. Basta sempre iniciar por letras ou *"underline"* e, em seguida, pode conter letras, números e *"underline"*. Não pode conter caracteres especiais, como $, !, @, etc.

- Sempre que houver uma entrada de dados, estes dados deverão ser armazenados em algum local.

- O local mais comum para isto, em um algoritmo, são as variáveis. Elas podem armazenar somente um dado por vez e cada variável pode armazenar somente um tipo de dado.

- Outro local para armazenamento de dados são as constantes. A característica das constantes é que o valor deve ser armazenado antes de iniciar a execução do algoritmo. Este valor não se alterará durante toda a execução.

- Para armazenar um valor em uma variável, utiliza-se o operador de atribuição, representado por uma seta apontada para a esquerda.

- Todas as variáveis e constantes, que serão utilizadas em um algoritmo, deverão ser declaradas. A declaração serve para definir que tipo de dado a variável poderá armazenar.

AUTOATIVIDADE

1 Quais são as partes (etapas) básicas de, praticamente, qualquer problema?

2 Na área da informática, como se costuma chamar as pessoas que utilizam um programa?

3 Quais são as regras básicas para compor o nome de um identificador?

4 O que são "variáveis"?

5 O que são "constantes"?

6 Considerando que o nome de cada uma das variáveis a seguir represente o conteúdo que ela armazenará, defina o tipo de dado ideal para cada variável:

Idade _____

Nome_do_carro _____

Placa_do_carro _____

Salario _____

Quantidade_de_parafusos _____

7 Cite 3 "tipos de dados" e dê 2 exemplos, para cada tipo de dado citado, de valores que se podem armazenar em cada tipo.

8 Um “operador de atribuição” serve para atribuir o quê?

9 O que significa “declarar um identificador”?

10 Para que serve o ponto e vírgula em um algoritmo escrito em portugol?

TÓPICO 3

OPERADORES

1 INTRODUÇÃO

Como já pudemos constatar com alguns exemplos anteriores, muitas vezes será necessário resolver cálculos matemáticos em um algoritmo.

Tanto os problemas matemáticos quanto as inúmeras situações, nas quais precisaremos resolver problemas computacionais, envolvem muita lógica.

Os algoritmos dispõem de operadores aritméticos (que nos ajudam, principalmente em problemas e cálculos matemáticos), relacionais (que ajudam, principalmente, na tomada de decisão, ou seja, qual rumo um algoritmo poderá seguir) e os operadores lógicos, que trabalham normalmente em conjunto com os operadores relacionais.

Veremos, neste tópico, cada tipo de operador detalhadamente.

2 OPERADORES ARITMÉTICOS

Como inserir operações aritméticas em um algoritmo? É quase igual ao que fazemos na matemática.

Os operadores básicos são:

Operador	Ação
+	Adição
-	Subtração
*	Multiplicação
/	Divisão de valores reais
div	Divisão de valores inteiros
mod	Resto da divisão de valores inteiros

Antes de entender os operadores **div** e **mod** (que provavelmente são novos para você), vamos ver uns exemplos de como usar os demais operadores aritméticos. Para isto, vamos atribuir valores a algumas variáveis:

```
SalarioAtual ← 1550.00
Aumento ← 120.00
Percentual ← 5.00
```

Como você pode ter percebido, as casas decimais foram separadas por um "." (um ponto), ou seja, foi utilizada a notação americana. Isto porque todas as linguagens de programação utilizam essa notação. No algoritmo, poderíamos fazer com vírgula também, mas, se você quiser fazer com ponto, já estará mais acostumado(a) quando passar para uma linguagem de programação. Também não se separam os milhares nos algoritmos.
Isto vale somente para dentro de um algoritmo. Quando precisarmos exibir valores a você (fora dos algoritmos), serão exibidos com vírgula (e ponto para separar os milhares), pois é assim que nós conhecemos.

Se quisermos somar o valor do salário atual de uma pessoa (1.550, armazenado na variável **SalarioAtual**) com R$ 120,00 (contido na variável Aumento), basta executar esta instrução:

```
SalarioAtual + Aumento
```

Em, praticamente, todos os casos, quando executamos um cálculo, iremos armazenar seu resultado. Para isto, utilizaremos o sinal de atribuição: Veja:

```
NovoSalario ← SalarioAtual + Aumento
```

Com isto, estamos armazenando (atribuindo) em NovoSalario o resultado do cálculo de SalarioAtual somado com Aumento.

Segue mais um exemplo:

```
NovoSalario ←  SalarioAtual + 50 * 2
```

Assim como na matemática, a multiplicação (ou divisão) deve ser executada antes da adição (ou subtração). Assim, quando essa linha do algoritmo for executada, deve-se primeiro multiplicar 50 * 2 e o resultado disto deve ser somado com o conteúdo da variável **SalarioAtual** (no exemplo, 1.550).

O conteúdo da variável **NovoSalario**, neste caso, seria, 1.650,00.

E se quiséssemos primeiro somar o valor da variável **SalarioAtual** com 50, e, depois disto, multiplicar por 2? Mais uma vez, é exatamente como faríamos na matemática: basta adicionar os parênteses.

Ficaria assim:

```
NovoSalario ← (SalarioAtual + 50) * 2
```

Não é fácil?

Para fixar bem, vamos a mais um exemplo:

Se quisermos aplicar um percentual de aumento sobre o salário atual, iremos aplicar o valor que estiver armazenado na variável Percentual sobre o valor da variável **SalarioAtual** e armazenar o resultado na variável **SalarioComPercentual**:

```
SalarioComPercentual ← SalarioAtual + (SalarioAtual * (Percentual / 100))
```

Não precisamos, neste momento, entender como funciona o cálculo do percentual – que, podemos adiantar, é apenas aplicar a conhecida "regra de três". Vamos nos ater, agora, ao uso dos parênteses. Este cálculo será executado na seguinte ordem:

Primeiro o que está nos parênteses mais internos: Percentual / 100, ou seja, 5 / 100 = 0.05.

Depois, o resultado de 5 / 100 será multiplicado por **SalarioAtual** (1.550 * 0,05 = 77,5).

Em seguida, o resultado de 1.550 * 0,05 será somado ao conteúdo da variável **SalarioAtual** (1.550 + 77,5 = 1.627,5).

Por fim, o resultado de todo este cálculo será armazenado na variável **SalarioComPercentual** (representado pela parte da instrução "**SalarioComPercentual ←** ".

Note que, quem está desenvolvendo o algoritmo, não precisa saber os valores que as variáveis contêm. Basta definir qual é a "regra" para resolver um problema, e estas regras resolverão este problema para quaisquer valores. Este é o princípio dos algoritmos (e consequentemente, das linguagens de programação e, por fim, de um programa de computador).

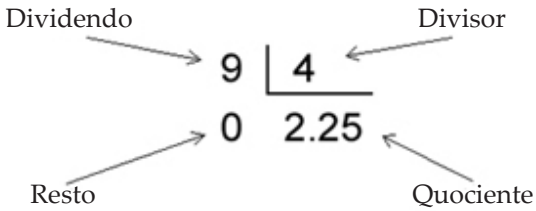
E os operadores **div** e **mod**?

Podem parecer estranhos, pois, na matemática, não estávamos acostumados a fazer cálculos com operadores formados por um conjunto de letras (**d**, **i** e **v**), mas apenas por operadores formados por símbolos (+, /, etc.).

Porém, há uma maneira bem fácil para entender o funcionamento destes operadores: tanto o **div** quanto o **mod**, realizam o mesmo cálculo que o "/", ou seja, efetuam a divisão entre dois valores. A diferença está em qual elemento (valor) de uma conta de divisão que cada operador nos traz.

Vamos utilizar o exemplo **9 dividido por 4** e comparar esta divisão com cada operador.

2.1 Operador "barra" (/)

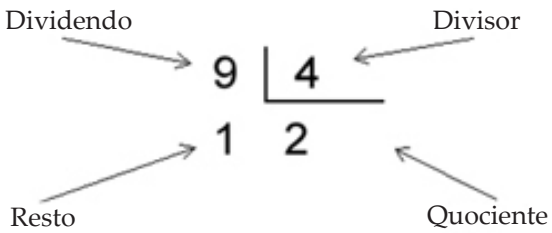


É o operador que usamos até hoje nos cálculos matemáticos, o qual retorna o quociente da divisão:

Neste caso, o resultado de **9 / 4** é **2.25**.

2.2 Operador "div"

O div também retorna o quociente, porém ele divide somente até chegar à parte inteira.



Assim, o resultado de **9 div 4** é **2**.

Como falamos anteriormente, fica um pouco estranho uma operação matemática usando um operador composto por letras, mas, entendendo o funcionamento do operador, fica fácil compreender o seu uso.

2.3 Operador "mod"

O mod faz a mesma operação que o div, porém não retorna o quociente e, sim, o resto da divisão.

Portanto, o resultado de **9 mod 4** é **1**. Para conferir, basta voltar ao cálculo anterior, no qual o resto da divisão é 1; este é o valor retornado quando o cálculo é feito com o **mod**.

Quando vamos usar esse operador "mod"? Por exemplo, quando queremos verificar se um número é par ou ímpar. Como se sabe, matematicamente, que um número é par? Quando ele é divisível por 2, certo? Em outras palavras, ele é par quando, ao dividir um número por 2 e a divisão for exata, ou seja, o resto é zero.
Portanto, se o resto da divisão de qualquer número por 2 for igual a zero, o número é par.
Vejamos o exemplo:
Numero ← 8
Se executarmos a instrução Numero mod 2, o resultado deste cálculo é 0 (zero).
Agora, se executarmos a atribuição:
Numero ← 9
O resultado de Numero mod 2 é 1, ou seja, é diferente de zero (a divisão não foi exata). Sabemos, desta forma, que o valor que Numero contém neste último exemplo não é par, ou seja, só pode ser ímpar.

3 OPERADORES RELACIONAIS

Operadores relacionais são utilizados para realizar comparações entre elementos (constantes, variáveis, valores explícitos etc.) de um mesmo tipo de dado.

Valor explícito significa utilizar o próprio valor diretamente e não, por exemplo, através de uma variável. Exemplo:
Considerando a instrução x ← 15 (variável x recebe o valor 15). Vamos analisar as duas expressões:
a) calculo ← x + 1
b) calculo ← 15 +1
Em qualquer uma das expressões, o valor da variável calculo será o mesmo, porém, no exemplo a), estamos utilizando o conteúdo da variável x, que é 15, somado com 1. Na expressão b), estamos utilizando explicitamente o valor 15 para tal cálculo.

Vamos aos operadores:

Operador	Significado	Exemplo	Explicação
=	Igualdade	a = b	Verifica se o conteúdo da variável a é igual ao conteúdo da variável b
		'a' = 'b'	Verifica se a letra "a" é igual à letra "b" (neste caso o resultado sempre será falso, pois a letra "a" nunca será igual à letra "b")
<>	Diferença	x <> 2	Verifica se o valor da variável x é diferente do número 2
		Nome <> 'André'	Verifica se o conteúdo da variável nome é diferente da palavra 'André'
>	Maior que	Valor > 1500	Verifica se o conteúdo da variável Valor é maior do que o número 1500
<	Menor que	Idade < 18	Verifica se o conteúdo da variável Idade é menor do que o número 18
>=	Maior ou igual	Salario >= 2800	Verifica se o conteúdo da variável Salario é maior ou igual ao número 2800
<=	Menor ou igual	Idade <= 18	Verifica se o conteúdo da variável Idade é menor ou igual ao número 18

Algumas considerações sobre essas operações:
a) Se pararmos para analisar, o resultado de qualquer operação relacional será verdadeiro ou falso.
Vamos pegar qualquer exemplo acima. Escolhemos, ao acaso, Nome <> 'André'.
Se executarmos a seguinte atribuição: Nome ← 'Andreas', ao verificar o resultado de Nome <> 'André', este será verdadeiro, pois o conteúdo da variável Nome realmente é diferente de 'André'.
E claramente, se executarmos Nome ← 'André', o resultado de Nome <> 'André' será falso, pois Nome não é diferente de 'André'.
b) A diferença entre o "maior que" e o "maior igual" pode ser bastante clara, mas mesmo assim, vamos reforçar.
Assumimos a execução de Idade ← 18
O resultado de Idade > 18 é falso, pois o conteúdo da variável Idade é igual a 18, e não maior.
Agora, se avaliarmos o resultado de Idade >= 18, este resultado será verdadeiro, pois o conteúdo da variável Idade pode ser tanto maior, quanto igual a 18. Neste caso, é igual.
Ainda: ao executarmos Idade ← 19, a mesma verificação Idade >= 18 também seria verdadeiro, pois como já dito, nesta verificação, a variável Idade também pode ser maior que 18.

4 OPERADORES LÓGICOS

Estes operadores são utilizados frequentemente em conjunto com operações relacionais e o resultado obtido, da mesma forma que acontece com os operadores relacionais, também será **verdadeiro** ou **falso**.

Os operadores lógicos são:

Operador	Significado	Funcionamento
e	Conjunção	Para que o resultado final seja verdadeiro, o resultado de todas as operações relacionais tem que ser verdadeiro. Basta o resultado de qualquer operação relacional envolvida ser falso, para que o resultado final seja falso também.
ou	Disjunção	Para que o resultado final seja verdadeiro, basta que o resultado de qualquer operação relacional envolvida seja verdadeiro. Para que o resultado final seja falso, o resultado de todas as operações relacionais envolvidas deve ser falso.
não	Negação	Inverte o resultado de qualquer operação (ou conjunto de operações).

O quadro a seguir mostra o resultado da junção de duas operações utilizando cada um dos operadores lógicos. Este quadro chama-se Tabela Verdade.

a	b	a E b	a OU b	NÃO a	NÃO b
V	V	V	V	F	F
V	F	F	V	F	V
F	V	F	V	V	F
F	F	F	F	V	V

Exemplos do uso de operadores lógicos junto com operações relacionais:

Exemplo 1 – Operador "**e**":
Considere **x** ← **11**.
A expressão é: (x > 0) **e** (x < 10)

O resultado da primeira expressão relacional é **verdadeiro**, pois 11 (conteúdo de x) é maior que 0 (zero). Porém, o resultado da segunda operação é **falso**, pois o conteúdo de "x" (11) não é menor do que 10.

Assim, o resultado final é **falso**, pois "11 não é maior que zero **e também** menor que 10".

Exemplo 2 – Operador "**ou**":
Considere
a ← 1
b ← 2
Expressão: (a = 1) **ou** (b = 3)

O resultado da primeira expressão é **verdadeiro**, pois a é igual a 1.

O resultado da segunda expressão é **falso**, pois b não é igual a 3.

Mas o resultado final é **verdadeiro**, pois basta uma das expressões serem verdadeiras. Como **a** é igual a **1**, **b** não precisa ser igual a **3** para que o resultado final seja verdadeiro.

Exemplo 3 – Operador "**não**":

Se comparado aos demais operadores, o uso deste operador pode ser o menos compreensível, talvez pela forma como ele tem que ser escrito nos algoritmos. Vamos ver:

O uso deste operador vem sempre antes de uma operação. Exemplo: Se considerarmos a operação **x > 0**, o uso do operador de negação com esta operação fica assim:

não x > 0

e a leitura desta operação poderia ser feita assim: "o valor da variável **x** não é maior que 0 (zero)".

Considerando que **x ← 5**, o resultado de **não x > 0** é **falso**, pois:

a) o resultado de **x > 0** é **verdadeiro**
b) como o operador é de **negação**, ele vai inverter o resultado da operação. Invertendo o **verdadeiro**, obtemos o resultado **falso**.

Mais um exemplo:

Considere
Nome ← 'Madalena'
Expressão:
não Nome = 'Iara'
o resultado é **verdadeiro**. Vamos ver por quê?

A expressão **Nome = 'Iara'** é **falso**, pois nome não é **Iara**, é **Madalena**.

Negando o resultado da expressão acima (ou seja, invertendo o **falso**), resulta em **verdadeiro**.

Mais tarde, veremos mais exemplos para assimilar melhor a ideia do operador de negação.

Agora que já estudamos diversos conceitos sobre construção de algoritmos, veremos agora como, definitivamente, construí-los.

Sempre que você escrever um algoritmo utilizando o portugol, você utilizará a seguinte estrutura básica:

```
Algoritmo <nome_do_algoritmo>;
       <área de declarações>
Início
       <Instruções>;
Fim.
```

Vamos entender melhor esta estrutura:

<nome_do_algoritmo> é um identificador à sua escolha. Por exemplo: **Algoritmo SomaNumeros;**

<área de declarações> todas as variáveis e constantes (e também outras estruturas que veremos mais tarde) precisam estar "declaradas".

Entre o **`Início`** e o **`Fim`** do algoritmo, escreveremos todas as instruções necessárias para que o algoritmo resolva o problema ao qual se propõe.

Logo veremos como escrever as tão comentadas "instruções" no portugol.

RESUMO DO TÓPICO 3

Neste tópico, você viu que:

- Há vários tipos de operadores na sintaxe de um algoritmo.
- Os operadores aritméticos servem para realizar operações matemáticas básicas. Exemplos: (A descrição está como "Identificador", pois poderia ser tanto uma variável, quanto uma constante).

a + 2	Identificador "a" somado com o valor "2"
Soma / Qtd	Identificador "Soma" dividido pelo identificador "Qtd"
(x + y) / 2	Somando dois identificadores e dividindo por 2
(v - 10) mod 5	O resultado do valor de "v" menos 10 será dividido por 5. Será obtido o resto desta divisão.

- Operadores relacionais são utilizados para fazer comparações entre valores. Estes valores podem estar explicitamente escritos, em forma de variáveis ou constantes ou mesmo em forma de cálculos. Exemplos:

Salario >= 2450	Verifica se valor do identificador "Salario" é maior ou igual a 2450
x <> y	Verifica se o valor do identificador "x" é diferente do valor do identificador "y"
num < (valor + 5)	Verifica se o valor do identificador "num" é menor do que o resultado do valor do identificador "valor" somado com 5

- Por fim, existem os operadores lógicos, utilizados geralmente em conjunto com os relacionais, podendo negar o resultado de uma expressão relacional ou, ainda, realizar uma conjunção ou disjunção entre expressões. Exemplos:

não (a > 10)	Se o valor de "a" for maior do que 10, o resultado da operação relacional será verdadeiro, porém o resultado da expressão final será falso, pois o operador "não" irá negar o verdadeiro. Se o valor de "a" for menor ou igual a 10, o resultado da operação relacional será falso. O resultado da expressão final, neste caso, será verdadeiro, pois o operador "não" irá negar o falso.
x >= 0 e x <= 10	O valor do identificador "x" tem que estar entre 0 e 10 (inclusive) para que a condição seja verdadeira.
x = 1 ou x = 2	O valor do identificador "x" pode ser, tanto 1, quanto 2, para que a condição seja verdadeira.

AUTOATIVIDADE

1 Explique o operador aritmético “mod”.

2 O que faz (verifica) a seguinte expressão: X <> 10?

3 Considerando as seguintes atribuições:

a ← 10;
b ← 12;
c ← 25;
d ← 51;

informe se o valor de cada expressão a seguir é verdadeiro ou falso:
(somente operadores relacionais)
a >= 10 _____
a >= b _____
b >= a _____
c <> a _____
a < d _____

(operadores relacionais e aritméticos)
a = (b – 2) _____
d < (b + c) _____
(a + b) >= (d – c) _____
d mod 2 = 0 _____
b + a < c _____
c <> 5 * 5 _____
a = 20 / 2 _____

(operadores relacionais e lógicos)
(a = 10) e (d = 51) _____
(a <> 10) e (d = 51) _____
(a <> 10) ou (d = 51) _____
(d > a) ou (b > c) _____
não a = 10 _____

(operadores relacionais, lógicos e aritméticos)
(a + b > c) e (d – c > a) _____
(a + b > c) ou (d – c > a) _____
(não b + c > 30) e (d > 50) _____
(a / 2 = 5) e (d = 50 + 1) _____
(a * 2 = 22) ou (d + 50 = 101) _____

TÓPICO 4

PRIMEIROS COMANDOS

1 INTRODUÇÃO

Os comandos mais básicos para o funcionamento de um algoritmo são os chamados comandos de entrada e saída. É através deles que:

- capturam-se informações externas, ou seja, dados que as pessoas que vão usar o programa de computador (representado pelo algoritmo) informarão;
- exibem-se informações (respostas) a estas pessoas.

2 O COMANDO 'ESCREVA'

Vamos voltar ao exemplo do João. Lembra que você precisou pedir os valores ao João? E no fim, informar o resultado a ele?

Muitas vezes, o algoritmo precisará solicitar um valor ou informar algo. Para isto, existe um comando chamado **Escreva**.

Por que "escreva"? Nós estamos fazendo algoritmos para resolver problemas através de um computador, certo? Com base nisto, quando você utilizar o comando **Escreva**, a ideia é que algo será escrito (aparecerá) na tela do computador.

Como se usa o **Escreva**?

```
Escreva(<conteúdo>);
```

onde **<conteúdo>** é uma mensagem ou um valor de variável, constante etc.

Por exemplo:

```
Escreva(Numero);
```

O valor da variável número será escrito.

Se quisermos escrever uma mensagem, deve-se delimitar a mensagem com apóstrofos (').

```
Escreva('Informe sua idade: ');
```

Se executarmos a instrução:

```
Escreva('Numero');
```

diferentemente da primeira instrução logo acima (`Escreva(Numero);`), não irá aparecer o conteúdo da variável **Numero** e, sim, a própria palavra Numero, pois ela está entre apóstrofos; assim, é considerada uma mensagem a ser exibida.

Por isto, o Escreva é um comando de saída; tudo o que for passado para ele, será exibido no monitor do computador, quando convertido para uma linguagem de programação.

Mais um conceito interessante a saber sobre o comando Escreva: você pode exibir várias informações de uma só vez, bastando separar os valores por vírgula.

Vejamos:

```
Idade ← 30;
Escreva('Sua idade é ',Idade);
```

Vamos analisar esta instrução. Estamos passando para o comando **Escreva** dois valores. O primeiro é um conjunto de caracteres (envolvidos por apóstrofos); o segundo é uma variável, cujo valor é 30.

O que será exibido? Exatamente esta mensagem:

```
Sua idade é 30
```

Outro exemplo:

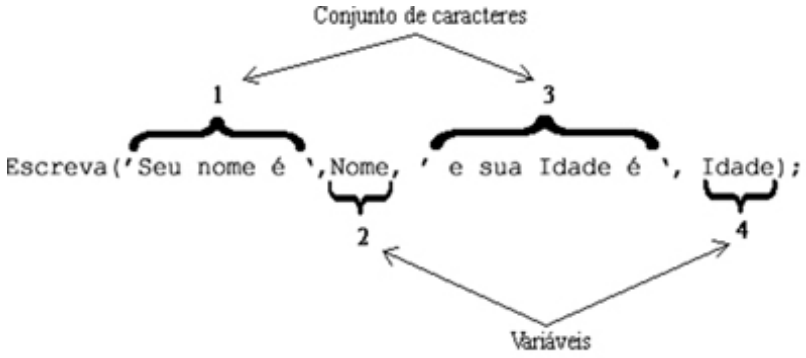
```
Nome ← 'Denise';
Idade ← 25;
Escreva('Seu nome é ',Nome, ' e sua Idade é ', Idade);
```

O resultado do comando **Escreva**, exemplificado acima, será:

```
Seu nome é Denise e sua Idade é 25
```

Neste caso, estamos passando quatro valores para o **Escreva**, conforme demonstrado na figura.

FIGURA 3 – PARÂMETROS SENDO PASSADOS PARA O COMANDO ESCREVA



FONTE: O autor

Como podemos perceber, sempre que passarmos para o comando Escreva, um conteúdo que não esteja envolvido com apóstrofos será considerado um identificador e então o seu valor será exibido.

3 O COMANDO 'LEIA'

Em muitos casos, precisamos de informações fornecidas pela pessoa que estiver utilizando o programa. Sempre que captarmos algum valor fornecido por essa pessoa (usuário do programa), iremos armazenar este valor.

O comando que "lê" um dado informado pelo usuário e armazena em uma variável é o comando **Leia**.

A sintaxe de utilização do comando é:

```
Leia(<variável>);
```

onde <variável> é o nome de uma variável na qual queremos armazenar um valor informado pelo usuário.

O funcionamento do **Leia** é o seguinte: quando esse comando for executado, o algoritmo (ou, no caso, o programa de computador) deverá deixar o usuário inserir caracteres pelo teclado livremente. Quando for identificado que o usuário pressionou a tecla <*enter*>, tudo o que foi digitado até o momento será armazenado na <variável>.

Nós não precisamos nos preocupar em identificar quando o usuário tecla <*enter*>. O próprio comando **Leia** se encarrega disso.

Assim, ao executar:
Leia (Nome);
tudo o que o usuário digitar, após pressionar <*enter*>, será armazenado na variável **Nome**.

Geralmente, um comando **Leia** é executado em conjunto com um comando **Escreva**. Por quê? Porque, sempre que deixamos o usuário digitar alguma coisa para armazenar, precisamos dizer a ele o que ele precisará informar. "Dizer a ele" é feito com o comando **Escreva**.

Vejamos um exemplo:

```
1   Algoritmo Exemplo;
2   var
3          Nome : Caractere;
4          Idade : Inteiro;
5          Salario : Real;
6
7   Início
8          Escreva('Informe seu nome: ');
9          Leia(Nome);
10
11         Escreva('Informe sua idade: ');
12         Leia(Idade);
13
14         Escreva('Informe seu salário: ');
15         Leia(Salario);
16  Fim.
```

O que foi feito no exemplo anterior: foram solicitadas três informações ao usuário. Logo após solicitar (exibir uma mensagem na tela pedindo a informação) uma informação, já é executado o comando **Leia** para captar (ler) tudo o que o usuário digita e armazenar na respectiva variável.

Um exemplo para finalizar: vamos utilizar os comandos já exibidos anteriormente:

```
Escreva('Informe seu nome: ');
Leia(Nome);
```

Neste caso, aparecerá na tela a mensagem "Informe seu nome:" e o programa ficará aguardando o usuário digitar algo e pressionar <*enter*>.

O usuário sabe o que fazer. Ao digitar seu nome e pressionar <*enter*>, o nome digitado será armazenado na variável **Nome**.

4 COMENTÁRIOS

Entre as linhas de um algoritmo, pode-se fazer necessário colocar alguns comentários, informando o que faz um determinado cálculo ou o porquê de ter utilizado tal instrução.

Um comentário não será executado pelo algoritmo. Ele serve apenas como consultas futuras rápidas para o programador.

Um comentário é delimitado pelos caracteres { e }, ou seja, chave aberta e chave fechada.

Exemplo:

```
1  Algoritmo Comentario;
2  var
3         SalarioAtual, SalarioFinal : Real;
4
5  Início
6         { As linhas abaixo solicitam e armazenam o salário atual }
7         Escreva('Informe o salário: ');
8         Leia(SalarioAtual);
9
10        { A linha abaixo calcula o salário final, aumentando em 30% o
11 salário atual }
12        SalarioFinal ← SalarioAtual + (SalarioAtual * 30 / 100);
13
14        { Exibindo o salário final com aumento }
15        Escreva('Novo salário: ', SalarioFinal);
16 Fim.
```

Nas linhas 6, 10 e 14, podemos observar o uso de comentários dentro de um algoritmo.

RESUMO DO TÓPICO 4

Neste tópico, você viu que:

- Os comandos mais básicos existentes nos algoritmos são os comandos de escrita e leitura ou, em outras palavras, entrada de dados e saída de informações.

- O comando **Escreva** serve para exibir uma informação qualquer ao usuário. Esta informação pode ser, entre outras, a solicitação de algo, como, por exemplo, "Informe a sua idade" ou a exibição de algum dado, por exemplo, "O montante da aplicação é R$ 1.000,00".

- O comando **Leia** serve para capturar um dado que o usuário informa e armazenar este dado em uma variável.

- Outro conceito utilizado nos algoritmos são os "comentários". São utilizados para registrar informações junto às instruções de um algoritmo. Entre as várias utilidades, pode-se explicar o que faz um determinado cálculo ou o porquê foi necessário executar determinadas instruções.

AUTOATIVIDADE

1 Para que serve o comando Escreva?

2 Para que serve o comando Leia?

3 Para que servem os comentários? Dê um exemplo do seu uso.

4 Faça um algoritmo que solicite e armazene o peso de uma pessoa (você determinará a variável na qual será armazenado o valor do peso). Em seguida, o algoritmo deverá exibir a mensagem "Seu peso é <peso>", onde, no lugar de <peso>, deverá ser exibido o peso que a pessoa informou.

Acima de cada uma destas instruções, faça um comentário (no formato portugol) explicando o que cada instrução faz.

5 Desenvolva um algoritmo que solicite e armazene as seguintes informações de uma pessoa:

a) Nome
b) Sobrenome
c) Idade
d) Endereço

Após ler as informações, o algoritmo deverá exibir uma mensagem semelhante a esta: "Olá, **Humberto Pereira**. Você tem **23** anos e mora na **Rua Antônio Pchara**".

Os dados em negrito serão as informações que o usuário forneceu.

TÓPICO 5

CONSTRUINDO O PRIMEIRO ALGORITMO COMPLETO

1 INTRODUÇÃO

Vamos construir um algoritmo gradativamente, analisando cada etapa, com o objetivo de resolver um primeiro problema completo.

Agora que chegamos aqui, já temos uma boa base pra construir um algoritmo. Vamos voltar ao problema do João e resolvê-lo.

Faremos o algoritmo dentro da sua estrutura correta com comentários indicando o que fazem as principais instruções.

```
1  Algoritmo Soma_dois_valores;
2  { Abre a área para declaração de variáveis }
3  var
4        { Declara três variáveis para armazenar valores do
5  tipo real }
6        Valor1, Valor2, Soma : Real;
7
8  Início
9        { Pede e armazena os dois valores que o usuário deseja
10 somar }
11       Escreva('Informe o primeiro valor: ');
12       Leia(Valor1);
13       Escreva('Informe o segundo valor: ');
14       Leia(Valor2);
15
16       { Após armazenados os dois valores, faremos a soma de-
17 les e armazenaremos o resultado na variável Soma}
18       Soma ← Valor1 + Valor2;
19
20       { Agora que temos o resultado armazenado, vamos exibir
21 esse resultado para o usuário }
         Escreva('A soma de ',Valor1,' + ',Valor2,' é ',Soma);
   Fim.
```

Como pudemos ver, o último **Escreva** está exibindo na tela o seguinte:

1. mensagem 'A soma de ';
2. o conteúdo da variável Valor1;

3. o caractere "+" (que está dentro de aspas, ou seja, é considerado um conteúdo texto e, assim sendo, será exibido na tela;
4. o conteúdo da variável Valor2;
5. a mensagem " é ";
6. o conteúdo da variável soma.

Supondo que o usuário informou o número 2 para Valor1 e o número 8 para Valor2, o conteúdo de Soma será o número 10. Assim, o resultado exibido pelo Escreva será:

```
A soma de 2 + 8 é 10
```

Agora que estudamos o básico para construir algoritmos, vamos tentar resolver outros problemas.

EU AJUDAREI VOCÊ A RESOLVER O PRÓXIMO PROBLEMA.

Vamos construir um algoritmo que calcule a média de três notas.

Primeiro, identificaremos, dentro das três etapas básicas de um algoritmo, o que precisaremos para resolver este problema:

Entrada
De quais dados o algoritmo vai precisar? As três notas do aluno.
Processamento
O algoritmo deverá calcular a média**.**
Saída
O algoritmo exibirá a média**.**

Segue o algoritmo:

```
1  Algoritmo Calcula_Media;
2  var
3        { Armazenam-se as três notas e a média calculada}
4        Nota1, Nota2, Nota3, Media : Real;
5
6  Início
7        Escreva('Informe a nota 1: ');
8        Leia(Nota1);
9
10       Escreva('Informe a nota 2: ');
11       Leia(Nota2);
12
13       Escreva('Informe a nota 3: ');
14       Leia(Nota3);
15
16       { Agora que há três notas armazenadas, calcula-se a
17 média e armazena-se-a   na variável Media }
18       Media ← (Nota1 + Nota2 + Nota3) / 3;
19
20       Escreva('A média é ', Media);
21 Fim.
```

Vamos analisar a instrução:

```
Media ← (Nota1 + Nota2 + Nota3) / 3;
```

Colocamos a soma entre parênteses, pois os cálculos sempre serão executados seguindo a precedência matemática, ou seja, a multiplicação e a divisão sempre vão ser executadas antes da adição e subtração.

Assim, se a instrução não tivesse parênteses: `Nota1 + Nota2 + Nota3 / 3`, seria executado primeiro (`Nota3 / 3`) e o resultado seria somado com `Nota1` e `Nota2`.

2 DICAS DE PROGRAMAÇÃO

Em meio aos conteúdos que estivermos estudando, vamos intercalar algumas dicas ou conceitos importantes quando desenvolvemos algoritmos. As mesmas dicas valem para programas de computador.

2.1 ENDENTAÇÃO

Esta palavra pode ter várias formas de escrita em português. Além de "endentação", você poderá encontrar edentação, identação ou indentação. Na verdade, a palavra é uma tentativa de tradução da palavra *indent* e *indentation*, que significa "parágrafo".

A endentação consiste em utilizar espaços (como parágrafos mesmo) alinhando blocos de comandos que pertençam diretamente a uma determinada estrutura.

Este recurso é muito fácil de ser utilizado e deixa um algoritmo ou programa muito mais fácil de ser lido.

Um programa não deixará de funcionar se não utilizar endentação. Porém, quem trabalha como programador, nunca deixa de usar este recurso. Em uma empresa de *software*, por exemplo, sempre será exigido que se programe utilizando a endentação correta.

Veja, no quadro a seguir, esta representação hierárquica de blocos endentados:

```
1  Bloco1
2        Instrução 1 pertencente ao Bloco1
3        Instrução 2 pertencente ao Bloco1
4        Início {Inicia o bloco de comandos que pertencem à
5  instrução 2 }
6              Instrução A pertencente ao Bloco da Instrução 2
7              Instrução B pertencente ao Bloco da Instrução 2
8        Fim { Fim das instruções pertencentes à Instrução 2 do
9  Bloco1 }
         Instrução 3 pertencente ao Bloco1
   Fim_Bloco1
```

Veja que as instruções 1, 2 e 3 (linhas 2, 3 e 8), que pertencem ao Bloco1, estão "a um parágrafo" à frente do Bloco1.

Já as instruções A e B (linhas 5 e 6) pertencem à Instrução 2 do Bloco1, portanto, estão a um parágrafo à frente, em relação à Instrução 2.

Através do algoritmo **Calcula_Media**, vamos ver um exemplo mais prático para entender como fazer as endentações.

Começamos pela linha 4. Veja que as variáveis estão mais à frente em relação ao comando **var**. Isto porque estas variáveis pertencem à área de variáveis delimitada pelo **var**. Em outras palavras, a declaração das variáveis "está dentro" da área de variáveis, portanto, está "endentada" em relação à palavra **var**.

Note, também, que todos os comandos do algoritmo (escreva, leia, cálculos etc.) estão "endentados" em relação ao Início e Fim. Isto significa que todos estes comandos pertencem diretamente à estrutura principal do algoritmo.

Eu lhe mostrarei, mais adiante, outros exemplos da utilização de endentação.

LEITURA COMPLEMENTAR

Al-Khorezmi nunca pensou que seu apelido, que significa "um nativo de Khorezmi", seria a origem de palavras mais importantes do que ele mesmo, como álgebra, logaritmo e algoritmo. Graças a esse pouco conhecido matemático árabe do século IX, hoje temos conhecimento de conceitos tão básicos quanto o número Zero da Índia ou boa parte da matemática desenvolvida na Grécia.

E sobre algoritmos? Algoritmos e estruturas de dados formam o núcleo da ciência da computação, sendo os componentes básicos de qualquer software. Ao mesmo tempo, aprender como programar está intimamente ligado a algoritmos, já que um algoritmo é a abstração de um programa. Logo, aprender algoritmos é crucial para qualquer pessoa que deseja desenvolver software de qualidade.

Paradigmas de programação estão naturalmente associados a técnicas de projeto e disciplinas introdutórias de ciência da computação são usualmente disciplinas de introdução a algoritmos. Inicialmente, a concepção de algoritmos necessita apenas de técnicas básicas de programação. Quando a complexidade dos problemas e sua análise tornam-se importantes, como no caso deste livro, algoritmos requerem lógica, matemática discreta e alguma fundamentação teórica do tipo teoria de autômatos e linguagens formais.

Entretanto, aprender algoritmos não é fácil, uma vez que precisamos ter a combinação correta de conhecimentos matemáticos e de bom senso. Citando Knuth, a melhor teoria é inspirada na prática e a melhor prática é inspirada na teoria. O balanceamento entre teoria e prática é sempre uma tarefa difícil.

FONTE: ZIVIANI, Nívio. **Projeto de algoritmos**. 3. ed. São Paulo: Cengage Learning, 2010. Prefácio.

RESUMO DO TÓPICO 5

Neste tópico, você viu que:

- Vimos, neste tópico, como construir um algoritmo completo, com toda a sintaxe exigida pelo portugol.

- Através de um exemplo, aprendemos, dentro de um problema específico, a identificar o que é necessário para cada etapa (entrada, processamento e saída). Esta distinção das etapas e a identificação do que é necessário em cada uma é muito importante quando analisarmos basicamente qualquer problema a ser transformado em um algoritmo.

- Por fim, vimos o conceito de endentação. Esta prática, que consiste em utilizar espaços semelhantes a parágrafos para alinhar estruturas que pertençam a um mesmo nível hierárquico, apesar de não necessária para que um algoritmo seja executado, é de extrema importância para a legibilidade e organização das instruções. Na prática, não se fazem algoritmos sem endentação.

AUTOATIVIDADE

1 Faça um trecho de algoritmo que solicite e armazene:

a) um valor correspondente ao salário de uma pessoa;
b) um valor correspondente aos benefícios de saúde que a pessoa recebe;
c) um valor correspondente ao vale-transporte.

Você precisa exibir qual é o valor total que a pessoa recebeu, entre salário e benefícios.

2 Desenvolva um algoritmo que leia um valor correspondente a uma temperatura em Fahrenheit, converta para graus centígrados e exiba a temperatura convertida. Para converter, basta utilizar a fórmula C = (F – 32) / 1,8, onde "f" é a temperatura lida.

3 Construa um algoritmo que solicite e armazene o preço de venda de uma mercadoria e o valor pago pelo cliente. O algoritmo deverá calcular e exibir o troco a ser entregue ao cliente.

4 Desenvolva um algoritmo que solicite e armazene o preço de custo de uma mercadoria, calcule e mostre o preço de venda. Este preço é obtido acrescentando 28% ao preço de custo.

5 Faça um algoritmo que solicite e armazene o salário de uma pessoa e um valor percentual. O algoritmo deve aplicar o percentual sobre o salário e exibir o salário atualizado.

6 Crie um algoritmo que solicite e armazene:

a) a uma distância percorrida (em km);
b) o tempo, em horas, que durou para percorrer esta distância.

O algoritmo deverá informar qual foi a velocidade média utilizada ao percorrer o período.

7 A fórmula para calcular a área de uma circunferência é $A = \pi R^2$. Sendo o valor de $\pi = 3.14159$, faça um algoritmo que:

a) solicite e armazene o valor do raio de uma circunferência, no caso a variável R;
b) efetue o cálculo da área, elevando o valor de R ao quadrado e multiplicando por π;
c) apresente o valor da variável A.

8 Faça um algoritmo que leia um valor correspondente a uma quantidade de horas (por exemplo: 1,5 corresponde a uma hora e meia). O algoritmo deverá exibir o equivalente em minutos.

UNIDADE 2

ESTRUTURAS AVANÇADAS DE ALGORITMOS

OBJETIVOS DE APRENDIZAGEM

Caro(a) acadêmico(a), o objetivo desta unidade é:

- demonstrar estruturas mais avançadas de algoritmos para resolver problemas mais complexos;
- praticar estes problemas mais complexos para exercitar a lógica através dos algoritmos.

PLANO DE ESTUDOS

Esta unidade está dividida em seis tópicos. Ao final de cada tópico, você encontrará atividades práticas para resolver problemas de lógica com certa complexidade através de estruturas de algoritmos.

TÓPICO 1 – ESTRUTURAS DE SELEÇÃO

TÓPICO 2 – ESTRUTURAS DE REPETIÇÃO

TÓPICO 3 – DICAS DE PROGRAMAÇÃO

TÓPICO 4 – VETORES

TÓPICO 5 – MATRIZES

TÓPICO 6 – SUBALGORITMOS

TÓPICO 1

ESTRUTURAS DE SELEÇÃO

1 INTRODUÇÃO

Em muitos casos, dependendo de uma determinada situação, pode-se desejar executar uma ou mais instruções e, dependendo da situação, pode-se desejar executar outras instruções.

Para permitir que um algoritmo siga caminhos diferentes dependendo de determinadas condições, utilizamos as estruturas de seleção.

Conforme Kochanski e Andrietti (2005, p. 49), "[...] a estrutura de seleção permite a definição de uma condição para que um conjunto de instruções seja ou não executado. Por este motivo, as estruturas de seleção também são conhecidas como estruturas condicionais".

2 ESTRUTURA DE SELEÇÃO "SE-ENTÃO"

Vamos iniciar com um exemplo: após calcular a média das notas de um aluno, queremos informar se ele está aprovado ou reprovado. Vamos definir uma regra: se a nota for 6,5 ou mais, a pessoa está aprovada, caso contrário, está reprovada.

Podemos perceber que, quando definimos essa regra, utilizamos a palavra "se": **se** a média for 6,5 ou mais.

Justamente esta palavra é o comando de seleção. Veja qual é a estrutura deste comando:

```
Se <condição> então
Início
       <Instruções>
Fim
```

<condição> é uma situação ou regra definida. Nas condições, utilizaremos, em vários casos, os operadores relacionais e lógicos vistos anteriormente. Por exemplo, como vamos representar a condição "média é 6,5 ou mais" na pseudolinguagem? Para facilitar a construção desta condição, basta "falar a condição de uma forma diferente": média é maior ou igual a 6,5. Pronto!

Não existe o operador relacional "maior ou igual"? Basta utilizá-lo, ficando assim: (media >= 6,5). Neste caso, "media" é a variável que contém a média armazenada do aluno.

O "`então`" faz parte da estrutura "`se`", veja:

```
Se Media >= 6,5 então
```

"o aluno está aprovado".

Você percebeu que, na estrutura "se <condição> então", há os comandos "Início" e "Fim". Este "Início" e "Fim" são utilizados para delimitar o conjunto de instruções que será executado caso a condição seja verdadeira.

Vejamos como fica a estrutura adaptando-a para a nossa solução:

```
Se Media >= 6,5 então
Início
       Escreva('O aluno está aprovado com média: ', Media);
Fim
```

Vamos interpretar o que irá acontecer neste trecho de código-fonte através de um exemplo:

Digamos que as três notas do aluno sejam:
Nota1 ← 8,5;
Nota2 ← 7,0;
Nota3 ← 9,5;

A média será 8,33, ou seja, o valor da variável Media é 8,33.

Assim, Media é maior do que 6,5. Resulta que o **resultado** da condição Media >= 6,5 é **verdadeiro**. Se for verdadeiro, o algoritmo irá executar o conjunto de instruções que estiver logo depois do **então**. Neste caso, será exibida a mensagem "O aluno está aprovado com média: 8,33".

Se a condição fosse falsa, a execução pularia para a próxima linha depois do "Fim", ou seja, não seria executada a instrução `Escreva('O aluno está aprovado com média: ',Media);`

2.1 SELEÇÃO COMPOSTA

Mas falta alguma coisa, não? E se a média for menor do que 6,5? Neste caso, queremos informar que o aluno está reprovado. Para isto, existe a estrutura **Seleção Composta**, a qual é resolvida utilizando-se o comando **Senão**, que auxilia o comando **Se-então**.

Quando a condição de um **Se-então** for falsa, como já vimos, o bloco de instruções que está logo após o **Então** não será executado e a execução passará para a primeira instrução após este bloco.

Se adicionarmos o comando **Senão** logo após este bloco, quando a condição do Se-então anterior a este **Senão** for **falsa**, o bloco de instruções que estiver junto ao **Senão** será obrigatoriamente executado.

Vejamos:

```
Se Media >= 6,5 então
Início
       Escreva('O aluno está aprovado com média: ',Media);
Fim
Senão
Início
       Escreva('O aluno está reprovado com média: ',Media);
Fim
```

Suponhamos que a média armazenada na variável Media seja 5,5. Quando a execução chegar ao comando **Se**, será verificada a condição Media >= 6,5.

Como esta condição é **falsa**, ou seja, 5,5 não é maior ou igual a 6,5, a execução pulará para a próxima linha após o fim do **Se**. Nesta linha está o comando **Senão**. Sempre que a execução chegar a um comando **Senão**, o bloco de instruções que estiver junto ao **Senão** será executado.

Em outras palavras, sempre que a condição de um **Se-então** for **falsa** e houver um **Senão** junto à estrutura do **Se-então**, as instruções do **Senão** serão executadas.

2.2 SELEÇÃO ENCADEADA

No exemplo anterior, nós tínhamos somente duas condições: Aprovado e Reprovado. Quando isto acontece, apenas uma estrutura de **seleção composta**, ou seja, um **Se-então** com um **Senão** resolve o problema.

Porém, quando há mais de duas situações, por exemplo, aprovado, em exame e reprovado, é necessário utilizar uma **Seleção Encadeada**, que nada mais é que um **Se-então** subordinado a outro **Se-então**.

Vamos assumir que, se a média for 6,5 ou mais, o aluno está aprovado. Se a média for 5,0 ou mais (mas menor do que 6,5), o aluno está em exame e se for menor do que 5,0, está reprovado.

Como ficaria a estrutura:

```
1   Algoritmo Calcula_Media;
2   var
3         { Variáveis para armazenar três notas e a média calculada}
4         Nota1, Nota2, Nota3, Media : Real;
5
6   Início
7         Escreva('Informe a nota 1: ');
8         Leia(Nota1);
9
10        Escreva('Informe a nota 2: ');
11        Leia(Nota2);
12
13        Escreva('Informe a nota 3: ');
14        Leia(Nota3);
15
16        { Agora que há três notas armazenadas, calcula-se a média e
17  armazena-se-a na variável Media }
18        Media ← (Nota1 + Nota2 + Nota3) / 3;
19
20        Se Media >= 6,5 então
21        Início
22              Escreva('O aluno está aprovado com média: ',Media);
23        Fim
24        Senão Se Media >= 5 então
25        Início
26              Escreva('O aluno está em exame com média: ',Media);
27        Fim
28        Senão
29        Início
30              Escreva('O aluno está reprovado com média: ',Media);
31        Fim
32  Fim.
```

Sempre que um bloco de instruções dentro de uma estrutura de seleção for executado, todos os blocos abaixo do bloco executado não serão executados. Exemplo: se a média for 8,0 (maior do que 6,5), será executado o bloco das linhas 21 a 23. Depois de executar este bloco, a execução do algoritmo passará para a linha 32, ou seja, os blocos das linhas 24 a 27 e das linhas 28 a 31 serão "pulados" automaticamente.
Isto acontece devido ao "**Senão**". É este comando que faz um encadeamento entre cada **Se-Então**, fazendo com que, ao entrar em uma estrutura, já salte pelas demais.

Vamos interpretar a estrutura:

a) Se a média for 8.

Quando a execução chegar em **Se** Media >= 6,5 **então**, a condição será verdadeira (8 > 6,5). Assim, será executada a instrução logo após o **então** e as estruturas abaixo, como já mencionado, não serão executadas.

b) Se a média for 5,5.

Quando a execução chegar em **Se** Media >= 6,5 **então**, a condição será falsa (5,5 > 6,5). A execução pulará para o primeiro **Senão** após esta estrutura **Se-então** que acaba de ser verificada.

A execução cairá na linha **`Senão Se`** `Media >= 5` **`então`**. A condição Media >= 5 será testada e o resultado será verdadeiro (5,5 > 5). Será, então, executado o bloco de instruções logo após o **então**, exibindo a mensagem `'O aluno está em exame com média: 5,5`.

c) Se a média for 4,0.

Quando a execução chegar em **Se** Media >= 6,5 **então**, a condição será falsa (4 > 6,5). A execução pulará para o primeiro **Senão** (se existir) após esta estrutura **Se-então** que acaba de ser verificada.

A execução cairá na linha **`Senão Se`** `Media >= 5` **`então`**. A condição Media >= 5 será testada e o resultado será falso (4 > 5). A execução pulará, mais uma vez, para o próximo **Senão** (se existir). Este último **Senão** não tem mais nenhuma condição a ser analisada (ou seja, não há mais nenhum **Se-então** junto a ela). Assim sendo, será executado o bloco de instruções junto ao último **Senão**.

3 ESTRUTURA ESCOLHA-CASO

Mais simples, porém mais restrito que a estrutura **Se-Então**, o **Escolha-Caso** (também chamado de estrutura de **seleção de múltipla escolha**) é utilizado quando há necessidade de tomar "caminhos diferentes" de acordo os possíveis valores de um único identificador (geralmente, uma variável). Vamos à sintaxe:

```
Escolha (<identificador>)
       caso <valor_1> :
              Início
                     Bloco de instruções
              Fim
       caso <valor_2> :
              Início
                     Bloco de instruções
              Fim
       caso <valor_3> :
              Início
                     Bloco de instruções
              Fim
       caso <valor_N> :
              Início
                     Bloco de instruções
              Fim
       senão :
              Início
                     Bloco de instruções
              Fim
Fim
```

Vamos aproveitar a estrutura para reforçar o conceito de endentação. Veja que cada comando "caso" está a um parágrafo em relação ao "Escolha", pois cada "caso" pertence diretamente ao "Escolha". Veja, também, que cada "bloco de instruções" ficará a um parágrafo em relação ao "Início" do "Caso". Isto porque as instruções sempre pertencerão aos "casos", ficando assim um parágrafo em relação à estrutura direta à qual pertence.

Algumas características da estrutura **Escolha-Caso**:

- A variável escolhida pode ser somente dos tipos **caractere** e **inteiro**.
- Se tiver somente uma instrução dentro de um **caso**, não é necessário utilizar **Início** e **Fim** para este **caso**.
- Se desejar executar uma ou mais instruções quando o valor da variável escolhida não for nenhum dos valores tratados, utiliza-se o **Senão**.

Vamos entender melhor através de um exemplo prático:

Faremos um algoritmo que lê o preço unitário de uma mercadoria e a quantidade comprada desta mercadoria por um cliente.

De acordo com a quantidade comprada, haverá um desconto, conforme o quadro a seguir:

Quantidade comprada	**Desconto**
1 unidade	5%
2 unidades	8%
3 unidades	10%
Acima de três unidades	13%

Ao final, o algoritmo deverá apresentar o valor a ser pago.

Vamos à resolução:

```
1   Algoritmo EscolhaCaso;
2
3   var
4       ValorUnitario, Desconto, ValorPagar, ValorTotalSemDesconto,
5       ValorDoDesconto : Real;
6       QtdComprada : Inteiro;
7
8   Início
9
10        Escreva('Informe o valor unitário da mercadoria: ');
11        Leia(ValorUnitario);
12
13        Escreva('Informe a quantidade comprada: ');
14        Leia(QtdComprada);
15
16        Escolha (QtdComprada)
17              caso 1 : Desconto ← 5;
18              caso 2 : Desconto ← 8;
19              caso 3 : Desconto ← 10;
20              Senão Desconto ← 13
21        Fim;
22
23        { Multiplica o valor unitário pela quantidade comprada para
24  obter o valor total a pagar}
25        ValorTotalSemDesconto ← ValorUnitario * QtdComprada;
26
27        { Calcula somente o valor a ser dado de desconto }
28        ValorDoDesconto ← ValorTotalSemDesconto * (Desconto / 100);
29
30        { Do valor total, diminui o valor a ser descontado, resul-
31  tado     no valor a pagar }
32        ValorPagar ← ValorTotalSemDesconto - ValorDoDesconto;
33
34        Escreva('Valor a pagar: ', ValorPagar);
35  Fim.
```

Interpretando:

O algoritmo solicita e armazena os valores correspondentes à quantidade a ser comprada e o valor unitário.

Obtendo a informação da quantidade comprada, verifica qual é essa quantidade e armazena na variável **Desconto** o valor de desconto a ser aplicado.

Podemos perceber que não há **Início/Fim** nos blocos **caso**. Isto porque, conforme já citado anteriormente, há apenas uma instrução dentro de cada bloco **caso**.

Na linha 25, calculamos e armazenamos o valor total a pagar.

Na linha 28, calculamos o valor a ser dado de desconto, aplicando o desconto selecionado através da estrutura **Escolha-Caso** sobre o valor total a pagar.

O valor final a pagar é o valor total, menos o valor de desconto. Este cálculo é feito na linha 32.

Logo a seguir, estamos exibindo o valor que realmente será pago.

RESUMO DO TÓPICO 1

Neste tópico, você viu que:

- Como vimos, o **Se-então** e o **Escolha-Caso** são estruturas de seleção.

- Se pararmos para observar, concluímos que tudo o que se faz com uma estrutura **Escolha-Caso** pode ser feito com uma estrutura **Se-então**. Mas a recíproca não é verdadeira, ou seja, nem tudo o que fazemos com o **Se-então** podemos fazer com o **Escolha-Caso**.

- Quando usar cada estrutura, então? Tudo depende da situação. Quando precisamos executar blocos diferentes de instruções dependendo apenas de diferentes valores que uma variável pode assumir, damos preferência para o **Escolha-Caso**, mesmo que o **Se-então** consiga resolver esta situação.

- Para qualquer outra situação, é necessário utilizar o **Se-então**.

AUTOATIVIDADE

1 Reescreva o mesmo algoritmo de conversão de temperaturas, adaptando-o da seguinte forma: além de ler a temperatura, o algoritmo deverá ler se o usuário quer converter de Fahrenheit para centígrados ou de centígrados para Fahrenheit. A fórmula para converter graus centígrados para Fahrenheit é F = (C * 1,8) + 32.

2 Escreva um algoritmo que solicite e armazene três valores. O algoritmo deverá calcular e exibir o maior dos três valores.

3 Faça um algoritmo que leia um número e informe se ele é menor que zero, se ele é maior que zero ou se ele é o próprio valor zero.

4 Desenvolva um algoritmo que leia um número inteiro e informe se o número é par ou é ímpar.

5 Faça um algoritmo que leia um número inteiro de zero a nove e exiba o valor por extenso.

6 Desenvolva um algoritmo que solicite e armazene dois valores numéricos reais. Em seguida, o algoritmo deverá exibir as seguintes opções:

+ Adição
- Subtração
* Multiplicação
/ divisão

O algoritmo, após exibir as opções acima, deverá ler a escolha do usuário e efetuar o cálculo entre os dois valores lidos, de acordo com a escolha do usuário. Ao final, o algoritmo deverá exibir o resultado do cálculo.

7 Construa um algoritmo que leia o salário de um funcionário. O algoritmo deverá calcular um aumento no salário de acordo com o quadro a seguir:

Se o salário for:	Acrescentar:
Menor que R$ 800,00	40%
Entre R$ 800,00 e R$ 1.000,00	35%
Entre R$ 1.001,00 e R$ 1.500,00	25%
Acima de R$ 1.500,00	15%

TÓPICO 2

ESTRUTURAS DE REPETIÇÃO

1 INTRODUÇÃO

Em muitos casos, é necessário repetir uma instrução ou um mesmo conjunto de instruções mais de uma vez. Para melhor entender, vamos começar com um exemplo sem repetição: Um algoritmo precisa solicitar e armazenar o nome e a idade de uma pessoa. Como você faria este algoritmo?

```
1  Algoritmo Exemplo;
2  var
3         Nome : Caractere;
4         Idade : Inteiro;
5  Início
6         Escreva('Informe seu nome: ');
7         Leia(Nome);
8         Escreva('Informe sua idade: ');
9         Leia(Idade);
10        { Demais instruções }
11 Fim.
```

Agora, e se desejarmos fazer a mesma coisa, mas para 3 pessoas?

Ah, poderíamos copiar e colar as instruções, não é mesmo? Desta forma:

```
1  Algoritmo Exemplo;
2  var
3         Nome : Caractere;
4         Idade : Inteiro;
5  Início
6         Escreva('Informe seu nome: ');
7         Leia(Nome);
8         Escreva('Informe sua idade: ');
9         Leia(Idade);
10
11        Escreva('Informe seu nome: ');
12        Leia(Nome);
13        Escreva('Informe sua idade: ');
14        Leia(Idade);
15
16        Escreva('Informe seu nome: ');
17        Leia(Nome);
18        Escreva('Informe sua idade: ');
19        Leia(Idade);
20
21        { Demais instruções }
22 Fim.
```

Certo, até funcionaria. Mas, e se precisássemos fazer isto para 50 pessoas? Ou ainda, se não soubéssemos a quantidade de pessoas? Seria inviável, não é mesmo?

Para este tipo de situação, existe uma solução: as **estruturas de repetição**, também conhecidas como laços de repetição ou *looping*.

Talvez você já tenha ouvido a expressão "entrou em *looping*". Essa expressão significa que uma estrutura de repetição entrou numa situação que nunca mais irá parar de repetir. Neste caso, quando se trata de um programa de computador, somente fechando o programa para parar com a estrutura. Mais adiante, veremos com mais detalhes o conceito de *looping*.

Estas estruturas repetirão um bloco de instruções até que uma determinada **condição** seja atendida. **Condição**, nós já sabemos o que é. É uma expressão que resultará sempre em verdadeiro ou falso, exatamente como são usadas na estrutura **Se-Então**.

2 ENQUANTO-FAÇA

A primeira estrutura que iremos estudar é a **Enquanto-faça**. O próprio nome da estrutura já sugere: **Enquanto** uma determinada condição for **verdadeira**, **faça** executar um bloco de instruções.

Qual é a sintaxe:

```
Enquanto (<condição>) faça
Início
      <Instruções que serão executadas se a condição for verdadeira>
Fim
```

Como a estrutura funciona:

Quando a execução chegar à linha **Enquanto** <condição> **faça**, será verificado se a <condição> é verdadeira. Se a condição for verdadeira, será executado o bloco de instruções que estiver após o faça, entre os comandos `Início` e `Fim`.

Veja um dos conceitos mais importantes a saber sobre a estrutura do **Enquanto-faça**: toda vez que a execução chegar ao **Fim** do **Enquanto-faça**, ela voltará para o início da própria estrutura, ou seja, para a linha **Enquanto** <condição> **faça** e verificará novamente se a condição é verdadeira. Se for, entra no **Enquanto-faça** e executa as instruções novamente. Isto vai se repetir até o momento em que a execução voltar para a linha que verifica a condição do **Enquanto-faça** e esta condição for falsa. Quando isso acontecer, a execução continuará na linha após o **Fim** do **Enquanto-faça**.

Vamos ver um exemplo em que uma estrutura irá se repetir até que o usuário diga que não quer mais continuar:

```
1  Algoritmo EnquantoFaca;
2  var
3         continuar : Caractere;
4  Início
5         Escreva('Informe "s" para continuar e outro caractere para
6  não    continuar: ');
7         Leia(continuar);
8
9         Enquanto (continuar = 's') faça
10        Início
11                Escreva('Você quis continuar!');
12
13                Escreva('Informe "s" para continuar: ');
14                Leia(continuar);
15        Fim
16        Escreva('Você não quis mais continuar!');
17 Fim.
```

Ao iniciar a execução, será exibida uma mensagem indicando ao usuário que, se ele deseja continuar, deve informar a letra 's'. A resposta é armazenada na variável continuar.

Se a variável **continuar** tiver 's', a condição do **Enquanto-faça** será verdadeira, pois está verificando se o conteúdo da variável continuar é igual a 's'. Neste caso, será executado tudo o que estiver entre o **Início** e **Fim** da estrutura. Ou seja, vai aparecer a mensagem "Você quis continuar!" e, em seguida, na linha 13, é exibida a mensagem 'Informe "s" para continuar: '. A linha 9 aguarda o usuário informar novamente se deseja continuar.

Ao executar a linha 15, **Fim** do **Enquanto**, a execução volta para o início desta estrutura e verifica a condição novamente. Se a pessoa respondeu 's' na linha 14, mesmo que a execução volte ao início, o conteúdo da variável **continuar** não se altera. Assim, a condição será verdadeira e a execução entra no **Enquanto-faça** novamente.

Desta forma, sempre que a pessoa responder 's' (sim), a execução repetirá. A partir do momento que a pessoa informar qualquer coisa diferente de 's', quando o **Enquanto-faça** for verificar a condição, teremos uma condição **falsa**. Suponhamos que o usuário informou 'x'. O conteúdo da variável **continuar**, que agora é 'x', é diferente de 's', fazendo com que a execução saia do **Enquanto-faça**, partindo para a linha 16.

Neste momento, aparecerá a mensagem 'Você não quis mais continuar!'.

Vamos testar mais um exemplo. Digamos que, já na execução da linha 7, o usuário tenha informado algo diferente de 's'. Quando a condição for avaliada, na linha 9, esta será falsa, e a execução não entrará nenhuma vez sequer no **Enquanto**. Já na primeira vez, a execução exibirá a mensagem 'Você não quis mais continuar!'.

Costuma-se chamar a condição para parar uma estrutura de repetição, de *FLAG* (pronuncia-se flég). No exemplo anterior, a *flag* é "continuar diferente de 's' ", ou seja, quando a variável "continuar" for diferente de "s", a estrutura irá parar. Reforçando: "Enquanto continuar for igual a 's', a estrutura continuará executando".

3 PARA-FAÇA

Esta estrutura é recomendada quando se sabe a quantidade de vezes que a estrutura deverá repetir. Sempre será definido um limite inferior e outro superior e uma variável que passará por cada valor dentro destes dois limites.

A estrutura é:

```
Para <variável> ← <limite inferior> até <limite superior> faça
Início
       <instruções>
Fim
```

<variável> é uma variável qualquer, do tipo inteiro, que irá receber cada valor dentro dos limites.

Vamos ver um exemplo:

```
Para i ← 1 até 10 faça
Início
       Escreva(i);
Fim
```

Quando a execução chega ao **Para-faça**, é atribuído o limite inferior à variável, neste caso, "**i**". Após esta atribuição, será executado o bloco de instruções pertencentes ao **Para-faça**. No exemplo, somente será escrito o valor da variável "**i**".

Ao ir para a próxima linha, ou seja, o **Fim**, por se tratar de uma estrutura de repetição, a execução volta ao início do **Para**. Nesta volta, o "**i**" é automaticamente incrementado em 1 pelo **Para-faça**, ou seja, automaticamente o "**i**" passa a ser 2.

Esta ação de chegar ao fim, voltar e incrementar a variável de controle se repetirá até que o valor desta variável ultrapasse o limite superior.

Trabalharemos mais com a estrutura **Para-faça** na próxima unidade, onde veremos os conceitos de vetores e matrizes.

4 REPITA-ATÉ

Como já vimos, se, na primeira vez que a execução chegar a uma estrutura de repetição do tipo **Enquanto-faça**, a condição for falsa, nenhuma instrução pertencente ao **Enquanto-faça** será executada.

Ao contrário do **Enquanto-faça**, o **Repita-até** testa a condição somente no fim. Por isto, a estrutura **Repita-até** executa, pelo menos uma vez, as instruções que estiverem dentro dela.

```
Repita
       Escreva('Idade: ');
       Leia(Idade);
       Escreva('A idade informada é: ', Idade);
Até (Idade = 0);
```

A estrutura irá repetir até que a pessoa informe "0" (zero) para a idade.

Mesmo que seja informado zero da primeira vez, as instruções serão executadas esta vez.

Vimos três estruturas de repetição. Qual delas usar? Bom, isto você terá que definir qual é a estrutura que melhor se adapta a cada problema. Mas não se preocupe. Com a prática, você começará a identificar onde e quando utilizar cada estrutura.

RESUMO DO TÓPICO 2

Neste tópico, você viu que:

- Sempre que precisarmos repetir um mesmo conjunto de instruções por várias vezes, utilizaremos as chamadas estruturas de repetição.

- Vimos a estrutura **Enquanto-faça**. Ela permite que se repitam instruções infinitamente ou até que o usuário deseje parar de executar (baseando-se em uma condição, chamada de *flag*).

- Também dispomos da estrutura **Para-faça**. Um pouco menos versátil do que a **Enquanto-faça**, porém muito útil em determinadas situações, é uma estrutura que repete as instruções dentro de uma quantidade predeterminada de vezes. Define-se um intervalo inicial e um intervalo final, e a estrutura faz com que uma determinada variável passe por cada um dos valores dentro deste intervalo.

- Existe, também, a estrutura **Repita-até**, cuja característica principal é que, pelo menos uma vez, as instruções contidas nela serão executadas, o que já não acontece necessariamente com a estrutura **Enquanto-faça**, por exemplo, na qual pode acontecer de a execução nunca passar pelas instruções da estrutura.

AUTOATIVIDADE

1 Quando é necessário utilizar uma estrutura de repetição?

2 Quando se dá preferência por utilizar a estrutura Para-faça ao invés de Enquanto-faça?

3 Explique o que é *flag* e dê um exemplo.

4 Crie um algoritmo que leia o salário de 8 pessoas. O algoritmo deverá informar:

a) quantas pessoas ganham mais que 1.000 reais;
b) a média salarial entre as 8 pessoas.

5 Faça um algoritmo que possa imprimir o menu abaixo:

MENU ESTADO CIVIL
1 – Solteiro(a)
2 – Desquitado(a)
3 – Casado(a)
4 – Divorciado(a)
5 – Viúvo(a)

Opção:
a) O usuário deverá selecionar uma das opções, digitando o número correspondente.
b) Emitir mensagem de erro caso seja escolhida uma opção inexistente.
c) Ao finalizar, o algoritmo deverá apresentar a porcentagem de cada estado civil.

6 Faça um algoritmo que leia um número inteiro. O algoritmo deverá exibir se o número é primo ou não.

Conceito de um número primo: Um número natural é um número primo quando ele tem exatamente dois divisores distintos: o número um e ele mesmo. Por convenção, os números 0, 1 e -1 não são considerados primos nem compostos.

7 Faça um algoritmo que leia um número e calcule o seu fatorial. O algoritmo deverá apresentar uma mensagem semelhante a: O fatorial de X é Y.

O fatorial de um número é calculado multiplicando o número por todos os seus antecessores até chegar em 1. Exemplo: Se o número for 5, o cálculo será 5 * 4 * 3 * 2 * 1 = 120.
Assim, o fatorial de 5 é 120.

8 Desenvolva um algoritmo para calcular o Índice de Massa Corpórea (IMC) de várias pessoas. O usuário deverá informar seu peso e altura. O IMC é calculado dividindo-se o peso pela altura ao quadrado. O algoritmo deverá exibir a situação da pessoa de acordo com o quadro logo a seguir. O algoritmo deverá permanecer solicitando os dados das pessoas e informando sua situação até que seja informada a altura igual a 0 (zero).

Quadro de situações segundo a faixa de peso:

Cálculo IMC	Situação
Abaixo de 18,5	Você está abaixo do peso ideal
Entre 18,5 e 24,9	Parabéns — você está em seu peso normal!
Entre 25,0 e 29,9	Você está acima de seu peso (sobrepeso)
Entre 30,0 e 34,9	Obesidade grau I
Entre 35,0 e 39,9	Obesidade grau II
40,0 e acima	Obesidade grau III

9 Escreva um algoritmo que solicite o preço unitário e a quantidade de uma mercadoria a ser vendida. O algoritmo deverá permanecer solicitando estes valores até que o preço seja 0 (zero). A cada vez que forem lidas as informações, o algoritmo deverá verificar se a mercadoria foi vendida em uma quantidade maior do que cinco. Caso tenha sido, deverá ser dado um desconto de 15% no valor total da compra.

Ao final, o algoritmo deverá informar quanto deixou de ser arrecadado em função dos descontos oferecidos.

TÓPICO 3

DICAS DE PROGRAMAÇÃO

1 INTRODUÇÃO

Concluídas as estruturas básicas principais de um algoritmo, veremos agora mais algumas dicas básicas, porém importantíssimas em se tratando de algoritmos e futuramente de um programa de computador. Estas dicas são técnicas básicas de programação, utilizadas em várias situações.

2 CONTADORES

É bastante comum, quando utilizamos estruturas de repetição, precisarmos contar quantas vezes uma instrução irá executar (repetir).

Por exemplo, no exercício anterior, não sabemos quantas vezes o usuário informou 's' e, consequentemente, quantas vezes o algoritmo repetiu. Mas é possível calcular para saber esta informação.

Em termos de instrução, um contador consiste em atribuir a uma variável o valor que ela já tem armazenado no momento, somado a outro valor, cuja soma resulta em aumentar seu valor, simulando uma contagem. A sintaxe padrão é:

```
Variavel ← Variavel + <valor>;
```

Sendo <valor>, um número que está sendo adicionado ao valor de **Variavel**. Geralmente, o valor é 1, pois, na maioria das vezes, precisamos contar de 1 em 1. Porém, nada impede de contar de 2 em 2, de 3 em 3 e assim por diante, tudo depende da necessidade.

Vamos interpretar a seguinte instrução:

```
Contador ← Contador + 1;
```

Pode parecer estranho uma variável receber “ela mesma”, mas vamos ver um exemplo:

Suponhamos que o valor da variável **Contador**, no momento, seja 8.

Na instrução, estamos armazenando na variável **Contador** o resultado do cálculo **Contador + 1**. O resultado deste cálculo será 9 (**Contador**, que contém 8, mais 1). Este resultado será armazenado na variável **Contador** pelo sinal de atribuição, ou seja, **`Contador ← <resultado do cálculo>`** (**Contador** recebe <resultado do cálculo>).

Após a execução desta instrução, o valor de **Contador** passa, então, a ser 9, pois recebeu o resultado de 8 + 1.

Se esta instrução estiver dentro de uma estrutura de repetição, poderá acontecer de ela ser executada novamente. Se isto ocorrer, agora que o valor é 9, vamos ver o que acontecerá:

A variável **Contador** irá receber, novamente, o resultado da soma entre o conteúdo que ela tem armazenada no momento, mais 1 (**Contador** + 1). Agora, o resultado desta soma é 10. O 10 será armazenado em **Contador** pela atribuição.

Como podemos perceber, cada vez que a instrução for executada, a variável **Contador** aumentará seu valor em 1.

Ao terminar a execução da estrutura de repetição, a variável, que está sendo utilizada como contador, estará armazenando a quantidade de vezes que a estrutura se repetiu.

3 INICIALIZAÇÃO

Este é um conceito muito importante utilizado em várias situações. Em contadores, por exemplo, é essencial.

Como o próprio nome sugere, significa atribuir um valor **inicial** a uma variável, geralmente, no início do algoritmo.

Vamos voltar à instrução `Contador ← Contador + 1;`

Na primeira vez que esta instrução é executada, **Contador** receberá "que valor" + 1? É preciso garantir que a variável **Contador** sempre contenha um valor, pois, quando for executado o cálculo **Contador** + 1, a variável não pode estar "vazia".

Por isso, então, é preciso atribuir um valor inicial à variável. Este valor precisa ser "neutro", ou seja, não pode influenciar em nada na contagem real das vezes que uma instrução se repete.

No caso de uma soma, o valor neutro é 0 (zero), pois, qualquer valor somado com zero, resulta no próprio valor, ou seja, não altera o resultado.

Vamos ver uma estrutura que faça a contagem de quantas vezes o usuário respondeu que quer repetir a execução:

```
1  Algoritmo Exemplo_Contador;
2  var
3        Contador : Inteiro;
4        Resposta : Caractere;
5
6  Início
7        Contador ← 0; {Inicialização – atribui um valor ini-
8  cial, neste caso, zero}
9
10       Escreva('Deseja informar valores? ');
11       Leia(Resposta);
12
13       Enquanto (Resposta = 's') faça
14       Início
15
16             { Ação de contagem }
17             Contador ← Contador + 1;
18
19             Escreva('Deseja informar valores? ');
20             Leia(Resposta);
21       Fim
22       Escreva('A estrutura executou ',Contador,' vez(es).');
23 Fim.
```

Vamos simular duas situações:

1 Se for informado algo diferente de 's' na linha 11

A condição na linha 13 será falsa, então a execução saltará para a linha 22. Como o valor de **Contador** será 0 (zero), pois recebeu esse valor na linha 7 e, depois disso, seu valor não foi mais alterado, a mensagem exibida na linha 22 será "A estrutura executou 0 vez(es)".

E esta mensagem está certa, não é mesmo? A estrutura de repetição realmente não executou nenhuma vez.

2 Se for informado 's' na linha 11

Neste caso, a condição na linha 13 será verdadeira. A próxima instrução a ser executada será a linha 17. **Contador** receberá o resultado de 0 (zero, valor de **Contador** neste momento) mais 1. Agora, o valor de **Contador** passa a ser 1.

As linhas 19 e 20 serão executadas. Após isto, ao chegar à linha 21, a execução voltará para a linha 13.

2.1 Se o usuário informar algo diferente de 's' na linha 20

A condição será falsa e a execução saltará para a linha 22. A mensagem exibida será "A estrutura executou 1 vez(es)". A mensagem também condiz com a realidade: a execução realmente entrou somente uma vez na estrutura de repetição.

2.2 Se o usuário informar 's' na linha 20

A condição será verdadeira, entrando mais uma vez na estrutura de repetição e executando a linha 17 outra vez. Agora, **Contador** passa a valer 2.

Se o usuário sair da estrutura, a mensagem final será: "A estrutura executou 2 vez(es)". Se o usuário desejar não sair da estrutura (informando 's' mais uma vez), a estrutura irá executar e o valor de contador irá aumentar mais uma vez, ou quantas vezes o usuário desejar. Assim, o valor da variável **Contador** sempre terá armazenada a quantidade de vezes que a estrutura repetiu.

4 ACUMULADOR

Muito semelhante ao contador, porém não necessariamente soma de 1 em 1 ou 2 em 2. O conceito do acumulador é ir adicionando ao valor (que uma variável já tem) outro valor qualquer, informado pelo usuário ou por qualquer outra fonte de dados. Uma vez pode ser 30, outra pode ser 1,8 e assim por diante.

Um exemplo prático: Queremos ler as idades de várias pessoas e calcular a média destas idades.

Para calcular a média, precisamos somar todas as idades e dividir pela quantidade de idades.

Podemos, então, fazer uma estrutura de repetição que permaneça solicitando idades até que o usuário deseje parar de informar.

Ao invés de ficar sempre exibindo uma pergunta para o usuário se ele deseja continuar a informar, neste caso, podemos usar o próprio valor informado para a idade. Ou seja, se a idade informada for 0 (zero), a estrutura de repetição pode parar.

A cada idade lida, vamos acumular a soma desta idade em uma variável e contar mais uma idade. Ao terminar a execução, teremos a soma de todas as idades e a quantidade de idades lidas.

```
1   Algoritmo Media_Idades;
2
3   var
4         {Armazenará a quantidade de idades lidas}
5         QuantasIdades,
6         {Armazenará a soma de todas as idades lidas. É o nosso
7   acumulador }
8         SomaIdades,
9         {Variável utilizada para ler uma idade}
10        Idade
11        : Inteiro; { Todas estas variáveis são do tipo inteiro}
12
13        Media : Real;
14
15  Início
16        { Inicializa o contador de idades }
17        QuantasIdades ← 0;
18        { Inicializa o acumulador }
19        SomaIdades ← 0;
20
21        { Solicita a primeira idade, que já servirá de flag na
22  condição do Enquanto}
23        Escreva('Informe a idade: ');
24        Leia(Idade);
25
26        { Se a idade for diferente de zero, acumula os valores }
27        Enquanto (Idade <> 0) faça
28        Início
29              { Conta mais uma idade }
30  QuantasIdades ← QuantasIdades + 1;
31
32        { O acumulador recebe o valor que já tem até o momento,
33        mais o valor da idade que acaba de ser lido }
34        SomaIdades ← SomaIdades + Idade;
35
36              Escreva('Informe a idade: ');
37              Leia(Idade);
38        Fim
39
40  {Depois que saiu do enquanto, basta dividir a soma das idades pela
41  quantidade de idades}
42        Media ← SomaIdades / QuantasIdades;
43        Escreva('A média de idades é: ', Media);
44  Fim.
```

Se, na primeira vez (linha 24), o usuário informar idade 30:

Na linha 27, a condição será verdadeira (30 é diferente de 0). Assim, a próxima linha a ser executada é a linha 30: QuantasIdades receberá 0 + 1 (valor de QuantasIdades no momento, mais 1).

Linha 34: SomaIdades receberá 0 + 30 (zero é o valor que SomaIdades tem agora, mais o valor que a variável Idade tem neste momento).

Linha 37: Vamos supor que o usuário informou 25.

A execução vai para a linha 38, retornando à 27. A condição agora é verdadeira novamente (25 é diferente de 0).

Linha 30: QuantasIdades receberá 1 + 1 (primeiro 1 é o valor que QuantasIdades tem agora, mais o valor 1).

Linha 34: Soma idades receberá 30 + 25 (30 é o valor que SomaIdades tem agora, mais 25, valor de Idade no momento).

Linha 37: Agora vamos supor que o usuário informou 0 (zero), ou seja, ele deseja finalizar.

A execução vai novamente para a linha 38, retornando à 27. A condição do **Enquanto-faça** agora é falsa (0 não é diferente de 0).

A execução passa, então, para a linha 42, que armazenará na variável **Media**, o resultado de 55 / 2. **Media** armazenará o valor 27,5.

Por fim, a linha 43 exibirá a mensagem "A média das idades é 27,5".

Lembre que, tanto no contador, quando no acumulador, precisamos inicializar as variáveis e utilizar a técnica "variável recebe ela mesma".

5 MAIOR VALOR

Em muitas situações, precisamos saber qual é o maior valor, dentre vários valores lidos pelo algoritmo.

Para esta situação, precisamos ter uma variável que sempre armazenará, a cada valor lido, o maior valor até o momento. Quando chegar ao fim do algoritmo, a variável conterá o maior valor. Vamos ver um exemplo:

```
1  Algoritmo ArmazenarMaiorValor;
2
3  var
4         MaiorValor, {Esta variável sempre guardará o maior valor}
5         Valor : Real;
6         LerMaisUm : Caractere;
7
8  Início
9
10 {Começa com 's' fazendo a condição da estrutura de repetição ser
11 verdadeira }
12        LerMaisUm ← 's';
13
14        MaiorValor ← 0; { Inicializa com 0 (zero). Veremos o porquê}
15
16
17        Enquanto (LerMaisUm = 's') faça
18        Início
19
20              Escreva('Informe o número: ');
21              Leia(Valor);
22   {Verifica agora se o valor, que acaba de ser lido, é maior do que
23 o maior valor armazenado até o momento}
24
25  Se (Valor > MaiorValor) Então
26               Início
27               { Se o valor for maior, armazena o número da variável
28  Valor na variável MaiorValor. Isto faz com que, sempre que encon-
29  trar um valor maior do que todos até o momento, este valor seja
30  armazenado }
31                     MaiorValor ← Valor;
32               Fim
33  {Se o valor lido não for maior, a execução não entrará no Se-então
34  e pulará para cá, dando continuidade}
35
36               Escreva('Deseja ler mais um valor: ');
37               Leia(LerMaisUm);
38
39         Fim
40
41         Escreva('O maior valor lido foi ', MaiorValor);
42
43  Fim.
44
```

Note que a variável **MaiorValor** foi inicializada com 0 (zero). Isto porque qualquer valor positivo, que for informado na primeira vez, será maior do que 0 (zero), fazendo com que entre no **Se-então** e sempre armazene este primeiro valor como sendo o maior. E ele é! Veja: se, na primeira vez, o usuário informar o valor **0,01**, ele é maior do que 0 (zero). Neste caso, vai entrar no **Se-então** e o algoritmo vai armazenar, na variável **MaiorValor**, o número 0,01. Mas isto está certo? Claro! Se o algoritmo parasse agora, qual seria o valor informado? Seria, realmente, 0,01. E é este o valor armazenado na variável **MaiorValor**.

6 MENOR VALOR

Da mesma forma como, às vezes, precisamos saber qual é o maior valor lido entre uma quantidade de valores, podemos precisar saber qual é o menor valor.

A principal diferença entre estas duas técnicas é que precisamos inicializar a variável, que guardar o menor valor, com o maior valor possível.

Por exemplo, se na primeira vez que executar o algoritmo, o usuário informar o número 1.000.000 (um milhão), e o algoritmo parar por aqui, este será o menor valor lido (pois é o único). Mas para isto, como já mencionado, precisamos garantir que a variável, que guardará sempre o menor valor, tenha o maior valor possível no início.

Assim, a diferença básica entre verificar qual é o menor e verificar qual é o maior é que, para verificar o maior valor, pode-se iniciar com 0 (zero) se os valores serão positivos. Para o menor valor, o ideal é inicializar com o maior valor possível, por exemplo, 9999999.

Veremos, mais adiante, que há formas melhores para inicializar este tipo de variável, em uma linguagem de programação. Porém, em um algoritmo, costuma-se utilizar o valor mencionado há pouco.

Vamos ver como ficaria um exemplo que armazena e exibe o menor valor lido.

```
1  Algoritmo ArmazenarMenorValor;
2
3  var
4         MenorValor, {Esta variável sempre guardará o menor valor}
5         Valor : Real;
6         LerMaisUm : Caractere;
7
8  Início
9
10 {Começa com 's' fazendo a condição da estrutura de repetição ser
11 verdadeira }
12        LerMaisUm ← 's';
13
14        MenorValor ← 99999;
15
16        Enquanto (LerMaisUm = 's') faça
17        Início
18
19              Escreva('Informe o número: ');
20              Leia(Valor);
21
22              { Verifica agora se o valor, que acaba de ser lido, é
23 menor do que o menor valor armazenado até o momento }
24
25              Se (Valor < MenorValor) Então
26              Início
27  {Se o valor for menor, armazena o número da variável Valor na va-
28  riável MenorValor. Isto faz com que, sempre que encontrar um valor
29  menor do que todos até o momento, este valor seja armazenado}
30                    MenorValor ← Valor;
31              Fim
32
33              { Se o valor lido não for menor, a execução não en-
34  trará no Se-então e pulará para cá, dando continuidade }
35
36              Escreva('Deseja ler mais um valor: ');
37              Leia(LerMaisUm);
38
39        Fim
40
41        Escreva('O menor valor lido foi ', MenorValor);
42
43 Fim.
44
```

Estudadas estas simples técnicas de programação, vamos seguir adiante, passando para a próxima estrutura de repetição.

7 LOOPING

Já comentamos que, quando uma estrutura de repetição entra em uma situação onde nunca mais irá parar de executar, ela entrou em *looping*.

Isto não é nenhuma técnica, mas, sim, um pequeno alerta, pois o programador deve preocupar-se em nunca deixar uma estrutura entrar em *looping*.

Vamos ver como isto poderia acontecer. Veja o seguinte algoritmo:

```
1  Algoritmo Looping;
2
3  var
4         Numero : Inteiro;
5
6  Início
7         Numero ← 0;
8
9   Repita
10                Escreva(Numero);
11          Até (Numero <> 0);
12  Fim.
```

A estrutura **Repita-até** irá repetir até que o valor da variável **Numero** seja diferente de zero. Porém, a variável **Numero** recebe zero na linha 7 e nunca mais muda de valor.

Assim, quando a execução chegar à linha 11, voltará para a linha 9. Escreverá o valor da variável **Numero** (linha 10) e, ao chegar à linha 11 novamente, como o valor de **Numero** ainda é zero, voltará para a linha 9 novamente. Isto se repetirá infinitamente, ou seja, entrou em *looping*.

RESUMO DO TÓPICO 3

Quando resolvemos problemas computacionais, existem algumas dicas muito importantes sobre programação, tais como:

- **Contadores** – Há situações nas quais precisamos saber quantas vezes uma estrutura de repetição executou, por exemplo. Para estes casos, usamos a técnica de "contador", que consiste em atribuir a uma variável o conteúdo que ela já possui até o momento, somado ao valor de contagem, que geralmente é 1. Desta forma: `contador ← contador + 1`.

- **Inicialização** – Dependendo do caso, uma variável precisa, já ao iniciar o algoritmo, armazenar um valor. Basta atribuir um valor qualquer (na inicialização, geralmente é 0) à variável. Exemplo: `x ← 0`.

- **Acumuladores** – Utilizados quando for necessário somar vários valores e, a cada valor, ir acumulando o montante desta soma em uma variável.
  Exemplo: SomaSalarios ← SomaSalarios + Salario;
  Neste exemplo, cada vez que for executada esta instrução, o valor da variável **Salario** será adicionado à soma de salários calculada até o momento e o resultado desta soma será armazenado novamente na variável acumuladora (no caso, **SomaSalarios**).

- **Maior valor** – Quando precisarmos saber qual é o maior valor dentre uma relação de valores numéricos, basta criar uma variável cujo objetivo é sempre armazenar o maior valor "até o momento". Se entrar um valor maior ainda, a variável passará a armazenar este, e assim por diante. Quando chegar no fim, o maior valor estará armazenado na variável. Para que esta técnica funcione, a variável que guarda o maior valor precisa ser inicializada com o menor valor possível, geralmente 0 (zero).

- **Menor valor** – Para saber qual é o menor valor entre uma relação de valores numéricos, a técnica é basicamente a mesma utilizada para saber qual é o maior valor. As diferenças principais são:

  - a variável, que armazena o menor valor, precisará ser iniciada com o maior valor possível;

  - quando entrar um valor, deve-se verificar se ele é menor do que o menor valor armazenado até o momento. Se for, armazena-se este.

- ***Looping*** – Em uma estrutura de repetição, quando uma *flag* nunca for satisfeita, a estrutura nunca irá parar de executar (repetir). Quando isto acontecer, dizemos que a estrutura "entrou em *looping*". Exemplo: "Enquanto (x <= 0) faça". Se a variável **x** nunca receber o valor 0 (zero) ou um valor menor do que 0 (zero), a estrutura nunca irá parar de repetir.

AUTOATIVIDADE

1 Para que servem os contadores?

2 O que é inicialização e para que serve?

3 Para que servem os acumuladores?

4 Para cada uma das técnicas a seguir, imagine um problema que necessita da técnica para ser resolvido (diferente dos problemas já apresentados no livro), descreva o problema imaginado e resolva-o utilizando algoritmos (ao todo, serão três algoritmos):

a) Contador
b) Acumulador
c) Menor valor

5 O que é "*looping*"?

6 Faça um algoritmo diferente do apresentado no livro, cuja estrutura de repetição entre em *looping*.

TÓPICO 4

VETORES

1 INTRODUÇÃO

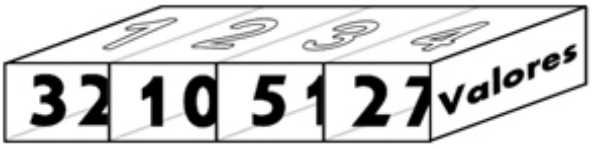
Vimos, até agora, que uma variável é um "local na memória do computador", que recebe um nome e pode armazenar um valor por vez.

Os vetores são um tipo de variável onde se podem armazenar vários valores ao mesmo tempo. A melhor forma inicial de entender como isto funciona é através de uma representação gráfica.

2 REPRESENTAÇÃO VISUAL DE UM VETOR

A seguir, veremos um exemplo com uma representação visual de como seria estruturado um vetor. Suponhamos que o nome do vetor seja **Valores**.

FIGURA 4 – REPRESENTAÇÃO VISUAL DE UM VETOR



FONTE: O autor

Conforme representado na imagem anterior, teremos uma única "caixa", de nome **Valores** e com vários "compartimentos" numerados (no exemplo, temos os compartimentos 1, 2, 3 e 4). Em cada um destes compartimentos, conseguimos colocar um único valor, e todos eles devem ser de um único tipo especificado. Em outras palavras, se a variável vetor for do tipo **Inteiro** (tipo utilizado no exemplo acima), só poderemos colocar valores inteiros em qualquer um dos compartimentos. Se a variável for do tipo **Caractere**, só poderemos colocar valores alfanuméricos nos compartimentos, e assim por diante.

Em resumo: no exemplo anterior, temos uma única variável (Valores), com 4 compartimentos numerados, cada um dos compartimentos está armazenando um valor do tipo **Inteiro**.

A numeração de cada compartimento pode ser chamada de "índice" ou "posição". Por exemplo, podemos dizer que o número 10 está no índice 2 ou o número 27 está na posição 4.

Uma forma comum de falar quando nos referimos a vetores: "Valores de 1", por exemplo, refere-se "ao conteúdo que a variável Valores tem no índice 1". No exemplo anterior, Valores de 1 contêm 32, Valores de 3 contêm 51, e assim por diante.

3 DECLARAÇÃO DE UM VETOR

Como se declara uma variável do tipo vetor? A sintaxe é esta:

```
<variável> : Vetor [<número do primeiro compartimento>..<número do último compartimento>] de Inteiro;
```

Onde:

`<variável>` é o nome da variável que armazenará os dados. No exemplo da "caixa", a variável é **Valores .**

`Vetor`: é a palavra que indica que a variável será do tipo vetor.

Dentro de colchetes, temos dois valores numéricos inteiros separados por ".." (pronuncia-se "ponto-ponto", pois pronunciar "dois pontos" pode confundir com o caractere "**:**").

"`de Inteiro`" significa que o vetor é "de valores inteiros".

Vamos ver como fica a declaração da variável **Valores** utilizada no exemplo:

```
Valores : Vetor [1..4] de Inteiro;
```

Vamos interpretar novamente cada parte da declaração:

- estamos criando uma variável chamada **Valores**;
- do tipo **Vetor**;
- que terá 4 áreas para armazenar valores;
- as áreas são numeradas de 1 a 4;
- o vetor poderá apenas armazenar valores inteiros.

Antes de vermos como trabalhar com os vetores, vamos ver mais um exemplo, agora armazenando valores do tipo **Caractere**.

Vamos criar um vetor para armazenar o nome de 5 pessoas.

```
Pessoas : Vetor [1..5] de Caractere;
```

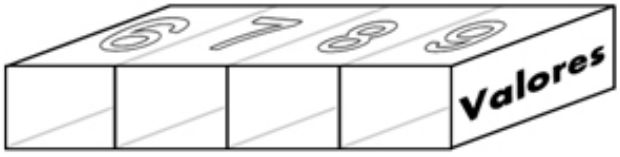
Representaremos de uma forma visual como ficaria esta variável:

FIGURA 5 – REPRESENTAÇÃO VISUAL DE UM VETOR DO TIPO CARACTERE

FONTE: O autor

Fique atento(a), caro(a) acadêmico(a), não é necessário iniciar os índices de um vetor a partir do 1. Se declararmos, por exemplo, `Valores : Vetor [6..9] de Real;` teremos apenas os índices 6, 7, 8 e 9, obtendo, assim, o seguinte vetor:

FIGURA 6 – REPRESENTAÇÃO VISUAL DE UM VETOR COM ÍNDICES INICIANDO EM 6



FONTE: O autor

No exemplo anterior, temos um vetor com índices iniciando do 6 até 9, sem valores armazenados no momento.

Não é comum iniciar um índice que não seja 1 ou até mesmo 0 (zero).

4 COMO TRABALHAR COM VETORES

Já vimos como declarar a existência de uma variável do tipo Vetor, indicando quantos valores ele poderá armazenar e de que tipo são estes valores. Veremos, agora, como utilizar um vetor.

Primeiramente, vamos utilizar como exemplo o vetor já exibido anteriormente:

```
Valores : Vetor [1..4] de Inteiro;
```

Vamos começar relembrando como armazenar um valor a uma variável que não seja um vetor. Basta atribuir o valor desejado à variável: `Variavel ← 32;`

Se tentássemos fazer a mesma coisa com o vetor Valores, ou seja, Valores ← 32, isto provocaria um erro. Para entender por que isto não é possível, preste muita atenção na dica a seguir, ela será muito importante para compreender como utilizar um vetor:

Sempre que utilizarmos um valor armazenado em um vetor, precisamos especificar em qual "casa" (índice) o valor está (ou será) armazenado.

Por exemplo, ao atribuir um valor ao vetor, precisamos "dizer ao algoritmo": "Olhe, o valor que eu vou armazenar no vetor tem que ser armazenado no índice 2". Ou ainda: "O valor que eu quero utilizar está armazenado no índice 1".

Em termos de sintaxe, como fazemos isto? Simplesmente colocando o índice entre colchetes após o nome da variável. Desta forma:

```
Valores[1] ← 32;
```

Pronto! Colocamos o valor 32 no índice 1 de Valores.

E para colocar o valor 10 no índice 2 da variável "Valores"? Agora ficou fácil:

```
Valores[2] ← 10;
```

E se desejarmos que o algoritmo solicite os valores a uma pessoa e armazene no vetor?

```
Escreva('Informe o 1° valor: ');
Leia(Valores[1]);
```

Neste caso, o **Leia** vai captar o que a pessoa digitar e vai colocar a informação na variável Valores, porém no índice 1.

Em muitas situações, é necessário que o usuário informe valores para preencher todo um vetor. Será que, nestes casos, teremos que fazer tudo isso?

```
Algoritmo ExemploVetor;
var
       Valores : Vetor [1..4] de Real;
Início
       Escreva('Informe o 1° valor: ');
       Leia(Valores[1]);
       Escreva('Informe o 2° valor: ');
       Leia(Valores[2]);
       Escreva('Informe o 3° valor: ');
       Leia(Valores[3]);
       Escreva('Informe o 4° valor: ');
       Leia(Valores[4]);
Fim.
```

O código acima até funciona, mas daria muita mão de obra, concorda?

Se tivéssemos que armazenar mais um valor? Teríamos que repetir mais uma vez o par de instruções (Escreva e Leia) que solicita e armazena o próximo valor.

Como poderíamos automatizar um pouco este processo?

Bom, se pararmos para analisar, estamos utilizando manualmente cada um dos índices possíveis do vetor.

Com uma estrutura de repetição, podemos fazer com que uma variável "passe" por todos estes índices possíveis, ou seja, primeiro a variável receberá 1, depois aumentará para 2 e assim por diante, até chegar ao último índice do vetor. A cada vez que esta variável está com um valor correspondente a um índice, podemos utilizar o vetor em conjunto com a variável, da seguinte forma:

Suponhamos que o nome da variável seja "**i**" e, neste momento, **i** contém 3. Se executarmos a instrução `Valores[i] ← 51;` estaremos colocando o valor 51 na "casa" 3 (valor da variável **i**) da variável **Vetor**.

Qual é a estrutura de repetição ideal a ser utilizada nesta situação? Bom, sabemos qual é o limite inferior (neste exemplo, 1) e o limite superior (4). Podemos concluir, com isto, que a melhor estrutura para esta situação seria o **Para-faça**.

Como ficaria:

```
1  Algoritmo Vetor_ParaFaca;
2  var
3         Valores : Vetor [1..4] de Real;
4         i : Inteiro;
5  Início
6
7         Para i ← 1 até 4 faça
8         Início
9               Escreva('Informe o ',i,'° valor: ');
10              Leia(Valores[i]);
11        Fim
12 Fim.
```

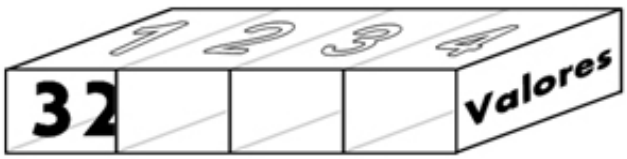
Na primeira vez que entrar no **Para-faça**, o valor de **i** é 1. Perceba a mensagem que aparecerá ao usuário:

Depois de "**Informe o**", será exibido o valor de **i**, ou seja, "**Informe o 1**", seguido de "**° valor:**". A frase completa será "**Informe o 1° valor:**".

Em seguida, o valor informado será armazenado na posição 1 de Valores. Por quê? Novamente: neste momento, **i** vale 1. Se estamos executando `Leia(Valores[i]);` consequentemente o valor vai ser armazenado na posição 1 de Valores.

Supondo que o usuário informou o valor 32, teremos, neste momento, a seguinte situação armazenada:

FIGURA 7 – REPRESENTAÇÃO VISUAL DE UM VETOR QUE RECEBEU UM VALOR



FONTE: O autor

Quando a execução chegar à última linha do **Para-faça**, ou seja, chegar em **Fim**, esta execução voltará para o início do **Para-faça** e aumentará o valor de **i**, ou seja, **i** passa a valer 2.

A frase exibida agora será: "**Informe o 2° valor:**"

Ao executar `Leia(Valores[i]);` o valor informado pelo usuário será armazenado na segunda casa do vetor Valores. Isto acontecerá até que o valor de **i** seja 4, onde será solicitado e armazenado um valor na última casa do Vetor e, após executar a quarta vez, o **Para-faça** terminará sua execução e o algoritmo seguirá executando as próximas instruções. Quando isto acontecer, o vetor estará preenchido com valores informados pelo usuário.

Uma boa sugestão, quando se trabalha com vetores, é armazenar a quantidade máxima de valores que se deseja armazenar no vetor, em uma constante. Com isto, sempre que desejássemos aumentar ou diminuir a capacidade do vetor, bastaria alterar o valor da constante.

Vamos ver o exemplo completo com a constante.

```
1  Algoritmo Exemplo_Vetor1;
2
3  const
4         Qtd ← 4;
5
6  var
7         Valores : Vetor [1..Qtd] de Real;
8         i : Inteiro;
9  Início
10        Para i ← 1 até Qtd faça
11 Início
12              Escreva('Informe o ',i,'° valor: ');
13              Leia(Valores[i]);
14        Fim
15 Fim.
```

Interpretando:

Na linha 4, declaramos uma constante com o valor 4.

Na linha 7, declaramos o vetor Valores. Veja que os índices de **Valores** vai de 1 até 4, que é o valor da constante **Qtd**. Neste vetor, armazenaremos valores do tipo real.

A estrutura **Para-faça** fará com que a variável **i** passe pelos valores 1 até 4. Como já é sabido, quando o **Para-faça** chegar a 4 (valor armazenado em **Qtd**), vai executar só mais esta vez, armazenando o dado que o usuário informar, na posição 4 do vetor.

Agora, vamos repetir a pergunta: Se tivéssemos que armazenar mais um valor?

Bastaria alterar o valor da constante **Qtd** para 5 ao invés de copiar mais uma vez as instruções que solicitam e leem um valor.

Agora (alterando o valor da constante), o vetor seria declarado com 5 índices, pois os índices vão de 1 até o valor de **Qtd**, que agora seria 5. O **Para-faça** também executaria 5 vezes, pois, da mesma forma, estamos fazendo com que a estrutura do **Para-faça** vá de 1 até o valor de **Qtd**.

Agora que temos o vetor preenchido com informações vindas do usuário, podemos "trabalhar" com esse vetor.

Vamos fazer um algoritmo que exiba a média dos valores armazenados? Como iniciaríamos isto?

Vamos pensar, primeiro, no cálculo da média. Como se calcula a média dos valores? Somam-se todos eles e divide-se pela quantidade total de valores armazenados.

A quantidade, nós já temos. Está na variável **Qtd**.

Como, então, somar todos os valores? Precisaremos "percorrer" o vetor, ou seja, passar por cada um dos seus valores armazenados e, a cada valor que passarmos, deveremos acumular a soma deste valor em uma variável, para que, ao terminar de percorrer, tenhamos acumulada a soma de todos eles na variável.

Utilizaremos muito o termo "percorrer um vetor". Tal termo significa executar uma estrutura de repetição que faça uma variável passar por cada um dos índices possíveis do vetor. Assim, todas as vezes que mencionarmos "percorrer o vetor", significa fazer um Para-faça que atribua gradativamente a uma variável tais índices, conforme exemplos já exibidos anteriormente.

Vamos criar uma variável chamada **Soma** que irá acumular a soma de cada um dos valores do vetor.

A cada valor que passarmos, vamos atribuir à **Soma** o valor que ela já tem acumulado até o momento, mais o valor do vetor.

```
 1  Algoritmo ExemploSoma;
 2  const
 3         Qtd = 4;
 4  var
 5         Soma : Inteiro;
 6
 7  Início
 8         Soma ← 0;
 9         Para i ← 1 até Qtd faça
10         Início
11                Soma ← Soma + Valores[i];
12         Fim
13  Fim.
```

Supondo que os valores armazenados no vetor foram estes:

```
Valores[1] ← 3
Valores[2] ← 90
Valores[3] ← 55
Valores[4] ← 76
```

Iniciamos soma com zero. Já veremos o porquê disso.

Já sabemos que o **i** irá variar de 1 até 4 (linha 9).

Quando o **i** for 1, Soma receberá o valor que ela tem armazenado + o número armazenado em Valores no índice 1 (isto acontecerá na linha 4).

Se não atribuirmos zero à **Soma**, na primeira execução, a variável **Soma** vai receber "qual valor" + 3?, se **Soma** ainda não tiver valor algum?

Atribuindo o valor zero antes do **Para-faça**, na primeira execução, a variável **Soma** receberá 0 + 3.

Na segunda execução, **Soma** receberá 3 + 90. O 3 é o valor que **Soma** tem no momento, mais o valor do vetor no índice 2, que é o índice que a variável **i** tem agora.

Na terceira execução, **Soma** receberá 93 + 55. Novamente: o 93 é o valor que **Soma** tem no momento, mais o valor do vetor no índice 3, valor de **i** neste momento.

Quando o **Para-faça** finalizar a execução, a variável **Soma** terá acumulado o valor de 224, soma de todos os valores do vetor. Agora, é só dividir esta soma pela quantidade, para obter a média.

Como fica o algoritmo completo.

```
1  Algoritmo Exemplo_Vetor2;
2
3  const
4         Qtd ← 4;
5
6  var
7         Valores : Vetor [1..Qtd] de Real;
8         i : Inteiro;
9         Soma : Real;
10        {Soma é do tipo real, pois armazenará a soma dos valores do
11 vetor, que são do tipo real}
12 Início
13
14        {Geralmente, faz-se a inicialização no começo do algoritmo}
15        Soma ← 0;
16
17        Para i ← 1 até Qtd faça
18        Início
19                    Escreva('Informe o ',i,'° valor: ');
20                    Leia(Valores[i]);
21        Fim
22
23        { Abaixo está o acúmulo de valores na variável soma}
24        Para i ← 1 até Qtd faça
25        Início
26 Soma ← Soma + Valores[i];
27        Fim
28
29        Escreva('A média dos valores é ', (Soma / Qtd));
30
31 Fim.
```

Note que na linha 29, onde se exibe a média, não é necessário armazenar a média só para exibir. Se o objetivo é este (só exibir), basta colocar o cálculo direto no comando **Escreva**.

E se quisermos exibir todos os valores armazenados em um vetor? A lógica é a mesma já estudada. Nós <u>não podemos</u> fazer isto:

```
Escreva(Valores);
```

Por quê? Como já visto: na instrução anterior, qual é o valor da variável **Valores** exibido? Não estamos especificando qual é o valor a ser exibido.

Repetindo: Sempre que utilizarmos um valor de um vetor, temos que <u>especificar onde (em que índice)</u> está o valor.

Também não basta executar somente algo assim:

```
Escreva(Valores[i]);
```

Se fizermos isto só uma vez, será exibido somente um valor, correspondente ao índice que **i** tem no momento.

Mas como faremos, então, para exibir todos?

A solução é fazer mais uma estrutura de repetição, que faça uma variável passar, novamente, por cada índice possível do vetor. A cada índice que a variável passar, exibiremos o conteúdo da variável **Valores**, naquele índice.

```
1  Para i ← 1 até Qtd faça
2  Início
3         Escreva(Valores[i]);
4  Fim
```

Quando **i** for 1, ao ser executado `Escreva(Valores[i]);` será exibido o conteúdo de valores no índice 1.

Quando **i** for 2, ao ser executado `Escreva(Valores[i]);` será exibido o conteúdo de valores no índice 1. Assim acontecerá sucessivamente até o fim do vetor.

Já vimos três formas básicas de trabalhar com um vetor:

1. preencher o vetor com valores;

2. utilizar cada valor do vetor para um determinado cálculo ou outra necessidade;

3. exibir todos os valores de um vetor.

O que há em comum em todos os itens anteriores? Para todos eles, precisamos **percorrer o vetor**.

Quando utilizarmos vetores, raramente não precisaremos percorrê-lo.

O termo "percorrer" pode ser encontrado como "varrer" em alguns livros. Exemplo: "Deve-se varrer o vetor em busca do maior valor".

RESUMO DO TÓPICO 4

Neste tópico, você viu que:

- Até antes deste tópico, apenas utilizamos variáveis onde era possível armazenar um valor de cada vez.

- A partir do conceito de vetores (e, consequentemente, matrizes), podemos armazenar vários valores ao mesmo tempo, em uma única variável.

- Nos vetores, cada valor está armazenado em um índice, e, para acessar tal valor, basta utilizar a variável seguida do índice entre colchetes. Exemplo: Valores[1]. Na maioria dos casos, utilizamos variáveis para representar os índices. Esta forma de utilização, junto com uma estrutura de repetição, faz com que se percorram todos os índices válidos de um vetor. Isto auxilia quando se deseja trabalhar com todos os valores de um vetor de uma vez.

AUTOATIVIDADE

1 O que são vetores?

2 Escreva um algoritmo que leia 10 valores e armazene-os. Em seguida, o algoritmo percorrerá todos os valores e, ao final, exibirá qual é o maior valor armazenado no vetor.

3 Escreva um algoritmo que leia 20 valores e armazene todos eles. Em seguida, o algoritmo deverá exibir quantos valores negativos estão armazenados no vetor.

4 Desenvolva um algoritmo que solicite 10 valores a armazene-os. Após armazenados, o algoritmo deverá exibir quantos valores pares estão armazenados no vetor.

5 Faça um algoritmo que solicite 15 valores e armazene todos em um vetor. Ao final, o algoritmo deverá exibir quantos valores múltiplos de três estão armazenados.

6 Refaça o algoritmo anterior, porém, após armazenados os valores, o algoritmo deverá solicitar mais um valor e, em seguida, exibir todos os valores múltiplos deste último valor lido.

7 Escreva um algoritmo que leia 20 valores e armazene todos eles. Em seguida, o algoritmo deverá exibir uma mensagem informando se a quantidade de valores pares é maior do que a quantidade de valores ímpares; se a quantidade de valores ímpares é maior do que a quantidade de valores pares ou se a quantidade é igual.

8 Desenvolva um algoritmo que solicite ao usuário quantos nomes ele deseja informar. Esta quantidade não pode ser maior do que 50. Em seguida, o algoritmo deverá solicitar um nome para busca.

Após solicitar o nome, o algoritmo deverá percorrer todos os nomes armazenados e verificar se o nome de busca existe armazenado. Se tiver, exibir a mensagem "Nome encontrado"; se não tiver, exibir a mensagem "Nome inexistente".

TÓPICO 5

MATRIZES

1 INTRODUÇÃO

Assim como os vetores, matrizes são um tipo de variável que pode armazenar, ao mesmo tempo, vários valores. Como o próprio nome sugere, este tipo de variável armazena valores em forma de matriz. Sim! Aquela matriz matemática que aprendemos no colégio. Vamos ver uma representação visual, com uma matriz 2 x 3 (ou seja, duas linhas por três colunas).

Matriz1

	1	2	3
1	32	12	29
2	40	81	18

No exemplo acima, temos uma matriz chamada **Matriz1** com alguns valores inteiros armazenados.

Outro exemplo:

MatrizNomes

	1	2
1	Humberto	Steven
2	Jéssica	Walter

Neste caso, há uma matriz chamada **MatrizNomes**, de dimensão 2 x 2 (duas linhas por duas colunas), que pode armazenar valores do tipo **Caractere**.

Nos vetores, como já vimos, para encontrar cada um dos valores armazenados na variável, precisamos indicar o índice em que o valor está armazenado. Com as matrizes, acontece a mesma coisa, porém, para chegar

a cada valor, precisa-se especificar um par de índices, ou seja, linha e coluna, separados por vírgula. A sintaxe fica assim:

```
Matriz1[2,1];
```

Neste exemplo, estaríamos obtendo o valor 40 (Matriz1 na linha 2 e coluna 1).

Mais um exemplo:

```
MatrizNomes[1,1];
```

Obteríamos, assim, o nome "Humberto".

2 DECLARAÇÃO DE MATRIZES

A declaração de uma variável do tipo matriz é muito semelhante à declaração de vetores. Basta informar que o tipo da variável é matriz ao invés de vetor e acrescentar a segunda coordenada. Vejamos como ficaria a declaração das duas variáveis exemplificadas anteriormente:

```
Matriz1 : Matriz [1..2,1..3] de Inteiro;
MatrizNomes : Matriz [1..2,1..2] de Caractere;
```

Como pudemos perceber, informamos, entre os colchetes, primeiro (antes da vírgula) o intervalo de linhas e, depois (após a vírgula), o intervalo de colunas.

Assim como nos vetores, é possível iniciar uma linha ou uma coluna em um índice que não seja 1 (um). Porém, na maioria das situações, não faz muito sentido. Podemos, então, acostumar-nos a iniciar sempre em 1.

3 PERCORRENDO UMA MATRIZ

Como faremos para percorrer uma variável do tipo matriz?

A ideia é começar pela primeira linha, passando, em seguida, por todas as colunas dela. Depois, passar para a segunda linha e, novamente, percorrer todas as colunas desta segunda linha e assim por diante.

Utilizando como exemplo a variável **Matriz1**, de dimensão 2 x 3, se começarmos pela primeira linha e percorrermos todas as colunas, quais serão as coordenadas possíveis?

[1,1] – Primeira linha com a primeira coluna
[1,2] – Primeira linha com a segunda coluna
[1,3] – Primeira linha com a terceira coluna

Depois, passaremos para a segunda linha, para a qual temos as coordenadas possíveis:

[2,1] – Segunda linha com a primeira coluna
[2,2] – Segunda linha com a segunda coluna
[2,3] – Segunda linha com a terceira coluna

A ideia, então, é utilizar uma estrutura **Para-faça** que passe por todos os índices de linha válidos. Ou seja, se tivermos duas linhas, fazemos um **Para-faça** de 1 até 2. Se tivermos 4 linhas, faremos um **Para-faça** que vá de 1 até 4.

Ao passar por cada execução do **Para-faça** que percorre as linhas, faz-se um outro **Para-faça**, para percorrer todas as colunas da linha que estiver passando no momento.

Vamos visualizar um trecho de uma estrutura **Para-faça** que faça uma variável percorrer a linha 1 e a linha 2 de uma matriz:

```
Para L ← 1 até 2 faça
```

Ao entrar na estrutura pela primeira vez, a variável **L** receberá 1. Enquanto o **L** tiver 1, temos que fazer uma outra variável passar por todos os valores possíveis de colunas da matriz. Como mencionado anteriormente, faremos isto com mais um **Para-faça**.

Vamos ver como ficaria:

```
1  Para L ← 1 até 2 faça
2  Início
3         Para C ← 1 até 3 faça
4         Início
5                 <Instruções>
6         Fim
7  Fim
```

Interpretando:

Na primeira instrução do algoritmo, **L** recebe 1 e entra no **Para-faça**.

A primeira instrução encontrada é outro **Para-faça**. Este irá executar 3 vezes (1 até 3).

Ao executar esta segunda estrutura, a variável **C** receberá 1 e a execução entrará na estrutura.

Se utilizarmos como coordenada da matriz, a combinação das variáveis L e C, obteríamos o valor que está armazenado na matriz, em tal coordenada.

Por exemplo, neste momento **L** contém 1 e **C** contém 1. Se utilizarmos **Matriz1[L,C]**, obteremos o valor 32 que, no exemplo, é o valor armazenado na coordenada 1,1 da **Matriz1**.

Ao chegar ao fim do **Para-faça** interno, a execução retornará para o início deste **Para-faça**. Neste momento, a variável **C** será incrementada para 2.

Novamente, ao utilizar **Matriz1[L,C]**, obteremos o valor de **Matriz1** na linha 1 (pois o valor de **L** não se alterou) e coluna 2. No exemplo, obteríamos o valor 12.

Vamos ver um algoritmo completo que solicite valores ao usuário para preencher uma variável chamada **Mat1**. Em seguida, o algoritmo percorre a matriz exibindo todos os valores nela armazenados.

```
1    Algoritmo Primeira_Matriz;
2    const
3           QtdLin = 2;
4           QtdCol = 2;
5    var
6           Mat1 : Matriz [1..QtdLin, 1..QtdCol] de Real;
7           L, C : Inteiro;
8
9    Início
10          { Percorre a matriz solicitando valores e armazenando }
11          Para L ← 1 até QtdLin faça
12          Início
13                 Para C ← 1 até QtdCol faça
14                 Início
15                 Escreva('Informe o valor da casa ',L,',',C,': ');
16                        Leia(Mat1[L,C]);
17   Fim
18   Fim
19
20          { Percorre a matriz exibindo seus valores }
21          Para L ← 1 até QtdLin faça
22          Início
23                 Para C ← 1 até QtdCol faça
24                 Início
25                        Escreva(Mat1[L,C]);
26                 Fim
27          Fim
28   Fim.
```

4 TESTE DE MESA

Vamos aproveitar o momento agora para aprender uma técnica de validação de algoritmos, chamada teste de mesa.

O teste de mesa consiste em simular valores e executar sequencialmente todas as instruções de um algoritmo a fim de verificar se ele está funcionando da forma desejada.

As primeiras instruções de um algoritmo, geralmente, são as declarações de constantes e variáveis. Tudo o que for declarado deve ser escrito (num papel ou num editor de textos, por exemplo), simulando os dados que efetivamente serão armazenados em um algoritmo.

No teste de mesa, armazenamos os dados em um papel ou editor; em um programa de computador, na prática, estes dados são armazenados na memória do computador. Conclui-se, então, que esta fase do teste de mesa (armazenamento de informações) simula o que está sendo armazenado na memória.

A cada instrução do algoritmo, deve-se alterar o estado (valores) das variáveis. Quando encontrar algum comando **Leia**, você deverá escolher um valor aleatório, simulando que o usuário está informando este valor. Ao final da simulação, basta analisar o valor das variáveis verificando se as instruções atenderam ao problema.

Para melhor entender como funciona, vamos fazer um teste de mesa do algoritmo `Primeira_Matriz`.

Ao executar as linhas 3, 4, 6 e 7, teremos este cenário:

```
QtdLin = 2
QtdCol = 2

Mat1

         1       2
   1   |  __      __  |
   2   |  __      __  |

L =
C =
```

Executando a linha 11, o cenário será este:

```
QtdLin = 2
QtdCol = 2


Mat1
        1     2
  1   |  __    __  |
  2   |  __    __  |


L = 1
C =
```

Ou seja, apenas o valor de L alterou para 1.

Executando a linha 13:

```
QtdLin = 2
QtdCol = 2

Mat1

        1     2
  1   |  __    __  |
  2   |  __    __  |

L = 1
C = 1
```

O valor de C passa a ser 1.

Analisando a execução da linha 15, o Escreva exibirá:

a) a mensagem `'Informe o valor da casa '`;

b) o valor da variável L (que agora é 1);

c) uma vírgula `','`;

d) o valor da variável C (que agora é 1);

e) dois pontos `': '`.

O resultado será:

```
Informe o valor da casa 1,1:
```

Veja que a execução da linha 15 não altera em nada o valor das variáveis.

Na linha 16, entra uma ação nossa, que estamos praticando o teste de mesa. Precisamos escolher um valor qualquer, simulando que o usuário o tenha informado. Vamos simular o valor 35. O **Leia** armazena o 35 na variável **Mat1**, na coordenada correspondente ao valor atual de **L** e **C**. Começaremos agora a utilizar o próprio estado atual do teste de mesa, ou seja, precisamos consultar quais os valores que **L** e **C** contêm no momento. Olhamos no pedaço de papel onde estamos anotando os estados (que no nosso caso é o quadro logo acima) e constatamos que o valor de **L** é 1 e o valor de **C** também é 1. Então, na simulação da linha 16, utilizamos estes valores. Resumindo, a matriz **Mat1**, na casa [1,1], receberá o valor que simulamos (35). Neste momento, nosso estado se altera novamente, ficando assim:

```
QtdLin = 2
QtdCol = 2

Mat1
        1     2
  1   | 35    __ |
  2   | __    __ |

L = 1

C = 1
```

Próxima instrução, linha 17. É o **Fim** do **Para-faça**. Neste momento, a execução volta para o início deste **Para-faça**, incrementando a variável **C** para 2.

```
QtdLin = 2
QtdCol = 2

Mat1

        1     2
  1   | 35    __ |
  2   | __    __ |

L = 1

C = 2
```

O **Para-faça** verifica se o valor de **C** não ultrapassou o intervalo final, ou seja, **QtdCol**. Como não ultrapassou, a execução irá entrar no **Para-faça**.

Agora, a mensagem exibida será:

```
Informe o valor da casa 1,2:
```

O **Leia** irá armazenar o valor informado em **Mat**, na coordenada [1,2]. Vamos simular mais um valor. Pode ser 14. Após executado o **Leia**, a situação será a seguinte:

```
QtdLin = 2
QtdCol = 2

Mat1
        1    2
  1   | 35   14 |
  2   | __   __ |

L = 1

C = 2
```

Mais uma vez, a execução chega ao **Fim** do **Para-faça**, retornando para o início e incrementando o valor de **C**, ficando agora, 3. O **Para-faça** identifica que ultrapassou o intervalo final e termina sua execução, saltando para a primeira instrução após o seu **Fim**.

Esta instrução é a linha 18, o **Fim** do primeiro **Para-faça**. Como é o **Fim** de um **Para-faça**, a execução retornará para o início, incrementando, agora, o valor da variável **L**.

```
QtdLin = 2
QtdCol = 2

Mat1
        1      2
  1   | 35     14  |
  2   | __     __  |

L = 2

C = 3
```

Como **L** não ultrapassou o intervalo final do **Para-faça** (neste caso, **QtdLin**), entra na execução.

A primeira instrução encontrada é, novamente, o **Para-faça** interno. A execução deste segundo **Para-faça** acontecerá da mesma forma que na primeira vez que executou: **C** passa a receber 1 novamente.

```
QtdLin = 2
QtdCol = 2

Mat1
        1      2
  1   | 35     14  |
  2   | __     __  |

L = 2

C = 1
```

O comando Escreva exibirá:

```
Informe o valor da casa 2,1:
```

O comando **Leia** colocará o valor informado pelo usuário, na coordenada **[L**,**C]**, ou seja, [2,1] da variável **Mat1**. Suponhamos o valor 79.

```
QtdLin = 2
QtdCol = 2

Mat1
        1      2
  1   | 35     14  |
  2   | 79     __  |

L = 2

C = 1
```

Próxima execução, linha 17, **Fim** do **Para-faça**, retorna ao início incrementando **C** para 2.

```
QtdLin = 2
QtdCol = 2

Mat1

        1      2
  1   | 35     14  |
  2   | 79     __  |

L = 2

C = 2
```

Mensagem exibida:

```
Informe o valor da casa 2,2:
```

O comando **Leia** armazenará o próximo valor informado, em **Mat1[2,2]**. Por final, simularemos o valor 43.

```
QtdLin = 2
QtdCol = 2

Mat1

        1     2
  1   | 35    14  |
  2   | 79    43  |

L = 2

C = 2
```

Novamente estamos na linha 17, que retorna ao início do **Para-faça**, incrementando **C** para 3, ultrapassando o limite final. A execução, então, passa para a linha 18, encontrando o **Fim** do **Para-faça** mais externo, retornando para o início deste. A variável **L** será incrementada, ultrapassando o limite final, passando a execução para a linha 19. O estado da memória (ou, no nosso caso, do teste de mesa) agora é:

```
QtdLin = 2
QtdCol = 2

Mat1
        1     2
  1   | 35    14  |
  2   | 79    43  |

L = 3

C = 3
```

Porém, os valores de **L** e **C** não nos interessam mais por enquanto. Como mencionado, eles apenas foram utilizados para alcançar cada casa possível da matriz. O que mais nos importa, agora, é que temos uma matriz preenchida com valores informados pelo usuário.

A parte mais importante do teste de mesa é esta: constatar que nosso algoritmo realmente conseguiu solicitar valores e preencher toda a matriz com estes.

Mas ainda temos mais um trecho do algoritmo.

AUTOATIVIDADE

Faça o teste de mesa do trecho restante do algoritmo.

RESUMO DO TÓPICO 5

Neste tópico, você viu que:

- As matrizes, assim como vetores, são variáveis nas quais é possível armazenar mais de um valor ao mesmo tempo. A diferença principal é que, devido à sua estrutura (igual a uma matriz matemática – daí seu nome), para acessar qualquer valor, utilizamos um par de índices, formando uma espécie de coordenada x,y (linha e coluna).

- Existe uma técnica para testar algoritmos, chamada "teste de mesa". Tal teste consiste em colocar em um papel, por exemplo, todas as variáveis, constantes etc., disponíveis no algoritmo e, em seguida, seguir todos os passos do algoritmo, simulando valores (quando o algoritmo solicita informações externas) e alterando o estado das variáveis.

- O teste de mesa auxilia, e muito, quando desejamos encontrar algum possível erro de lógica.

AUTOATIVIDADE

1 Faça um algoritmo que solicite valores e preencha uma matriz 4 x 4. Após armazenados os valores, o algoritmo deverá exibir qual é o maior valor armazenado na matriz e em qual linha x coluna ele está armazenado.

2 Crie um algoritmo que solicite valores e preencha uma matriz de ordem 5 x 4 (5 linhas por 4 colunas). Em seguida, o algoritmo deverá solicitar mais um valor para procurá-lo na matriz. O algoritmo deverá informar, no fim, em que linha x coluna está o valor. Se o valor não for encontrado, o algoritmo deverá exibir uma mensagem informando esta situação.

3 Desenvolva um algoritmo que leia valores e preencha uma matriz 4 x 4. O algoritmo deverá informar quantos valores maiores do que 5 estão armazenados na matriz.

4 Faça um algoritmo que solicite valores para preencher uma matriz 3 x 4. Depois, o algoritmo deverá solicitar mais um valor e criar uma segunda matriz, cujos valores são formados pelo resultado de cada um dos valores da primeira matriz, multiplicados pelo valor lido por último. Exibir a segunda matriz.

5 Crie um algoritmo que preencha uma matriz de dimensão 3 x 3 com valores fornecidos pelo usuário. O algoritmo deverá exibir a média dos valores de cada uma das linhas.

TÓPICO 6

SUBALGORITMOS

1 INTRODUÇÃO

Os subalgoritmos são também conhecidos por vários outros termos como: subprograma, sub-rotinas, módulos.

Um subalgoritmo é um trecho de algoritmo desenvolvido para realizar uma ação específica.

Dependendo da complexidade de um algoritmo, este pode ficar muito extenso. Neste caso, podemos dividi-lo em alguns subalgoritmos, cada um, realizando alguma ação (cálculos etc.) específica e bem definida.

Nós utilizamos subalgoritmos prontos praticamente desde o início deste livro. Um claro exemplo é o comando **Escreva**. Sim, ele é um subalgoritmo, com uma ação específica: recebe um conteúdo e envia este conteúdo para a saída de vídeo, ou seja, para o monitor do computador.

Não deve ser fácil fazer isto, mas algum programador desenvolveu este subalgoritmo para facilitar a nossa vida. Imagine que, para enviar um conteúdo para a saída de vídeo, fossem necessárias umas 9 instruções.

Agora imagine, também, que não existisse o **Escreva** e que, todas as vezes que fôssemos exibir alguma informação no monitor, precisássemos escrever estas 9 instruções. Seria muito trabalhoso, não? Mas como o **Escreva** existe, não precisamos perder tempo escrevendo todas estas instruções. Basta enviar para o subalgoritmo **Escreva** um conteúdo que desejemos exibir, e ele executará todas as instruções necessárias para que o conteúdo seja exibido na tela do computador.

Vejamos algumas **vantagens** em utilizar subalgoritmos, alguns exemplos:

- melhora a legibilidade do algoritmo principal, pois muitas instruções são transferidas para os subalgoritmos;

- modularização/reaproveitamento de código – construído, um subalgoritmo para realizar uma determinada ação, sempre que precisarmos executar esta ação, basta chamar o subalgoritmo, não sendo necessário executar todas as instruções novamente. Um subalgoritmo pode ser compartilhado para outros programadores, fazendo com que não se precise reescrever as instruções para resolver uma determinada ação que já foi resolvida ao construir tal subalgoritmo.

2 TIPOS DE SUBALGORITMO

Existem dois tipos distintos de subalgoritmos: os do tipo **Função** e os do tipo **Procedimento**.

A diferença entre os dois é muito simples: os subalgoritmos do tipo **Função** retornam um valor para o local onde foram chamados, enquanto os do tipo **Procedimento** não retornam valor algum. Ao estudarmos os primeiros exemplos, entenderemos melhor como funciona cada um destes tipos.

A sintaxe de um subalgoritmo é muito semelhante à de um algoritmo em si. Vejamos, primeiro, o tipo Procedimento:

```
Procedimento <nome_do_subalgoritmo> (<valores a receber>);
<área de declarações>
Início
       <instruções>
Fim;
```

Já vamos entender para que serve e como funciona cada uma das partes de um subalgoritmo. Porém, antes, vamos ver como é a construção de uma Função:

```
Função <nome_do_subalgoritmo> (<valores a receber>) : <tipo de retorno>;
<área de declarações>
Início
       <instruções>
Fim;
```

Vamos analisar as semelhanças e diferenças entre a estrutura de um subalgoritmo e de um algoritmo:

- No algoritmo, a palavra que antecede o nome do mesmo é “Algoritmo”, enquanto que nos subprogramas podem ser as palavras Procedimento ou Função, dependendo do tipo.

- Depois do nome do algoritmo há um ponto e vírgula. Após o nome de um subalgoritmo pode haver ou não um par de parênteses. Se tiver, haverá pelo menos uma variável declarada. Veremos logo a seguir a finalidade desta(s) variável(is).

- Se o subalgoritmo for do tipo Função, ainda se deve especificar o tipo de retorno. Também veremos este conceito a seguir.

- O corpo do subalgoritmo é praticamente igual: área de declaração de tipos (constantes, variáveis etc.), Início e Fim do subalgoritmo.

Tenha sempre isso em mente: tudo o que pode ser feito em um algoritmo, pode ser feito dentro de um subalgoritmo, ou seja, todos os comandos, instruções e estruturas podem ser utilizados.

Antes de entendermos como funcionam os conceitos que ficam localizados após o nome dos subalgoritmos, vamos ver o exemplo de um algoritmo que lê dois valores e informa qual é o maior deles (se forem iguais, informa qualquer um dos dois valores). Este algoritmo é só um exemplo que nos ajudará a entender os conceitos que virão logo a seguir.

```
1  Algoritmo Sub_Maior_Valor;
2
3  var
4         Valor1, Valor2, Maior : Real;
5
6  Início
7         Escreva('Informe o primeiro valor: ');
8         Leia(Valor1);
9
10        Escreva('Informe o segundo valor: ');
11        Leia(Valor2);
12
13        { Se Valor1 for maior}
14        Se (Valor1 > Valor2) então
15        Início
16 Maior ← Valor1;
17        Fim
18        { Se Valor2 for maior}
19        Senão Se (Valor1 < Valor2) então
20        Início
21                Maior ← Valor2;
22        Fim
23   {Se os dois forem iguais, atribui qualquer variável à "Maior"}
24        Senão
25        Início
26                Maior ← Valor1
27        Fim
28
29        Escreva('O maior valor é ',Maior);
30
31 Fim.
```

Nosso objetivo é passar a lógica principal, ou seja, o trecho do algoritmo que verifica se o valor é maior, para um subalgoritmo que terá sempre essa ação.

Primeiro, os parênteses com os <valores a receber>. Vamos procurar entender através de um exemplo:

Ao construirmos um subalgoritmo com a finalidade de verificar qual é o maior entre dois números, **o que o subalgoritmo precisa para realizar esta ação?** Ele precisa, justamente, de dois valores, certo?

Temos, então, que preparar o subalgoritmo para receber estes valores.

Como se faz esta preparação? Basta declarar variáveis entre os parênteses. Podem-se declarar quantas variáveis forem necessárias, e do tipo de valor que for necessário, tudo depende do que o subalgoritmo precisa para realizar sua ação.

Nosso subalgoritmo precisa receber dois valores, sendo estes do tipo Real. Vamos declarar duas variáveis, dentro dos parênteses:

```
(v1, v2 : Real)
```

O segundo passo, se o algoritmo for uma função, será definir qual é o tipo de valor que o subalgoritmo irá retornar. No nosso exemplo: se o subalgoritmo recebe dois valores, do tipo **Real** e retorna o maior deles, o tipo de retorno, neste caso, só poderá ser **Real** também, pois irá retornar um dos dois valores. Com isto, já concluímos, também, que o subalgoritmo é uma **Função**, pois retorna valor. Definido isto, temos mais uma parte do subalgoritmo:

```
Função <nome_do_subalgoritmo> (v1, v2 : Real) : Real;
```

Pronto! Agora só falta o nome do subalgoritmo. Vejamos, se a ação que o subalgoritmo irá realizar é verificar qual é o maior valor, nada mais indicado do que chamá-lo de **MaiorValor**.

Aqui está a declaração do subalgoritmo:

```
Função MaiorValor (v1, v2 : Real) : Real;
```

Agora, precisamos desenvolver toda a lógica (toda a ação) que a função irá executar. Como já vimos no algoritmo de exemplo, a lógica está feita. Basta retirá-la do algoritmo, adicioná-la na função e fazer pequenas adaptações.

```
1   Função MaiorValor (v1, v2 : Real) : Real;
2
3   var
4          Maior : Real;
5
6   Início
7          { Se v1 for maior}
8          Se (v1 > v2) então
9          Início
10                Maior ← v1;
11         Fim
12         { Se v2 for maior}
13         Senão Se (v1 < v2) então
14         Início
15                Maior ← v2;
16         Fim
17         { Se os dois forem iguais, atribui qualquer variável à "Maior" }
18         Senão
19         Início
20                Maior ← v1
21         Fim
22         MaiorValor ← Maior;
23  Fim
```

Vamos analisar o que precisou ser adaptado.

Veja que a comparação para verificar o maior valor é feita entre as variáveis **v1** e **v2**. Isto se deve ao fato de que, sempre que o subalgoritmo for chamado, deverão ser passados dois valores. As variáveis que receberão estes valores serão sempre **v1** e **v2**.

Assim, o maior valor entre **v1** e **v2** será armazenado na variável **Maior**. Se os valores forem iguais. o valor da variável **v1** (poderia ser **v2**, já que são iguais) será armazenado na variável **Maior**.

Uma das partes mais importantes a entender agora é a linha 22. Veja que estamos atribuindo o conteúdo da variável **Maior** ao nome da função (**MaiorValor**).

Lembra que uma característica essencial de um subalgoritmo do tipo Função é retornar um valor? É desta forma que se indica o retorno de um valor, ou seja, atribuindo o valor ao nome da função.

Veja que há disponível, dentro da função, mais de um valor (v1, v2 e Maior). Poderíamos retornar qualquer um deles. Mas o objetivo é retornar o maior, que está na variável **Maior**. Assim, ao finalizar a ação para a qual a função foi destinada, retornaremos o conteúdo da variável **Maior**.

A função está construída. Vamos ver agora como utilizá-la no algoritmo principal. Adaptaremos o algoritmo que já está pronto. Vamos retirar as linhas 13 a 27 (pois estas instruções foram passadas para a função). No local onde estavam estas linhas, vamos fazer a chamada à função.

Como se faz isto? Basta escrever o nome da função, passando os valores que ela precisa receber:

```
MaiorValor(Valor1, Valor2);
```

Veja que, ao chamar a função, estamos passando as variáveis que estão no algoritmo principal (que foi onde os valores foram lidos do usuário).

Este é um conceito fundamental quanto ao conceito de subalgoritmos: os identificadores (variáveis, constantes etc.) declarados dentro de um algoritmo não existem dentro de um subalgoritmo e vice-versa.

De acordo com essa afirmação, as variáveis **Valor1** e **Valor2** não existem dentro do subalgoritmo **MaiorValor**, bem como as variáveis **v1** e **v2** não existem dentro do algoritmo principal.

Certo, mas e a variável **Maior**, que está declarada tanto no algoritmo principal quanto no subalgoritmo? Neste caso, existem, durante a execução, duas variáveis chamadas **Maior**, sendo distintas uma da outra. Em outras palavras, são duas variáveis diferentes.

Os "valores a receber", nos subalgoritmos são comumente conhecidos como "parâmetros". Exemplo: "A função MaiorValor recebe dois parâmetros". Assim, a passagem de valores para um subalgoritmo também é chamada de "passagem de parâmetros".

Como visto no quadro anterior, foi feita a chamada ao subalgoritmo **MaiorValor**. Mas este subalgoritmo é uma função, ou seja, retorna valor. Onde estamos guardando o valor retornado? Até o momento, não estamos. Isto pode ser feito para que possamos exibir o resultado retornado para o algoritmo principal. Para armazenar este valor, retornado, basta fazer como se estivéssemos atribuindo o resultado de um cálculo a uma variável.

```
Maior ← MaiorValor(Valor1, Valor2);
```

Quando a execução chega a esta instrução, a variável **Maior** vai receber algo. Quando a execução chega a este "algo", encontra a chamada de um subalgoritmo. A execução então para no algoritmo principal e transfere-se para o subalgoritmo.

Nesta "transferência de execução", os valores que estamos enviando (neste caso, **Valor1** e **Valor2**) são transferidos, respectivamente, para as variáveis declaradas no subalgoritmo (**v1** e **v2**). Assim, se **Valor1** for 10 e **Valor2** for 20, **v1** recebe 10 e **v2** recebe 20, armazenando estes valores até o fim da execução do subalgoritmo.

Tenha sempre em mente este conceito: onde é feita a chamada de uma função, é como se ali estivesse o valor de retorno.

Isto quer dizer que, utilizando como exemplo os valores 10 e 20, a função retornaria 20 (o maior valor). Se, onde foi feita a chamada da função, é como se estivesse ali o valor de retorno, poderíamos dizer que a execução desta instrução:

```
Maior ← MaiorValor(Valor1, Valor2);
```

Seria a mesma coisa que esta:

```
Maior ← 20;
```

Ou seja, após executar **`Maior ← MaiorValor(Valor1, Valor2);`** o valor da variável **Maior** terá o valor 20 (obviamente, utilizando como exemplo os valores 10 e 20).

Agora, basta exibir na tela o maior entre os dois valores, que está armazenado na variável **Maior**.

Vamos ver como ficaria o algoritmo:

```
1   Algoritmo Sub_Maior_Valor;
2
3   var
4          Valor1, Valor2, Maior : Real;
5
6   Início
7          Escreva('Informe o primeiro valor: ');
8          Leia(Valor1);
9
10         Escreva('Informe o segundo valor: ');
11         Leia(Valor2);
12
13         { Chamada da função }
14         Maior ← MaiorValor(Valor1, Valor2);
15
16         Escreva('O maior valor é ',Maior);
17
18  Fim.
```

Mas em que local fica declarada a construção de um subalgoritmo?
Um algoritmo pode utilizar-se de quantos subalgoritmos (tanto funções, quanto procedimentos) forem necessários. Todos estes subalgoritmos serão construídos entre a área de variáveis e o "Início" (comando Início) do algoritmo principal.

Como ficaria, então, a junção completa do algoritmo principal e do subalgoritmo:

```
1  Algoritmo Sub_Maior_Valor;
2
3  var
4         Valor1, Valor2, Maior : Real;
5
```

```
6   { Início do subalgoritmo }
7   Função MaiorValor (v1, v2 : real) : real;
8
9   var
10         Maior : real;
11
12  Início
13         { Se v1 for maior}
14         Se (v1 > v2) então
15         Início
16                Maior ← v1;
17         Fim
18         { Se v2 for maior}
19         Senão Se (v1 < v2) então
20         Início
21                Maior ← v2;
22         Fim
23   {Se os dois forem iguais, atribui qualquer variável à "Maior"}
24         Senão
25         Início
26                Maior ← v1
27         Fim
28         MaiorValor ← Maior;
29  Fim;
30  { Fim do subalgoritmo }
31
32  Início
33         Escreva('Informe o primeiro valor: ');
34         Leia(Valor1);
35
36         Escreva('Informe o segundo valor: ');
37         Leia(Valor2);
38
39         { Chamada da função }
40         Maior ← MaiorValor(Valor1, Valor2);
41
42         Escreva('O maior valor é ',Maior);
43
44  Fim.
```

Como já visto, neste exemplo, armazenamos na variável **Maior** o resultado trazido pela função. Isto não é extremamente necessário. Se quisermos apenas exibir o resultado da função (sem precisar armazenar), basta chamar a função dentro do comando **Escreva**, da mesma forma que se faz quando se deseja exibir o valor de uma variável. Para isto, bastaria substituir as linhas 39 a 42 do algoritmo anterior, por esta linha:

```
Escreva('O maior valor é ',MaiorValor(Valor1, Valor2));
```

O **Escreva** irá exibir a mensagem e o valor retornado pela função.

Vamos ver um exemplo mais completo de um algoritmo, que utiliza uma função e um procedimento.

Problema: Desenvolver um algoritmo que leia o nome, o salário e o cargo de um funcionário de uma empresa. De acordo com o cargo, o funcionário receberá um aumento de salário (conforme quadro logo a seguir). Ao final, o algoritmo deverá exibir o nome do funcionário, o salário antigo e o novo salário com aumento.

Para auxiliar o algoritmo principal, faça:

a) uma função que receba o salário e o cargo do funcionário e calcule e retorne o novo salário (acrescentando o aumento conforme quadro a seguir);

Cargo	Aumento
1 – Analista de sistemas	5%
2 – Programador	8%
3 – Estagiário	7%
9 – Outros	2%

b) um procedimento que mostre as opções de cargo;

c) um procedimento que receba o nome, o salário antigo e o novo salário e exiba estas três informações.

```
1   Algoritmo Sub_Maior_Valor;
2
3   var
4          { Variáveis do algoritmo principal }
5          Nome, Cargo : caractere;
6          SalarioAtual,
7          NovoSalario : Real;
8
9   {***** Área de declaração das funções e procedimentos *****}
10
11  {Função que recebe o salário, o cargo e calcula o novo salário com aumento}
12  função CalculaNovoSalario (SalarioAtual : Real; Cargo : caractere) : Real;
13
14  var
15   {Armazenará o percentual de aumento de acordo com o cargo }
16         PercentualAumento : Real;
17
18  Início
19         Escolha (Cargo)
20                { Analista }
21                caso 1 : PercentualAumento ← 5;
```

```
22                { Programador }
23                caso 2 : PercentualAumento ← 8;
24                { Estagiário }
25                caso 1 : PercentualAumento ← 7;
26                { Outros }
27                caso 1 : PercentualAumento ← 2;
28        Fim;
29        { Calcula o novo salário e já faz a função devolvê-lo }
30  CalculaNovoSalario ← SalarioAtual + (SalarioAtual *PercentualAumento/100);
31
32  Fim; { Fim da função }
33
34  {Procedimento que exibe as opções de cargo }
35  procedimento ExibeCargos; { Não precisa receber nenhuma informação}
36  Início
37        Escreva('Escolha número referente ao cargo do funcionário: ')
38        Escreva('1 – Analista de sistemas');
39        Escreva('2 – Programador');
40        Escreva('3 – Estagiário');
41        Escreva('9 – Outros');
42  Fim; { Fim do procedimento }
43
44  {Procedimento que exibe os dados finais }
45  procedimento ExibeInformacoes(Nome : Caractere; Salario1, Salario2 : Real);
46  Início
47        Escreva('Nome do funcionário: ', Nome);
48        Escreva('Salário anterior: ', Salario1);
49        Escreva('Novo salário: ', Salario2);
50  Fim; { Fim do procedimento que exibe as informações finais }
51
52  { Fim da área de declaração das funções e procedimentos e início do
53  algoritmo principal }
54  Início
55
56        Escreva('Informe o nome do funcionário: ');
57        Leia(Nome);
58
59  Escreva('Salario atual: ');
60        Leia(SalarioAtual);
61  {Utiliza o procedimento para solicitar qual é o cargo do funcionário }
62        ExibeCargos;
63        Leia(Cargo);
64
65        { Executa a função que calcula o novo salário, enviado a ela
66  o salário atual e o cargo, e armazena o resultado retornado pela
67  função, na variável NovoSalario }
68        NovoSalario ← CalculaNovoSalario(Cargo, SalarioAtual);
69
70        { Chama o procedimento que exibirá as informações finais.
71  Para isto, estas informações são enviadas para o procedimento }
72        ExibeInformacoes(Nome, SalarioAtual, NovoSalario);
73
74  Fim.
75
76
77
```

A execução de um algoritmo sempre começa pelo algoritmo principal. As funções e procedimentos só serão executados quando forem chamados por ele (algoritmo principal), mesmo que a construção dos subalgoritmos seja no início.
Outro conceito interessante é que um subalgoritmo pode chamar outro. Tudo depende da necessidade.

Diante disto, a execução do algoritmo começa na linha 55.

O algoritmo solicitará e armazenará o nome e salário atual normalmente.

Quando chegar à linha 64, a execução passará para o procedimento (linha 36). O procedimento apenas exibirá isto na tela:

```
Escolha número referente ao cargo do funcionário:
1 - Analista de sistemas
2 - Programador
3 - Estagiário
9 - Outros
```

… e retornará à linha 65.

Com isto, o usuário já sabe o que ele precisa informar. Se o cargo for Programador, por exemplo, deverá informar 2, e assim por diante. A linha 65 armazenará a opção informada pelo usuário. A próxima execução será a linha 70. A execução colocará um valor na variável **NovoSalario**. Quando a execução for verificar que valor é esse, encontrará a chamada à função **CalculaNovoSalario**, enviando o salário atual e o cargo à função. A execução passará, então, para a linha 12. A função faz o cálculo do aumento de acordo com o cargo e retorna o valor calculado na linha 31 (como já vimos, para retornar um valor, basta atribuir este valor ao nome da função).

A execução volta para a linha 70, atribuindo o valor que a função retornou, para a variável **NovoSalario**. A próxima linha a ser executada é a 75, onde é encontrada a chamada a outro procedimento. Estamos enviando ao procedimento, nesta ordem exata, o nome do funcionário, o salário atual e o novo salário.

A execução vai para a linha 46. Quem recebe os dados são:

a) a variável **Nome** recebe o nome do funcionário (reforçando: enviado na linha 75);

b) a variável **Salario1** recebe o valor da variável **SalarioAtual**;

c) A variável **Salario2** recebe o valor da variável **NovoSalario**.

A ação do procedimento **ExibeInformacoes** será apenas exibir estas informações. O resultado exibido será algo semelhante ao conteúdo a seguir (simulando alguns valores):

```
Nome do funcionário: José da Silva
Salário anterior: 1000,00
Novo salário: 1050,00
```

LEITURA COMPLEMENTAR

Ordenação é o ato de se colocar os elementos de uma sequência de informações, ou dados, em uma relação de ordem predefinida. O termo técnico em inglês para ordenação é *sorting*, cuja tradução literal é "classificação".

Dada uma sequência de n dados:
$< a_1, a_2, \ldots, a_n >$
O problema de ordenação é uma permutação dessa sequência:
$< a'_1, a'_2, \ldots, a'_n >$
tal que:
$a'_1 \leq a'_2 \leq \ldots \leq a'_n$
para alguma relação de ordem.

Algumas ordens são facilmente definidas. Por exemplo, a ordem numérica ou a ordem alfabética – crescentes ou decrescentes. Contudo, existem ordens, especialmente de dados compostos, que podem ser não triviais de se estabelecer.

Um algoritmo que ordena um conjunto, geralmente representado num vetor, é chamado de algoritmo de ordenação. **Algoritmo de ordenação** em ciência da computação é um algoritmo que coloca os elementos de uma dada sequência em uma certa ordem – em outras palavras, efetua sua ordenação completa ou parcial. As ordens mais usadas são a numérica e a lexicográfica. Existem várias razões para se ordenar uma sequência. Uma delas é a possibilidade se acessar seus dados de modo mais eficiente.

Entre os mais importantes, podemos citar *bubble sort* (ou ordenação por flutuação), *heap sort* (ou ordenação por *heap*), *insertion sort* (ou ordenação por inserção), *merge sort* (ou ordenação por mistura) e o *quicksort*. Existem diversos outros, que o aluno pode com dedicação pesquisar por si. Para estudo, no entanto, nos concentraremos nos principais: *Selection Sort, Bubble Sort e Quicksort*.

Natureza dos Dados

Para melhor escolha de um método de ordenação, é preciso saber sobre a natureza dos dados que serão processados. Entre elas destacam-se duas: tempo de acesso a um elemento e a possibilidade de acesso direto a um elemento.

O tempo de acesso a um elemento é a complexidade necessária para acessar um elemento em uma estrutura. Ex.: Uma estante de livros com seus

títulos bem visíveis.

A possibilidade de acesso direto é a capacidade ou impossibilidade de acessar um elemento diretamente na estrutura. Ex.: Uma pilha de livros dentro de uma caixa, da qual precisamos tirar um a um para saber qual a sua natureza.

Para classificarmos estes dois ambientes de atuação, costumamos utilizar o meio em que estão armazenados os dados. Em termos computacionais utiliza-se a designação Ordenação Interna, quando queremos ordenar informações em memória. E Ordenação Externa, quando queremos ordenar informações em arquivo.

***Selection Sort* (Ordenação por Seleção)**

O ***Selection Sort*** utiliza um conceito de "selecionar o elemento mais apto". Ele seleciona o menor ou maior valor do vetor p.ex. e passa para a primeira (ou última posição dependendo da ordem requerida), depois o segundo menor para a segunda posição e assim sucessivamente com (n-1) elementos restantes até os dois últimos.

Teste de Mesa de *Selection Sort*

Vetor inicial:
4 3 1 2

Primeira passagem: Posição 0 - compara 4 com 3. Como 3 é menor que 4 este é fixado como mínimo, compara 3 com 1. Como este é menor do que 3 é fixado como mínimo. Compara 1 com 2. Como continua sendo menor, é fixado. Ao chegar ao final do vetor, como 1 é o menor elemento em comparação com o 4, eles trocam de posição.

1 3 4 2

Segunda passagem: Posição 1 - como já temos 1 como o menor elemento do vetor, passamos para a posição 1. Comparamos 3 com 4. Como é menor, 3 continua como mínimo. Compara com 2. Como 2 é menor este é fixado como mínimo. Ao chegar ao final do vetor, como 2 é o menor elemento em comparação com o 3, eles trocam de posição.

1 2 4 3

Terceira passagem: Posição 2 - pegamos o elemento da posição 2 (4) e comparamos com o 3. Como 3 é o último elemento do vetor e é menor do que 4, trocamos as posições. Como os dois elementos são os últimos do vetor, o *Selection Sort* encerra-se.

1 2 3 4

Algoritmo do *Selection Sort*

O algoritmo normalmente é implementado por duas repetições iterando sobre a estrutura em questão. Um exemplo de algoritmo é:

```
para i ← 0 até n-1
       minimo ← i
       para j ← i+1 até N
              se vetor[j] < vetor[minimo]
                     minimo ← j
                     auxiliar ← vetor[i]
                     vetor[i] ← vetor[minimo]
                     vetor[minimo] ← auxiliar
```

***Bubble Sort* (Ordenação Bolha)**

O ***bubble sort***, ou ordenação por flutuação (literalmente "por bolha"), é um algoritmo de ordenação dos mais simples. A ideia é comparar dois elementos e trocá-los de posição, até que os elementos de maior valor sejam levados para o final do vetor. O processo continua até a ordenação total do vetor lembrando a forma como as bolhas em um tanque de água procuram seu próprio nível, e disso vem o nome do algoritmo.

A complexidade desse algoritmo é de ordem quadrática ($O(n^2)$). Por isso, ele não é recomendado para programas que precisem de velocidade e operema com quantidade elevada de dados. Também é necessária uma **condição de parada**, geralmente uma ***flag*** ou variável que armazena se houve troca ou não na passagem. Se uma passagem chega ao seu final sem troca, a ordenação cessa.

Teste de Mesa

Vetor inicial:
4 2 5 1
Primeira Passagem: compara 4 com 2. 4 maior que 2. Mudam de posição.

2 4 5 1
Segunda Passagem: compara 4 com 5. 4 menor que 5. Permanece.

2 4 5 1
Terceira passagem: compara 5 com 1. 1 menor que 5. Mudam de posição.

2 4 1 5
Quarta passagem: compara 2 com 4. 2 menor que 4. Permanece.

2 4 1 5
Quinta passagem: compara 4 com 1. 1 menor que 4. Mudam de posição.

2 1 4 5
Sexta passagem: compara 4 com 5. 4 menor que 5. Permanece.

2 1 4 5
Sétima passagem: compara 2 com 1. 1 menor que 2. Trocam de posição.

1 2 4 5
Oitava passagem: compara 2 com 4. 2 menor do que 4. Permanece.

1 2 4 5
Nona passagem: compara 4 com 5. 4 menor do que 5. Permanece.

1 2 4 5
Décima passagem: não há mudanças. Sai do laço.

1 2 4 5

Algoritmo *Bubble Sort*

O algoritmo pode ser descrito em pseudocódigo como segue abaixo. V é um VETOR de elementos que podem ser comparados e n é o tamanho desse vetor.

BUBBLESORT (V[], n)

```
HouveTroca ← verdade { uma variável de controle }
enquanto HouveTroca = verdade faça
       HouveTroca ← falso
       para i de 1 até n-1 faça
              se V[i] vem depois de V[i + 1] então
                     troque V[i] e V[i + 1] de lugar e
                     HouveTroca ← verdade
```

Quicksort (Ordenação Rápida)

O *quicksort* (Ordenação Rápida) é um método de ordenação baseado no conceito de dividir e conquistar. Inicialmente ele seleciona um elemento o qual é chamado de pivô, depois remove este pivô e particiona os elementos restantes em duas sequências, uma com os valores menores do que o pivô e outras com valores maiores.

FONTE: ALGORITMOS e estruturas de dados/algoritmos de ordenação. Disponível em: <http://pt.wikibooks.org/wiki/Algoritmos_e_Estruturas_de_Dados/Algoritmos_de_>. Acesso em: 7 jan. 2013.

RESUMO DO TÓPICO 6

- Vimos neste tópico a existência e utilidade dos subalgoritmos, entre elas: uma melhor legibilidade do algoritmo principal, modularização e reaproveitamento de código.

- Existem dois tipos de subalgoritmos: funções e procedimentos.

- A diferença básica entre estes dois tipos é que as funções retornam um valor, enquanto os procedimentos não retornam.

- Quando se fala "retornar valor" significa retornar um valor para o ponto onde a função foi chamada.

- Tanto as funções quanto os procedimentos podem ou não receber valores para executar sua ação, tudo depende do que se necessita. Estes valores recebidos são chamados de parâmetros.

- A declaração (construção) dos subalgoritmos é feita no início, entre a área de variáveis e o início do algoritmo principal.

- Mesmo assim, a execução de qualquer subalgoritmo só acontece quando é feita uma chamada a ele pelo algoritmo principal.

- Variáveis e constantes que existem dentro de um algoritmo não são enxergadas dentro de um subalgoritmo e vice-versa.

- Os subalgoritmos nos ajudam a:

    - **modularizar ações** – passar blocos de instruções para subalgoritmo, dividindo as ações em módulos específicos;

    - **reutilização de código** – ações passadas para um subalgoritmo podem ser reutilizadas em outros algoritmos sem precisar ser reescritas;

    - **abstração** – depois de uma ação ser passada para um subalgoritmo, o programador não precisa mais se preocupar com tal ação. Basta passar a utilizá-la para resolver seu problema, sem precisar pensar em como o problema está sendo resolvido pelo subalgoritmo.

AUTOATIVIDADE

1 Explique o que é um subalgoritmo.

2 Quais são as utilidades de um subalgoritmo?

3 Quais são os tipos de subalgoritmos existentes?

4 Faça um algoritmo que leia o valor de uma mercadoria e um percentual de desconto a ser concedido. O algoritmo deverá passar estes valores a um subalgoritmo que calculará e retornará o valor a pagar. O algoritmo deverá exibir este valor.

5 Crie um algoritmo que receba um valor numérico. Este valor deverá ser passado para um subalgoritmo que retornará um valor lógico indicando se o valor é positivo ou negativo. Este subalgoritmo só deverá ser chamado se o valor for diferente de 0 (zero). Se for zero, o algoritmo deve exibir a mensagem "O valor informado é zero".

6 Escreva um algoritmo que leia um número e exiba uma mensagem informando se o número é primo ou não. O algoritmo utilizará um subalgoritmo que recebe um valor, verifica se ele é primo e retorna uma informação lógica indicando se é primo ou não. O algoritmo principal deverá utilizar esta informação para exibir a mensagem na tela.

7 Desenvolva um algoritmo que leia três números. Os três números serão enviados para um subalgoritmo que retornará o maior dos três. O algoritmo principal deverá exibir o maior valor.

8 Escreva um algoritmo para calcular exponenciação. O algoritmo deverá ler dois valores e exibir o resultado do primeiro elevado ao segundo (por exemplo, se os valores lidos forem 3 e 2, o algoritmo deverá exibir o resultado de 3^2). O algoritmo deverá utilizar um subalgoritmo que recebe dois valores, calcula e retorna o primeiro elevado ao segundo. Calcular somente com expoentes inteiros.

9 Desenvolva um algoritmo que leia um valor correspondente ao raio de uma esfera. O algoritmo deverá passar este valor para um subalgoritmo que calculará o volume da esfera e retornará este volume. O algoritmo principal deverá exibir o volume da esfera. A fórmula para calcular o volume de uma esfera, baseando-se no valor do raio, é $v = 4/3 * Pi * R^3$.

10 Faça um algoritmo que exiba este menu na tela:

1 – Adição
2 – Subtração
3 – Multiplicação
4 – Divisão

Este menu deverá ser exibido por um subalgoritmo.

O algoritmo principal deverá ler a opção do usuário e mais dois valores numéricos.

Os dois valores, mais a operação, deverão ser enviados a um subalgoritmo que fará o cálculo de acordo com a escolha do usuário e retornará o resultado.

Ex.: Se a escolha foi 2 (subtração) e os valores foram 2 e 8, o subalgoritmo retornará -6. O algoritmo deverá exibir o resultado.

UNIDADE 3

LINGUAGEM DE PROGRAMAÇÃO

OBJETIVOS DE APRENDIZAGEM

A partir desta unidade, você será capaz de:

- ter um contato com um ambiente e uma linguagem de programação;
- aprender técnicas de programação e estruturas principais de uma linguagem de programação;
- praticar e resolver problemas práticos em uma linguagem de programação.

PLANO DE ESTUDOS

Esta unidade está dividida em seis tópicos. Ao final de cada tópico, você encontrará atividades que auxiliarão a fixar as estruturas e praticar problemas práticos em uma linguagem de programação.

TÓPICO 1

INTRODUÇÃO À LINGUAGEM DE PROGRAMAÇÃO

1 INTRODUÇÃO

Até agora, não vimos nada funcionando na prática. Apenas estudamos exaustivamente vários conceitos existentes para desenvolver uma boa lógica computacional. Este estudo realizado, porém, é extremamente necessário para conseguirmos trabalhar com uma linguagem de programação sem sofrermos muito impacto.

A linguagem de programação que iremos estudar e na qual iremos construir nossos programas chama-se Pascal.

Esta linguagem foi escolhida por ser a mais parecida com a pseudolinguagem portugol. Mais uma vez, teremos pouco impacto ao migrar dos algoritmos para uma linguagem de programação, pois o Pascal é basicamente uma tradução dos comandos e estruturas do algoritmo para o inglês.

Os comandos de, praticamente, todas as linguagens de programação são palavras em inglês.

2 UM BREVE HISTÓRICO

A linguagem Pascal teve sua origem em Genebra, Suíça, no ano de 1972, pelas mãos do professor Niklaus Wirth.

Segundo Valdameri (2003), a linguagem Pascal foi concebida com base em outras linguagens estruturadas existentes, como, por exemplo, o ALGOL, e sua popularidade foi evidenciada quando a Universidade da Califórnia a adotou, em meados de 1973.

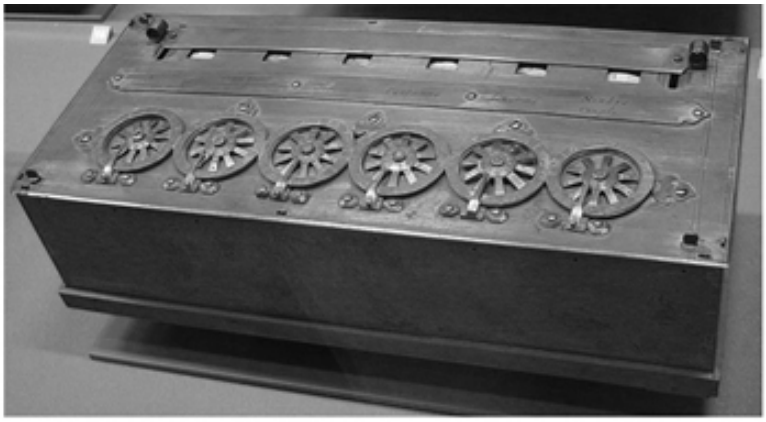
O nome desta linguagem foi uma homenagem ao filósofo e matemático Blaise Pascal.

FIGURA 8 – BLAISE PASCAL

FONTE: Disponível em: <http://www.brasilescola.com/uploa-d/e/pascal.jpg>. Acesso em: 19 jul. 2012.

Entre suas diversas contribuições para áreas como a geometria, física, para teoria das probabilidades etc., Blaise Pascal, para aliviar o trabalho de seu pai, que exercia a função de agente fiscal, inventou uma máquina de calcular. Esta foi a primeira máquina de calcular mecânica do mundo, planejada em 1642. A máquina ficou conhecida como pascalina.

FIGURA 9 – PASCALINA



FONTE: Wikipedia (2012)

Devido a esta invenção, Blaise Pascal é considerado o pai da Ciência da Computação.

Voltando à linguagem de programação Pascal, o objetivo de Niklaus Wirth era criar uma linguagem para o ensino da programação, uma linguagem

que fosse simples, incentivando o aprendizado através de programas claros e legíveis, permitindo a utilização de boas técnicas de programação.

Esta linguagem se tornou amplamente conhecida através do ambiente de programação Turbo Pascal, da Borland.

3 AMBIENTE DE PROGRAMAÇÃO

Como nosso objetivo é ter um primeiro contato com linguagem de programação, sem nos preocupar muito com instalações e configurações de ambientes de programação muito avançados, utilizaremos o ambiente **Pascalzim** (também conhecido como **Pzim**). Este ambiente foi desenvolvido no Departamento de Ciências da Computação da Universidade de Brasília.

Ele é, basicamente, um editor de texto simples, onde você escreverá seus programas. Este ambiente aplica uma coloração no código, fazendo com que, visualmente, seja fácil identificar os comandos e estruturas da linguagem.

Ao escrever um programa, este ambiente permite que executemos o programa e, então, consigamos colocar em prática tudo o que estudamos até agora.

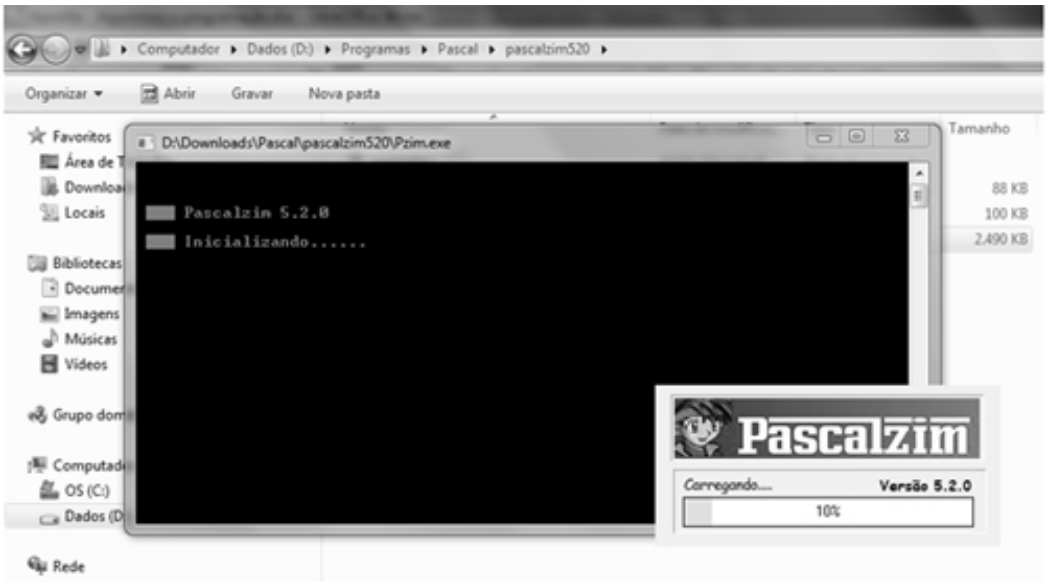
4 BAIXANDO E EXECUTANDO O AMBIENTE DE PROGRAMAÇÃO PASCALZIM

O Pascalzim pode ser baixado em <http://pascalzim.tripod.com → Download>. Descompacte o arquivo baixado em uma pasta, por exemplo, **C:\Pascalzim**.

Este ambiente não precisa ser instalado. Bastar dar um duplo clique sobre o arquivo **Pzim.exe** e o ambiente será executado.

A visão que você terá, ao executar o Pascalzim, é demonstrada pela figura:

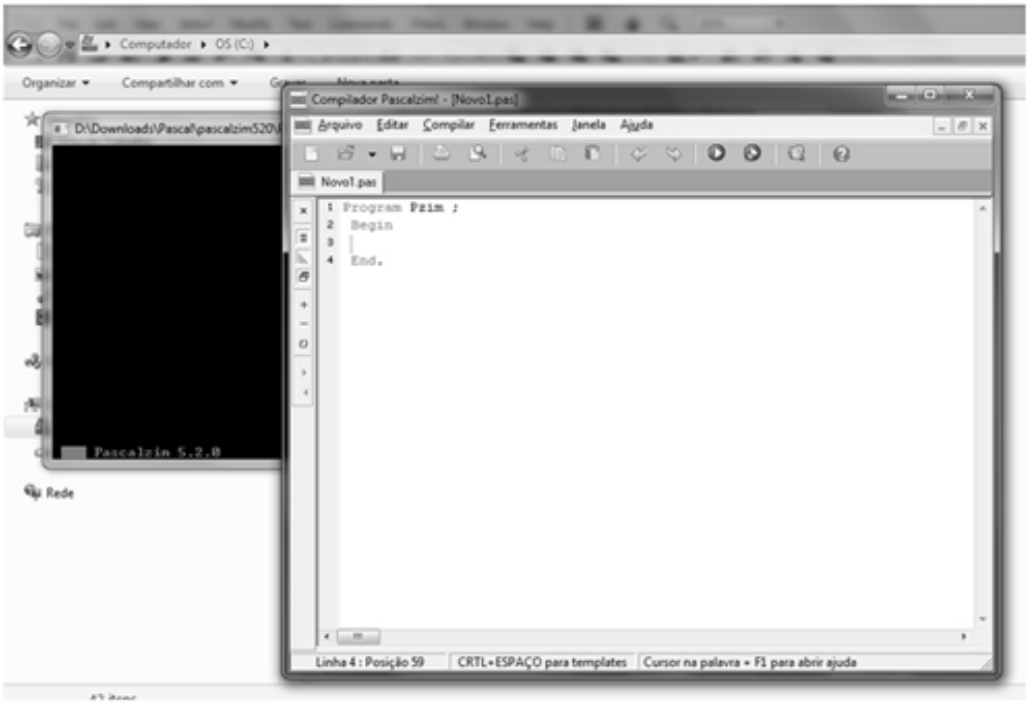
FIGURA 10 – ABRINDO O PASCALZIM



FONTE: O autor

Em seguida, será aberto o ambiente propriamente dito: o editor que comentamos há pouco. É diretamente neste editor que escreveremos nossos códigos-fonte, ou seja, nossos programas.

FIGURA 11 – EDITOR DE CÓDIGO-FONTE DO PASCALZIM



FONTE: O autor

Você pode perceber que o Pascalzim sempre manterá duas janelas abertas. Não trabalharemos inicialmente com a tela de fundo preto, onde está escrito "Pascalzim", em cor verde abaixo da tela.

Uma boa opção de visualização do editor é escolher a opção "Estilo moderno", no menu "Janela".

5 DICAS DO AMBIENTE PASCALZIM

O editor do Pascalzim é muito fácil de trabalhar. Cada programa ficará em um arquivo com extensão **.pas**. Para abrir um arquivo já feito ou salvar o seu arquivo (o seu programa), basta seguir, praticamente, os mesmos passos que você faria, por exemplo, no Bloco de Notas do Windows, bem como as ações de copiar, colar, recortar etc.

Vamos focar, nessa etapa, as ações diferentes do que já estamos acostumados, as quais envolvem, principalmente, executar os programas que estamos criando.

Você pode ouvir muito na área da informática a palavra "rodar" um programa. Rodar é a mesma coisa que executar.

Para rodar um programa, basta pressionar a tecla F9 ou clicar no botão  .

6 *CASE SENSITIVE*

Agora que começamos a estudar uma linguagem de programação, é importante conhecermos o conceito de *case sensitive* e *case insensitive*.

Toda linguagem tem um destes conceitos. Algumas até mesclam um pouco de cada conceito, mas a maioria segue um só.

Case sensitive significa **que a linguagem** faz distinção entre letras maiúsculas e minúsculas. Em outras palavras, se dizemos que uma linguagem é case sensitive, as letras maiúsculas são diferentes de minúsculas.

Exemplo: a variável **Nome** é diferente da variável **nome**, que é diferente da variável **NOME**, e assim por diante.

O mesmo acontece não só para as variáveis mas também para os comandos da própria linguagem.

O Pascal não é uma linguagem *case sensitive*. Isto quer dizer que não faz diferença utilizar variáveis e/ou comandos em maiúsculas e utilizar os mesmos em minúsculas.

A situação na qual maiúsculas e minúsculas farão diferença está nos valores armazenados em variáveis. Por exemplo, em uma variável chamada **Continua**, está armazenada a letra 'n' (minúscula). Se for feita a seguinte comparação:

```
Se (Continua = 'N') então
```

Mesmo que, por exemplo, o usuário tenha informado 'n', indicando que não deseja continuar, o resultado da condição do **Se-então** será falsa, indicando que a variável **Continua** não tem a letra 'n'. Isto porque o valor 'N' (maiúsculo), que é o valor que está sendo comparado com o valor armazenado na variável, é diferente deste valor armazenado.

Em resumo, o Pascal é *case insensitive* com nomes de identificadores, comandos e instruções, mas faz a distinção entre minúsculas e maiúsculas com valores armazenados em variáveis.

Praticamente todas as linguagens fazem distinção entre letras maiúsculas e minúsculas com valores armazenados de variáveis.

7 COMANDOS BÁSICOS

Como já mencionado, um programa em Pascal é basicamente uma tradução de um algoritmo para o inglês, salvo algumas diferenças de sintaxe que veremos mais tarde.

Vamos ver uma relação inicial de comandos básicos do algoritmo convertidos para a linguagem Pascal:

Em algoritmo	No Pascal
Algoritmo	Program
var (ou Variáveis)	var
const (ou Constantes)	const
Início	Begin
Fim	End
Escreva	Write
Leia	Read

8 EXECUTANDO O PRIMEIRO PROGRAMA

Até agora, estudamos muito algoritmo, muita lógica, mas não vimos nada realmente funcionar na prática. Talvez você esteja ansioso(a) por ver este funcionamento.

Antes, então, de estudarmos mais detalhadamente cada comando, vamos executar um primeiro programa, bem básico, só para ver como isto acontece. Em seguida, começaremos a estudar cada comando da linguagem de programação.

Escreva o código abaixo no editor do Pascalzim e execute-o (agora, já sabemos como executar, não é mesmo?).

```
Program PrimeiroPrograma ;
Begin
      Write('Oi! Este é meu primeiro programa.');
End.
```

Como já foi dito e como pudemos perceber através do código-fonte anterior, um programa é quase que uma tradução de um algoritmo. Perceba que os comandos em negrito são os comandos já utilizados nos algoritmos, apenas traduzidos.

Se tudo deu certo, aparecerá naquela tela preta, a qual mencionamos anteriormente, a mensagem:

```
Oi! Esse é meu primeiro programa.
```

Para finalizar o programa, basta pressionar a tecla **[Enter]**.

Agora, já sabemos para que serve aquela tela preta: toda a execução dos programas aparecerá nela.

9 TIPOS DE DADOS

O quadro anterior com a equivalência dos comandos básicos nos mostrou como ficaram alguns comandos do algoritmo no Pascal.

Veremos agora outro quadro que mostra, em Pascal, os tipos de dados que estudamos até agora em algoritmos.

Em algoritmo	No Pascal
Inteiro	Integer
Real	Real
Caractere	String
Lógico	Boolean

Quando nos referimos a um conjunto de caracteres, geralmente delimitados com apóstrofos (já vimos vários exemplos nos algoritmos), em uma linguagem de programação, costuma-se utilizar a palavra "string", pois esta palavra, como pudemos conferir no quadro anterior, representa o tipo de dado "caractere".

Vamos ver um exemplo do que o UNI quis dizer. Para isto, vamos utilizar a instrução a seguir:

```
Escreva('Olá! Vamos analisar esta mensagem.');
```

Estamos passando para o **Escreva** um conjunto de caracteres, ou seja, uma *string*.

Frases que poderemos ouvir:

– esta *string* contém 34 caracteres;

– o **Escreva** está exibindo uma única *string;*

– a *string* contém a palavra "analisar".

Repetindo: costuma-se, em uma linguagem de programação, referir-se a um conjunto de caracteres pela palavra *string*.

10 OUTRO TIPO DE DADOS

Em linguagens de programação, temos inúmeros tipos de dados, além dos tipos estudados nos algoritmos. Vamos ver mais um tipo de dados, que já utilizamos nos algoritmos, mas lá não fazíamos a distinção.

O tipo de dados **char** é como se fosse uma **string**, porém pode armazenar um único caractere.

Para que ele serve?

Lembra o exemplo no qual utilizávamos uma variável chamada **continua**? Esta variável armazenava a letra 's' ou a letra 'n', ou seja, um único caractere.

Poderia ser declarado do tipo **string**? Sim, pois as *strings* podem comportar um único caractere também.

Porém, no exemplo da variável **continua**, não é necessário declarar a variável continua como sendo do tipo **string**.

Mas se podemos utilizar o tipo **string**, por que não utilizar? Vamos tentar entender isto com um exemplo visual.

Digamos que o quadro a seguir represente a memória do computador:

Memória do computador

Cada variável declarada em um programa ocupará um espaço nesta memória. O tamanho deste espaço a ser ocupado depende do tipo de dado desta variável.

Este é um dos motivos pelos quais existem os tipos de dados, para que o programador possa ter um certo controle sobre o quanto o seu programa irá utilizar da memória do computador.

Ao declarar, por exemplo, a variável **continua**, como sendo do tipo **string**, estaríamos ocupando a memória desta forma:

Memória do computador
continua

Se declarássemos a variável **continua** como sendo do tipo **char**, a memória seria ocupada desta forma:

Memória do computador
continua

11 SINAL DE ATRIBUIÇÃO

Nos algoritmos, para atribuir um valor a uma variável, já sabemos que é utilizada uma seta para esquerda (←). No Pascal, basta substituir a seta pela junção dos dois-pontos com um sinal de igual (**:=**).

Vamos ver alguns exemplos:

```
1  Program Exemplo_Atribuicao ;
2
3  var
4         Nome : String;
5         Peso, PesoTotal : Real;
6         Quantidade : Integer;
7
8  Begin
9         Nome := 'Walter Moraes';
10        Peso := 78.5;
11        Quantidade := 32;
12        PesoTotal := Peso * Quantidade;
13
14 End.
```

Como você pode constatar, não há dificuldade nenhuma em atribuir valores em Pascal.

Note que, na linha 10, estamos atribuindo um valor fracionado à variável Peso. Perceba que o caractere que separa a parte decimal é um ponto, e não vírgula como estamos acostumados. Vamos relembrar: isto acontece porque, nas linguagens de programação, é utilizada a notação matemática que provém da cultura inglesa. Nós estamos acostumados a utilizar a notação provinda dos países da América não inglesa e da Europa, por isso a utilização da vírgula como separador de decimal pode ser encontrada nos algoritmos.

12 COMANDOS ESCRITA E LEITURA

12.1 WRITE

Já pudemos ter noção destes comandos através do quadro de equivalência dos comandos básicos.

O comando utilizado para exibir informações na tela é o **write**, simples tradução de **escreva**.

12.2 WRITELN

No Pascal, há, também, uma alternativa ao comando **write** que é o **writeln**. As letras "ln" provêm de *line* (linha, em português).

A diferença entre o **write** e o **Writeln** é que o **writeln** faz uma quebra de linha na tela após exibir a informação.

Vamos ver, na prática, a diferença entre estes dois comandos.

```
1  Program Exemplo1 ;
2
3  Begin
4         Write('Esta é a primeira linha');
5         Write('Esta é a segunda linha');
6  End.
```

No exemplo anterior, o resultado exibido na tela será este:

```
Esta é a primeira linhaEsta é a segunda linha
```

O que acontece é que o **write** da linha 4 do programa irá exibir a mensagem e manter a posição logo após a mensagem exibida. Desta forma, o próximo conteúdo a ser exibido ficará "colado" ao conteúdo anterior, como mostrado no quadro acima.

Quando utilizamos o **writeln**, após ser exibido o conteúdo, a posição passará para a linha de baixo, na tela.

Vamos alterar o programa anterior, simplesmente trocando o **write** por **writeln**.

```
1  Program Exemplo1 ;
2
3  Begin
4         WriteLN('Esta é a primeira linha');
5         WriteLN('Esta é a segunda linha');
6  End.
```

Após ser executada a linha 4, a posição do próximo conteúdo será a linha de baixo. Vamos representar esta posição através de um asterisco no quadro a seguir:

```
Esta é a primeira linha
*
```

Assim, quando for executada a linha 5, o conteúdo será exibido no ponto onde está o asterisco. Vamos ver como ficará a execução do programa anterior.

```
Esta é a primeira linha
Esta é a segunda linha
```

O **write** pode ser utilizado de todas as formas que já utilizamos o **escreva**. Vamos ver um código-fonte que mostra várias formas de utilização.

```
1  Program ExemplosWrite;
2  const
3        Valor1 = 10;
4  Begins
5        Writeln('Informações a serem exibidas:');
6        Writeln('Conteúdo da variável Valor1: ', Valor1);
7        Writeln('Exibindo o resultado de um cálculo: ', (Valor1 +
8  10));
   End.
```

No programa, temos uma constante chamada **Valor1** contendo o valor 10.

O resultado será este:

```
Informações a serem exibidas:
Conteúdo da variável Valor1: 10
Exibindo o resultado de um cálculo: 20
```

Vamos prestar atenção em algumas dicas bem interessantes:

1. note o uso do writeln. A cada mensagem exibida, será feita uma quebra de linha. Assim, cada mensagem aparecerá em uma linha diferente na tela;
2. no writeln da linha 6, no fim da mensagem, há um espaço entre os dois-pontos e o apóstrofo que finaliza a string. Este é um pequeno detalhe, mas é muito importante para a elegância na exibição de mensagens na tela. Veja o que nosso amigo UNI tem a dizer sobre isso:

É sempre essencial nos preocuparmos com a qualidade das informações que são exibidas na tela. As pessoas que usarão um programa que nós construímos são nossos clientes. É crucial sempre termos em mente, ao desenvolver um programa, que além de funcionar da melhor maneira possível, o programa agrade aos olhos dos nossos clientes. Resumindo, acentue as palavras, faça as pontuações corretas e, como citado no exemplo, faça espaçamentos que deixem uma boa qualidade visual das informações exibidas.

Na linha 7, é exibida uma mensagem – e como há conteúdo depois da mensagem, aquele pequeno espaço é importante – seguida de um cálculo. Veja que, para tornar a instrução bem organizada, o cálculo é colocado entre parênteses. Em alguns casos, é essencial colocar o cálculo entre parênteses para manter as regras matemáticas do cálculo. A dica, neste caso é: na dúvida, sempre isole os cálculos entre parênteses. Além de organizar o código-fonte, manterá as regras matemáticas.

Outra característica que pudemos perceber no programa é que a declaração de constantes continua igual à feita no algoritmo.

12.3 READ

O comando de leitura, na linguagem Pascal, é o **read**. Assim como o **write** é a tradução do **escreva**, o **read** é a tradução do **leia**.

12.4 READLN

O **readln** executa a funcionalidade igualmente ao **read**, porém adiciona a mesma característica que acontece com o **writeln**: o **readln** lê um valor, armazena-o em uma variável e, em seguida, coloca a posição da próxima informação a ser exibida, na linha seguinte na tela.

Vamos ver uma sequência de programas que esclarecerão melhor o uso, tanto do **write/writeln** quanto do **read/readln**.

```
 1  Program ExemploReadWrite1;
 2  var
 3         Nome : String;
 4         Idade : Integer;
 5  Begins
 6         Write('Informe seu nome: ');
 7         Readln(Nome);
 8         Write('Informe sua idade: ');
 9         Readln(Idade);
10
11         Writeln('Confira seus dados:');
12         Writeln('Seu nome é ', Nome);
13         Writeln('Sua idade é ', Idade);
14  End.
```

Vamos analisar bem minuciosamente o uso do "**ln**" nos comandos, além de outros detalhes importantes.

Na linha 6, utilizamos apenas **write** (e não **writeln**). Isto é interessante quando queremos que o dado que o usuário informar apareça logo após a mensagem que solicita a informação.

Simulando que o usuário informe "Edson Dutra", é isto que aparecerá na tela:

```
Informe seu nome: Edson Dutra
```

Dois detalhes a serem demonstrados:

a) Se utilizássemos o **writeln** ao invés do **write**, apareceria assim:

```
Informe seu nome:
Edson Dutra
```

Talvez seja isso que você possa querer fazer. Mas, geralmente, faz-se o dado informado aparecer ao lado da mensagem.

b) Já vimos esta situação, mas vamos rever com esse exemplo. Se não colocássemos aquele espaço no final da *string*, entre os dois-pontos e o apóstrofo:

```
Write('Informe seu nome:');
```

Quando o usuário informasse seu nome, apareceria assim na tela:

```
Informe seu nome:Edson Dutra
```

Veja que é um detalhe mínimo, mas, como já dito, fica mais elegante manter o espaço.

O **readln**, na linha 7, é o comando que ficará esperando o usuário digitar a informação até pressionar a tecla [enter]. O comando **readln** (ou **read** se for o caso), ao identificar que foi pressionada a tecla [enter], armazena todo o conteúdo que foi digitado desde o momento de sua execução, na variável que foi passada para ele (para o **readln**). No caso, a variável **Nome**.

Como está sendo utilizado o **readln**, após colocar o conteúdo digitado, na variável, uma linha será "quebrada" na tela, ou seja, o posicionamento irá pular uma linha.

Assim, quando a linha 8 for executada, a *string* será exibida na linha seguinte. Veja como ficará:

```
Informe seu nome: Edson Dutra
Informe sua idade:
```

O mesmo acontecerá com a linha 9. A informação captada será colocada na variável **Idade** e, na tela, será saltada mais uma linha.

Veja que, a partir da linha 11, todos os **write** são utilizados com o **ln**. Isto porque o objetivo é exibir cada informação em uma linha diferente.

Veja como ficará o resultado na tela, considerando que o nome da pessoa seja Edson Dutra e que a idade seja 52 anos.

```
Informe seu nome: Edson Dutra
Informe sua idade: 52
Confira seus dados:
Seu nome é Edson Dutra
Sua idade é 52
```

Alguns detalhes interessantes, ainda sobre a visualização de informações na tela:

Podemos perceber que as linhas nas quais são solicitadas as informações estão "juntas" com as linhas que exibem as informações. Não há nenhuma divisão visual. Se quiséssemos exibir uma pequena separação, desta forma:

```
Informe seu nome: Edson Dutra
Informe sua idade: 52

Confira seus dados:
Seu nome é Edson Dutra
Sua idade é 52
```

Veja que há uma linha em branco entre as linhas que solicitam dados e as linhas que exibem os dados.

Para fazer essa quebra de linha, basta executar um **writeln**, sem passar nenhuma informação a ele.

Vejamos:

```
1  Program ExemploLinhaEmBranco;
2  var
3         Nome : String;
4         Idade : Integer;
5  Begins
6         Write('Informe seu nome: ');
7         Readln(Nome);
8         Write('Informe sua idade: ');
9         Readln(Idade);
10        Writeln;
11        Writeln('Confira seus dados:');
12        Writeln('Seu nome é ', Nome);
13        Writeln('Sua idade é ', Idade);
14 End.
```

Como não foi passado nada para o **writeln** exibir, nada será visualizado na tela; e como o **writeln** quebra uma linha, esta quebra fará com que seja deixada uma linha em branco na tela.

13 LIMPAR A TELA

Muitas vezes, é necessário limpar as informações que estão na tela, tudo dependendo de como o programador deseja exibir as informações.

Para exemplificar de uma forma bem simples, vamos utilizar o programa anterior.

Digamos que, ao invés de simplesmente deixar uma linha em branco entre a solicitação de informações e a exibição destas informações, desejamos solicitar as informações e limpar a tela para exibi-las. O resultado ficaria assim:

a) Primeiro, aparecerão as solicitações de nome e idade. Vamos simular, mais uma vez, que o usuário informou os dados normalmente:

```
Informe seu nome: Edson Dutra
Informe sua idade: 52
```

b) Após isto, para exibir as informações seguintes, o programa irá limpar tudo o que apareceu na tela até agora, para exibir as próximas informações. Assim, ao invés de aparecer isto na tela (como aconteceu anteriormente),

```
Informe seu nome: Edson Dutra
Informe sua idade: 52

Confira seus dados:
Seu nome é Edson Dutra
Sua idade é 52
```

irá aparecer somente isto:

```
Confira seus dados:
Seu nome é Edson Dutra
Sua idade é 52
```

O recurso de limpar a tela – também conhecido como "apagar" a tela – pode ser utilizado sempre (e quantas vezes) que for necessário. Basta um comando para realizar esta ação: **clrscr;**

Esquisito, não? Na verdade, este comando é uma abreviação de "clear screen", cuja tradução significa "limpar a tela".

Vamos ver como ficará o programa onde, após solicitar as informações, a tela será limpa:

```
 1  Program ExemploLinhaEmBranco;
 2  var
 3         Nome : String;
 4         Idade : Integer;
 5  Begins
 6         Write('Informe seu nome: ');
 7         Readln(Nome);
 8         Write('Informe sua idade: ');
 9         Readln(Idade);
10         clrscr;
11         Writeln('Confira seus dados:');
12         Writeln('Seu nome é ', Nome);
13         Writeln('Sua idade é ', Idade);
14  End.
```

Fácil e simples, não é mesmo? A partir de agora, sinta-se à vontade para utilizar o **clrscr** sempre que achar necessário.

14 OPERADORES

Os operadores aritméticos e relacionais continuam exatamente iguais aos do algoritmo. Os operadores lógicos, portanto, são os únicos que mudam. Da mesma forma que acontece com muitos comandos entre algoritmos e a linguagem de programação, os operadores lógicos sofrem uma tradução para o inglês. Confira no quadro a seguir como ficam estes operadores convertidos para Pascal.

Operador	Em algoritmo (Portugol)	No Pascal
Conjunção	e	and
Disjunção	ou	or
Negação	não	not

Alguns exemplos de condições que utilizam operadores lógicos (obviamente, em conjunto com operadores relacionais):

```
(Salario > 1000) and (Salario < 2000)
(not Continua)
(Opcao = 1) or (Opcao = 2)
```

RESUMO DO TÓPICO 1

- Começamos, neste tópico, a estudar uma linguagem de programação. Foi escolhida a linguagem Pascal por ser de fácil aprendizado.

- O nome desta linguagem foi uma homenagem ao filósofo e matemático Blaise Pascal, considerado o pai da Ciência da Computação por ter inventado a considerada primeira máquina de calcular mecânica.

- Entre os vários ambientes de programação existentes, utilizaremos o Pascalzim, por ser de simples execução.

- Este ambiente permite uma boa visualização do código-fonte através de cores e também permite executar nossos programas.

- No ambiente Pascalzim, uma forma de executar um programa é pressionar a tecla F9.

- Estudamos neste tópico, também, como ficaram os diversos comandos que já havíamos visto nos algoritmos, porém, agora, em uma linguagem de programação. A linguagem é quase que uma tradução de um algoritmo para o inglês, com pequenas diferenças de pontuação, regras e alguns recursos e comandos a mais.

- Vimos a importância do uso dos tipos de dados para otimização da memória.

- Também aprendemos o conceito de *case sensitive*, que é a diferenciação de letras maiúsculas e minúsculas por parte de uma linguagem de programação. Por exemplo, a palavra "quantidade" é diferente de "Quantidade". A linguagem Pascal não é *case sensitive*, ou seja, não faz diferença uma palavra ser escrita em letras maiúsculas ou minúsculas, quando se trata de comandos da própria linguagem.

AUTOATIVIDADE

1 Por que foi escolhida a linguagem Pascal para o aprendizado de linguagens de programação?

2 Esta linguagem tem este nome em homenagem ao Blaise Pascal. Por que foi prestada esta homenagem a ele?

3 Qual é o atributo pelo qual Blaise Pascal é conhecido?

4 Quem criou a linguagem Pascal? Com que objetivo?

5 Cite dois exemplos de ambientes de programação para Pascal.

6 O que é *case sensitive*?

7 Cite uma importância dos tipos de dados em uma linguagem de programação.

TÓPICO 2

ESTRUTURAS DE SELEÇÃO

1 INTRODUÇÃO

Todas as estruturas estudadas nos algoritmos (seleção, repetição etc.) serão revistas, porém agora convertidas para a linguagem de programação.

Uma das principais diferenças encontra-se na sintaxe. Temos que nos preocupar intensamente com as pontuações da linguagem. Esquecendo um ponto e vírgula, por exemplo, o programa já não irá executar.

Junto com cada estrutura que iremos estudar, será especificada detalhadamente a pontuação necessária exigida pela linguagem de programação para a tal estrutura. No entanto, podemos adiantar que o comando **Fim** (ou, a partir de agora, **End**) de, praticamente, qualquer estrutura, será acompanhando por um ponto e vírgula (**End;**).

2 ESTRUTURA DE SELEÇÃO IF-THEN

É a estrutura correspondente ao **Se-então**. As principais diferenças entre a sintaxe do **Se-então** para o **If-Then** estão no uso do ponto e vírgula.

Vamos ver uma situação básica sobre a diferença principal (o já mencionado, ponto e vírgula).

```
 1  Program Primeiro_If;
 2  var
 3        Numero : Integer;
 4
 5  Begin
 6        Write('Informe um número positivo: ');
 7        Readln(Numero);
 8
 9        { Se o número for menor do que zero, ele não é positivo }
10        If (Numero < 0) Then
11        Begin
12              Write('O número não é positivo.');
13        End;
14
15  End.
```

Como pudemos constatar, o **End** vem acompanhado de um ponto e vírgula. Isto significa, para a linguagem de programação, a estrutura do comando **If-Then** termina ali (linha 13).

Há mais um comando utilizado junto ao **Se-então**, você lembra? É o comando **Senão**. Na linguagem de programação, este comando é o **Else**.

Agora que já sabemos como ficam todos os comandos do **Se-então** convertidos para a linguagem de programação, vamos ver um exemplo mais completo da estrutura **If-Then** e já aproveitaremos para prestar atenção nesta nova pontuação a ser utilizada.

Basicamente, o programa a seguir solicitará um número e, em seguida, informará se o número fornecido é maior do que 5, se é menor do que 5, ou se é o próprio 5.

```
1  Program Exemplo_If_1;
2  var
3              Numero : Integer;
4  Begin
5              Write('Informe um número: ');
6              Readln(Numero);
7
8
9              If (Numero > 5) Then
10             Begin
11                   Write('O número é maior do que 5.');
12             End
13             Else If (Numero < 5) Then
14             Begin
15                   Write('O número é menor do que 5.');
16             End
17             Else
18             Begin
19                   Write('O número é 5.');
20             End;
21
22
23 End.
```

Vamos visualizar a estrutura do **If-Then** como três blocos distintos:

a) Linhas 8 a 11: verifica se o valor da variável **Numero** é maior do que 5.

b) Linhas 12 a 15: verifica se o valor da variável **Numero** é menor do que 5.

c) Linhas 16 a 19: não faz verificação, pois, se o valor não é nem maior nem menor do que 5, só poderá ser o próprio valor 5.

Se você analisou bem o programa, poderá estar pensando: "Mas não foi dito que todos os **End** serão acompanhados de um ponto e vírgula?".

Bom, foi comentado que o **End** de praticamente qualquer estrutura, mas não de todas.

Você consegue deduzir por que os comandos **End** das linhas 11 e 15 não possuem ponto e vírgula?

Isto acontece porque a estrutura do **If-Then** (que começa na linha 8) só termina na linha 19. Os **End** das linhas 11 e 15 apenas finalizam um bloco de comandos, porém a estrutura completa do **If-Then,** como acabamos de explicar, terminará somente na linha 19. Nesta linha, há o comando **End** que realmente finaliza a estrutura **If-Then** iniciada na linha 8. Por isto, este último **End** vem acompanhado pelo ponto e vírgula.

3 A REGRA DO BEGIN/END

Há uma regra bastante interessante sobre o uso do **Begin** e **End** nas estruturas. Esta regra, basicamente, é a seguinte:

Se houver apenas uma instrução dentro de uma estrutura, esta estrutura não precisa do **Begin** e **End**. Se houver mais de uma instrução dentro de uma estrutura, é obrigatório o uso do **Begin** e **End**.

Vamos ver como fica o mesmo programa anterior (`Exemplo_If_1`), porém sem os **Begin** e **End** que não são obrigatórios.

```
1  Program Exemplo_If_2;
2  var
3         Numero : Integer;
4  Begin
5         Write('Informe um número: ');
6   Readln(Numero);
7
8         If (Numero > 5) Then
9               Write('O número é maior do que 5.')
10        Else If (Numero < 5) Then
11              Write('O número é menor do que 5.')
12        Else
13              Write('O número é 5.');
14
15 End.
```

Ainda temos os 3 blocos de comandos distintos:

a) Linhas 8 e 9

b) Linhas 10 e 11

c) Linhas 12 e 13

Cada um destes blocos de comandos possui apenas uma instrução. Por isto, como já explicado, não é necessário o uso do **Begin** e **End**.

É importante analisarmos mais um detalhe. Desde o início, aprendemos que as instruções precisam ter um ponto e vírgula. Como é, então, que as instruções das linhas 9 e 11 estão sem ponto e vírgula?

Esta situação segue o mesmo conceito dos **End** sem ponto e vírgula. O **if** começa na linha 8 e "termina" na linha 9; mas este "termina" significa que o bloco de comandos pertencentes a esta primeira parte do **if** é que termina, não a estrutura completa do **if**. A estrutura continua após a execução da instrução da linha 9. Como a estrutura ainda não acabou, não recebe ponto e vírgula.

Note que a estrutura continua (após a linha 9). Isto não quer dizer que a execução do **IF** continua. Por exemplo: Se o número for maior do que 5, ou seja, a condição da linha 8 é verdadeira. A execução, então, processará a linha 9 e, ao "esbarrar" no **else** da linha 10, já irá saltar para a linha 14.

Voltando ao programa `Exemplo_If_1`, veja que as instruções dentro dos blocos de **If** contêm ponto e vírgula. Isto porque quem delimita os blocos de comandos são, novamente, o **Begin** e **End**. Para gravar melhor este conceito, mantenha em mente esta afirmação: Quando utilizarmos **Begin/End**, cada instrução individual dentro deste bloco será acompanhada por ponto e vírgula.

Estes conceitos também são válidos para as demais estruturas, tanto de seleção quanto repetição.

4 O USO DO ELSE

É importante termos consciência do uso deste comando no "encadeamento" entre os blocos de **If-Then** que pertencem a uma mesma estrutura.

Perceba a estrutura já utilizada no programa anterior (`Exemplo_If_2`), sem o uso do **Else**:

```
1  Program Exemplo_If_2;
2  var
3         Numero : Integer;
4  Begin
5         Write('Informe um número: ');
6         Readln(Numero);
7
8         If (Numero > 5) Then
9               Write('O número é maior do que 5.');
10        If (Numero < 5) Then
11              Write('O número é menor do que 5.');
12        If (Numero = 5) Then
13              Write('O número é 5.');
14
15 End.
```

Como retiramos os **Else**, cada **If-Then** tornou-se uma estrutura independente (antes, tínhamos uma única estrutura, com vários **If-Then/Else** encadeados).

Neste caso, precisamos adicionar mais um **If-Then** em substituição ao **Else**, na linha 12 e também colocar um ponto e vírgula nas linhas 9 e 11. Tudo bem, mas não é este o ponto importante que vamos analisar.

Veja que, mesmo retirando os **Else**, a estrutura funcionará corretamente. Suponhamos que o número informado tenha sido o 10.

Quando a execução chegar à linha 8, a condição testada será verdadeira, pois o valor da variável **Numero** (que em nossa simulação é 10) realmente é maior do que 5. Isto significa que a linha 9, que é uma instrução pertencente ao **If-Then** da linha anterior, será executada. Com isto, aparecerá na tela a mensagem "O número é maior do que 5".

Como as estruturas **If-Then** seguintes não estão encadeadas com o **If-Then** anterior, a execução, após passar (executar) pela linha 9, passará para a linha 10. Neste caso, a condição será falsa, pois **Numero** não é menor do que 5. A execução, então, pulará a linha 11 e passará para a linha 12. Nesta linha, a condição também será falsa, pois **Numero** não é igual a 5. A execução pulará a linha 13 e finalizará o programa.

Vamos resumir o que aconteceu, aos olhos do usuário:

O programa solicitou um valor e o usuário informou 10. Em seguida, apareceu na tela uma mensagem informando que o número é maior do que 5.

Na visão do usuário, o programa funcionou, não é mesmo? Correto, funcionou. Porém, preste atenção no que o UNI tem a nos dizer:

Quando construímos programas, temos sempre que ter em mente o seguinte: Não basta fazer um programa funcionar, precisamos fazê-lo funcionar da melhor forma!

Se retirar os **Else**, já vimos que o programa vai funcionar. Mas não da melhor forma. Vamos ver por quê:

Quando a execução chegou à linha 8, verificou que o valor da variável **Numero** realmente era maior do que 5 e entrou a execução neste primeiro **If-Then**.

Após isto, o programa não precisa mais verificar se o valor da variável é menor ou se é igual a 5. Porém, foi isto que aconteceu (e pudemos constatar isto na interpretação que fizemos logo anteriormente).

O usuário nem percebeu, porque este processamento aconteceu muito rapidamente, porém é um tempo que o computador parou para analisar (nas linhas 10 e 12).

Isto praticamente não faz diferença em um pequeno programa como este. Porém, se for um programa que processa muitas informações, esse tempo a mais que o programa leva, para verificar uma condição (por exemplo) sem necessidade, pode começar a fazer diferença.

Podemos até chegar a pensar: “Mas, se num programa pequeno assim não faz diferença, por que nos preocuparmos com isso?”.

Bom, um programador sempre terá que pensar da forma como o UNI nos deu a dica. Independente do tamanho do programa, sempre deverá pensar em fazê-lo funcionar da melhor forma.

Agora que vimos todos os conceitos da estrutura **If-Then**, além de dicas que servem para outras estruturas também, vamos ver alguns exemplos de programas funcionando com esta estrutura. Alguns deles já estudamos nos algoritmos, mas será interessante ver como ficam construídos em Pascal.

Programa que lê um número e informa se este número é par ou ímpar.

```
1  Program ParImpar;
2  var
3        Numero : Integer;
4
5  Begin
6        Write('Informe um número: ');
7        Readln(Numero);
8
9        { Se o resto da divisão de qualquer número por 2 for
10 zero, o número é par }
11       If (Numero mod 2 = 0) Then
12       Begin
13             Write('O número ',Numero,' é par.');
14       End
15       Else { Se o resto não for zero, o número é ímpar }
16       Begin
17             Write('O número ',Numero,' é ímpar.');
18       End;
19
20 End.
```

Mais um problema:

Programa que lê uma idade e indica em que faixa etária esta idade está, de acordo com o quadro a seguir:

Faixa de idade	**Faixa etária**
0 a 11 anos	Infância
12 a 21	Adolescência
22 a 65	Adulto
65 ou mais	Idoso

Desenvolvimento:

```
 1  Program FaixaEtariaPorIdade;
 2  var
 3         Idade : Integer;
 4
 5  Begin
 6         Write('Idade: ');
 7         Readln(Idade);
 8
 9         { Verifica se foi informada uma idade abaixo de zero }
10         If (Idade < 0) then
11         Begin
12                Write('Você informou um valor negativo');
13         End
14         { A idade está entre 0 e 11 anos }
15         Else If (Idade >= 0) and (Idade <= 11) Then
16         Begin
17                Write('Você está na fase infância.');
18         End
19         { Idade entre 12 e 21 (inclusive) }
20         Else If (Idade >= 12) and (Idade <= 21) Then
21         Begin
22                Write('Você está na fase da adolescência.');
23         End
24         { Idade entre 22 e 65 (inclusive) }
25         Else If (Idade >= 22) and (Idade <= 65) Then
26         Begin
27                Write('Você está na fase adulta.');
28         End
29         { Idade acima de 65 anos }
30         Else If (Idade > 65) Then
31         Begin
32                Write('Você está na fase idosa.');
33         End;
34
35  End.
```

Lembra o que o UNI nos informou há pouco? Que temos sempre que procurar fazer um programa não só funcionar mas funcionar da melhor maneira? Com base nisto, vamos analisar este programa a fim de aprender algumas dicas interessantes e também melhorá-lo no que for possível.

Primeiramente, vamos assumir que o usuário informou a idade 9.

Começaremos analisando a linha 10. Se o usuário tivesse informado, por exemplo, -15, o valor é negativo. Neste caso, o programa trata esta situação, informando que o número fornecido foi negativo.

Sempre que se constrói um programa, devem-se analisar todas as possibilidades e "tratar" cada situação, fazendo com que, por mais que o usuário informe valores não condizentes, o programa não dê erros.

Vamos à linha 15. Resumidamente, esta linha verifica se a idade está **entre** 0 e 11 (inclusive). Isto está correto, pois é este o período correspondente à infância, e o programa vai funcionar corretamente.

Porém, vamos tentar responder à seguinte pergunta: "É realmente necessário verificar se a **Idade** é maior ou igual a 0 (zero)?".

Vejamos: se a execução do algoritmo chegar à linha 15, é porque a condição da linha 10 resultou em "falso" (porque, se resultasse "verdadeiro", seria executada a linha 12 e, devido à utilização dos **Else**, a execução passaria, depois da linha 12, direto para a linha 34 e então finalizaria).

Se a condição da linha 10 é falsa, o número não é menor do que 0 (zero). Se o número for maior do que 0 (que é o caso do nosso exemplo – 9), a execução passará para a linha 15.

O ponto onde queremos chegar é: perceba que não é necessária a condição (Idade >= 0). Se a execução chegou a esta linha (15), é porque, através da linha 10, já se identificou que a idade seria maior do que zero. Assim, basta apenas utilizar a condição (Idade <= 11).

Você pode se perguntar: Mas, se funciona corretamente utilizando a condição (Idade >= 0), por que não deixar a condição ali?

Cada condição a ser verificada pela execução do programa é uma fração de tempo a mais que o programa precisa parar para analisar. Se é um tempo a mais, e esse tempo é desnecessário, então a regra é sempre retirar este tipo de instrução.

A mesma coisa acontece com as estruturas seguintes. Vejamos a linha 20. Para estar na fase da adolescência, a idade tem que estar entre 12 e 21. A condição, então, está correta, irá funcionar. Porém, a parte da condição (Idade >= 12) também não é necessária. Novamente: se a execução chegar à linha 20, é porque a condição da linha 15 é falsa. Sendo falsa, a idade é maior do que 11. Se é maior do que 11, significa que só pode ser maior ou igual a 12.

Espero que tenha ficado esclarecida esta situação. Nunca se esqueça de pensar nestas possibilidades na hora de desenvolver um programa.

Vamos analisar, agora, mais uma situação. A condição da linha 30. Estamos verificando se a idade é maior do que 65. Mais uma vez, a pergunta: é necessário verificar se a idade é maior do que 65?

Se a execução chegar à linha 30, todas as outras condições foram falsas. Isto quer dizer que a idade não é menor do que zero, não é menor do que 11 nem menor do que 21 e também não é menor do que 65. A idade, então, somente poderá ser maior do que 65. Conclui-se que não é necessário verificar se a idade é maior, pois, repetindo, se a execução chegar a esta linha (30), a idade será maior do que 65. Basta, então, deixar somente o **Else** nesta linha.

Agora que otimizamos o programa, vamos ver como ficaria?

```
1   Program FaixaEtariaPorIdade;
2   var
3         Idade : Integer;
4
5   Begin
6         Write('Idade: ');
7         Readln(Idade);
8
9         { Verifica se foi informada uma idade abaixo de zero }
10        If (Idade < 0) then
11        Begin
12              Write('Você informou um valor negativo');
13        End
14        { A idade está entre 0 e 11 anos }
15        Else If (Idade <= 11) Then
16  Begin
17              Write('Você está na fase infância.');
18        End
19        { Idade entre 12 e 21 (inclusive) }
20        Else If (Idade <= 21) Then
21        Begin
22              Write('Você está na fase da adolescência.');
23        End
24        { Idade entre 22 e 65 (inclusive) }
25        Else If (Idade <= 65) Then
26        Begin
27              Write('Você está na fase adulta.');
28        End
29        { Idade acima de 65 anos }
30        Else
```

```
31          Begin
32                Write('Você está na fase idosa.');
33          End;
34
35    End.
```

Agora, já vimos todos os detalhes da estrutura **If-Then** e várias dicas sobre pontuações e outros conceitos. Vamos seguir adiante.

5 ESTRUTURA DE SELEÇÃO CASE

Corresponde à estrutura **Escolha-Caso**. Sua sintaxe é um pouquinho diferente de uma simples tradução. Veja a sintaxe:

```
Case <variavel> of
       valor1 : Instrução;
       valor2 : Instrução;
       Else Instrução;
End;
```

<variavel> é a variável a qual você quer tratar um valor.

valor1 e **valor2** são os valores propriamente ditos, que a variável pode ter e que você queira tratar.

Se a variável não tiver armazenando no momento nenhum dos valores que você quer tratar, a execução cairá no **Else**. Não é obrigatório o uso deste. Se o **Else** não for colocado e a variável não tiver nenhum dos valores tratados, simplesmente nada será executado pelo **Case** para esta situação.

Para o **Case**, também é válida a regra da quantidade de instruções, ou seja, se, para um determinado valor, mais de uma instrução precisar ser executada, é necessário utilizar o **Begin/End**.

Veja um exemplo da sintaxe da estrutura:

```
Case <variavel> of
       valor1 : Instrução;
       valor2 :
              Begin
                     Bloco de instruções
              End;
       Else Instrução;
End;
```

A seguir, um exemplo básico de um programa que solicita um número de 0 a 9 e exibe o número por extenso:

```
1  Program Extenso;
2
3  var
4         Numero : Integer;
5
6  Begin
7         Write('Número: ');
8         Readln(Numero);
9
10        Case Numero of
11              0 : write('Zero');
12              1 : write('Um');
13              2 : write('Dois');
14              3 : write('Três');
15              4 : write('Quatro');
16              5 : write('Cinco');
17              6 : write('Seis');
18              7 : write('Sete');
19              8 : write('Oito');
20              9 : write('Nove');
21              Else
22              Begin
23              writeln('O número informado não está entre 0 e 9.');
24                    write('Programa finalizado.');
25              End;
26        End;
27 End.
```

O único local onde foi usada mais de uma instrução foi no **Else**. Por isso, a necessidade do **Begin/End**.

RESUMO DO TÓPICO 2

- As estruturas **If-Then**, na linguagem Pascal, correspondem ao **Se-então** do algoritmo, com pequenas regras de sintaxe a mais, exigidas pela linguagem de programação.

- Nestas estruturas, é muito importante o uso do **Else** para encadear as estruturas **If-Then** que pertencem a um mesmo conjunto. Isto faz parte da "otimização" de código.

- É interessante estar atento(a) às regras de ponto e vírgula nas estruturas **If-Then** mais complexas. Às vezes, uma instrução ou mesmo um **End** não será acompanhado por um ponto e vírgula, pois a estrutura **If-Then** continua com mais blocos adiante.

- Os delimitadores de blocos **Begin/End** também possuem uma regra básica em seu uso: se eles delimitarem apenas uma instrução, não seria necessário utilizá-los. Já se delimitarem mais de uma instrução, é obrigatório seu uso.

- Uma dica muito importante que aprendemos é que não basta fazer um programa apenas funcionar. O programa sempre terá que ser escrito da melhor maneira possível.

- A estrutura de seleção **Case** não é apenas uma tradução do algoritmo para a linguagem. Esta estrutura contém consideráveis diferenças quanto à sua sintaxe em comparação ao algoritmo. Porém, sua funcionalidade é exatamente a mesma.

AUTOATIVIDADE

1 Qual é a regra básica sobre a utilização do **Begin/End** em uma estrutura: quando é obrigatório e quando não é?

2 Qual é a importância de utilizar o *"else"* para encadear uma estrutura *"if"*, ao invés de apenas utilizar estruturas *"if"* independentes?

3 Crie um programa que solicite o salário de uma pessoa. O programa deverá dar um reajuste de 9% no salário se este for até R$ 1.000,00. Se for maior, deverá dar um reajuste de 6%. Exibir o novo salário.

4 Uma loja concede pagamento a prazo em 3, 5 e 7 vezes. A partir do preço à vista de um determinado produto, calcule o preço total a pagar e o valor da prestação mensal, referentes ao pagamento parcelado. Se o pagamento for parcelado em três vezes, deverá ser dado um acréscimo de 12% no total a ser pago. Se for parcelado em 5 vezes, o acréscimo é de 22%. Parcelado em 10 vezes, acrescenta-se 38%.

5 Desenvolva um programa que solicite o cargo, o salário e a idade de uma pessoa e calcule o aumento que ela receberá de acordo com os dados a seguir:

Cargo	**Idade**	**Aumento**
1 - Programador	Até 20 anos	5%
1 - Programador	Acima de 20 anos	8%
2 – Analista de sistemas	Até 25 anos	8,5%
2 – Analista de sistemas	Acima de 25 anos	10%
3 – Qualidade de software	*Independente da idade*	7%

O programa deverá exibir o novo salário da pessoa.

Faça o programa de duas formas, cada uma utilizando uma estrutura de seleção diferente.

6 Faça um programa que solicite três valores correspondentes aos lados de um possível triângulo e informe:

- Se for um triângulo, exibir qual é o tipo (equilátero, isósceles, escaleno).
- Se for um triângulo retângulo, exibir uma mensagem informando.
- Se não for triângulo, exibir uma mensagem informando.

Regras matemáticas:

a) **Para que as medidas formem um triângulo**, cada um de seus lados deve ser maior que o valor absoluto (módulo) da diferença dos outros dois lados e menor que a soma dos outros dois lados. Resumo da regra:
| b - c | < **a** < b + c
| a - c | < **b** < a + c
| a - b | < **c** < a + b
b) **Para que seja um triângulo equilátero**, todos os seus lados devem ser iguais.
c) **Para que seja um triângulo isósceles**, deve possuir apenas dois lados iguais.
d) **Para que seja um triângulo escaleno**, todos os seus lados devem ser diferentes.
e) **Para que seja um triângulo retângulo**, a soma do quadrado entre dois lados deve ser igual ao quadrado do lado restante. Ex.: $a^2 = b^2 + c^2$. Note que um triângulo retângulo pode ser somente isósceles ou escaleno, nunca equilátero.

TÓPICO 3

ESTRUTURAS DE REPETIÇÃO

1 INTRODUÇÃO

Assim como acabamos de fazer com as estruturas de seleção, iremos rever os conceitos de estruturas de repetição, porém agora convertidas para uma linguagem de programação.

Tal conversão também é, basicamente, uma tradução dos comandos.

2 ESTRUTURA DE REPETIÇÃO WHILE-DO

Como a tradução já deixa claro, é a estrutura que corresponde ao **Enquanto-Faça**. Funciona exatamente da mesma forma e também possui as regras de **Begin/End**, porém é muito raro utilizar um **While-Do** sem **Begin/End**, pois, na maioria das situações onde é necessário o **While-Do**, a própria situação exige mais de uma instrução.

Vamos ver o exemplo de um programa que lê uma quantidade indefinida de idades e, ao final, o programa exibe quantas pessoas são maiores de idade e quantas são menores (considerando 18 anos como maioridade). A *flag* de saída será quando a idade for zero ou se for um valor negativo.

```
1 Program Exemplo_While_Do;
2
3 var
4        Idade, QtdMaiores, QtdMenores : Integer;
5
6 Begin
7        { Inicializa as quantidades com zero }
8        QtdMaiores := 0;
9  QtdMenores := 0;
10
11       { Solicita a idade pela primeira vez }
12       Write('Idade: ');
13       Readln(Idade);
14
15       { Se a idade for maior do que zero }
16       While (Idade > 0) do
17       Begin
18              { Se a idade for maior ou igual a 18 }
19              If (Idade >= 18) Then
```

```
20              Begin
21              { Conta mais um, na quantidade de maiores de idade }
22                      QtdMaiores := QtdMaiores + 1;
23              End
24              Else
25              Begin
26              {Conta mais um, na quantidade de menores de idade }
27                      QtdMenores := QtdMenores + 1;
28              End;
29
30              { Solicita a próxima idade }
31              Write('Idade: ');
32              Readln(Idade);
33      End;
34      { Exibe as quantidades }
35      Writeln(QtdMaiores, ' são maiores de idade.');
36      Writeln(QtdMenores, ' são menores de idade.');
37
38 End.
```

3 ESTRUTURA DE REPETIÇÃO FOR-DO

Correspondente ao **Para-faça**, também é utilizado, preferencialmente, para repetir instruções quando conhecemos a quantidade de vezes que a estrutura será executada.

A sintaxe é:

```
For <variavel> := <valor_inicial> To <valor_final> Do
Begin
      Instruções
End;
```

Um exemplo com valores:

```
 1  Program Exemplo_While_Do;
 2
 3  var
 4         I : Integer;
 5
 6  Begin
 7
 8         For I := 1 To 5 Do
 9         Begin
10                Writeln('Esta é a ',I,'a. vez que está executando.');
11         End;
12
13  End.
```

O resultado seria:

```
Esta é a 1a. vez que está executando.
Esta é a 2a. vez que está executando.
Esta é a 3a. vez que está executando.
Esta é a 4a. vez que está executando.
Esta é a 5a. vez que está executando.
```

Até agora, trabalhamos bastante com esta estrutura, fazendo a <variavel> passar de um valor menor para um maior. No exemplo anterior, o "I" (<variavel>, neste caso) passa de 1 até 5.

Um conceito importante é que, quando se deseja fazer a <variavel> passar de um número maior para um menor, não se pode utilizar o "**To**" entre os valores. Deve-se utilizar o **DownTo**. Exemplo:

```
 1  Program Exemplo_DownTo;
 2
 3  var
 4        I : Integer;
 5
 6  Begin
 7
 8        For I := 5 DownTo 1 Do
 9        Begin
10              Writeln('Valor de I: ',I);
11        End;
12
13 End.
```

O resultado deste programa será:

```
Valor de I: 5
Valor de I: 4
Valor de I: 3
Valor de I: 2
Valor de I: 1
```

4 ESTRUTURA DE REPETIÇÃO REPEAT-UNTIL

É equivalente ao **Repita-até**. Reforçando a característica desta estrutura: pelo menos, uma vez, as instruções contidas dentro desta estrutura serão executadas, ao contrário do **While-Do**, por exemplo, onde pode ser que as instruções nunca sejam executadas, pois a condição já pode ser falsa desde a primeira vez.

Programa exemplo:

```
 1   Program RepeatUntil;
 2
 3   var
 4          QtdVezes : Integer;
 5        Continua : String;
 6   Begin
 7
 8          QtdVezes := 0;
 9
10          Repeat
11
12                 QtdVezes := QtdVezes + 1;
13
14                 Writeln('Executou o repeat ', QtdVezes,' vez(es).');
15
16                 write('Deseja continuar? ');
17                 readln(Continua);
18
19          Until (Continua = 'n');
20
21   End.
```

Veja que, ao chegar à linha 10, nada é verificado para decidir se entrará na estrutura ou não. A execução simplesmente entrará, exibirá a quantidade de vezes que já entrou na estrutura e, em seguida, pergunta ao usuário se ele deseja continuar. Se ele informar que não deseja (continua = n), a condição será verdadeira. Quando a condição do **Until** de um **Repeat** é verdadeira, a estrutura termina.

5 O COMANDO *BREAK*

Já sabemos que uma estrutura de repetição irá parar, quando sua condição (*flag*) for falsa. Porém, em alguns casos, é necessário parar a estrutura (finalizar sua execução) antes de chegar em seu "**End**" e retornar para verificar se a condição é verdadeira ou falsa.

Para estes casos, existe o comando **break**. Sua finalidade é única: parar a execução de uma estrutura de repetição.

Exemplo:

```
 1  Program Pzim ;
 2
 3  var
 4        QuantidadeTotal, I : Integer;
 5        Continua : String;
 6
 7  Begin
 8
 9        Write('Informe a quantidade que deseja repetir: ');
10        Readln(QuantidadeTotal);
11
12        For I := 1 to QuantidadeTotal do
13        Begin
14
15              Writeln('O valor de "I" é ', I);
16              Writeln; { Pula uma linha na tela }
17
18              Write('Deseja prosseguir até ', QuantidadeTo-
19  tal, '? (s/n)');
20              Readln(Continua);
21
22              if (Continua <> 's') and (Continua <> 'S') then
23              Begin
24                    Break;
25              End;
26        End;
27
28  End.
```

É solicitado ao usuário quantas vezes a estrutura de repetição irá executar (linha 9 e 10). Vamos supor que o usuário informou 8.

Em seguida, esta execução inicia. Cada vez que a estrutura repetir, é exibido o valor atual de "**I**". Após exibir este valor, o programa pergunta se o usuário deseja continuar. Mesmo que a estrutura de repetição ainda não tenha chegado ao fim (8), se o usuário informar que não quer mais continuar (informar, por exemplo, a letra "**n**" para **Continua**), o programa irá identificar isto na linha 22 (o valor da variável **Continua** não é, nem "**s**" minúsculo, nem "**s**" maiúsculo),

a execução entrará na estrutura **If-Then** e executará o **break**. Neste momento, a execução cai para a linha 27, ou seja, depois do **End** da estrutura de repetição **For-Do**.

Reforçando: mesmo que a estrutura **For-Do** não tenha executado até o fim (8, no exemplo), ela (a estrutura) poderá terminar dependendo do dado que o usuário informar na linha 20.

RESUMO DO TÓPICO 3

- Revimos as estruturas de repetição, já estudadas nos algoritmos, porém agora convertidas para a linguagem de programação Pascal.

- Constatamos que qualquer uma das estruturas (**While-Do**, **For-Do** e **Repeat-Until**) são apenas traduções do "portugol" para o inglês, para que estejam convertidas para a linguagem, porém o funcionamento é exatamente o mesmo.

- Estudamos, também, um comando novo até então, o **break**. Este comando é utilizado quando se deseja parar a execução de uma estrutura de repetição mesmo que sua condição não tenha atingido o resultado "falso".

AUTOATIVIDADE

1 Onde é utilizado e para que serve o **DownTo**?

2 Qual é a diferença básica entre as estruturas de repetição **While-Do** e **Repeat-Until**?

3 Refaça o exercício que verifica se um número é primo, agora na linguagem de programação. Obs.: procure não consultar o exercício feito em algoritmo. Você vai perceber que, mesmo um exercício que já foi resolvido, trará desafios ao tentar resolvê-lo novamente.

4 Desenvolva um programa que leia, de uma quantidade indeterminada de pessoas, os seguintes dados:

a) Nome
b) Salário
c) Idade

Ao final, o programa deverá exibir:
a) Nome da pessoa mais velha
b) Nome e salário da pessoa que tem o salário mais alto
c) Salário da pessoa mais nova

5 Na matemática, um **número perfeito** é um número inteiro para o qual a soma de todos os seus divisores positivos próprios (excluindo ele mesmo) é igual ao próprio número.

Construa um programa que leia um número e informe se o número é perfeito ou não.
Por exemplo, o 6 é um número perfeito, pois 1 + 2 + 3 = 6. No caso, 1, 2 e 3 são os únicos divisores de 6.
Exemplos de números perfeitos: 6; 28; 496; 8128; 33.550.336; 8.589.869.056.

6 Em uma empresa, foram solicitadas informações de cada funcionário para realizar uma pesquisa. Para cada funcionário, é informado o nome e quantos filhos ele possui. Ao final, a pesquisa deverá informar:

a) Nome da pessoa que mais possui filhos
b) Quantos filhos a pessoa do item acima (item a) possui
c) Se existem na empresa mais pessoas com filhos ou mais pessoas sem filhos (apresentar uma mensagem informando)
d) Percentual de pessoas sem filhos

7 Um órgão governamental fará um recadastramento de pessoas físicas e jurídicas para obter informações sobre impostos. Para cada pessoa ou empresa, serão solicitadas as seguintes informações:

- Tipo de pessoa (física/jurídica)
- Salário ou faturamento bruto mensal
As pessoas físicas pagarão 3,5% do seu salário para o imposto.
As pessoas jurídicas pagarão 5,2%.
Para cada pessoa, exibir quanto de imposto ela irá pagar.

O programa deverá fornecer as seguintes informações:
a) Foram recadastradas mais pessoas físicas ou jurídicas?
b) Qual é a percentagem de pessoas físicas que fizeram recadastramento?
c) Qual foi o imposto (em valor) mais alto pago por uma pessoa jurídica?
d) Qual é a média geral de imposto pago por pessoas físicas?

* Não se sabe quantas pessoas virão fazer o recadastramento. O algoritmo deverá estar preparado para isso.

8 Abriu um novo cinema na cidade e, no dia da estreia, estão cobrando 10 reais a entrada normal e 5 para estudantes. O cinema deseja saber algumas informações sobre a primeira sessão (a sessão de estreia).

Para isto, você fará um programa em que, quando uma pessoa comprar um ingresso para esta sessão, o atendente informe se a pessoa é estudante ou não. Você define a *flag* de saída.

Após vendidos todos os ingressos da sessão, o programa deverá exibir:
a) Quanto o cinema arrecadou na sessão?
b) Qual foi o percentual de estudantes em relação ao total de pessoas?
c) Foi arrecadado mais com entradas normais ou com estudantes?
d) Foram comprados mais ingressos de estudante ou normais?

TÓPICO 4

ARRAYS

1 INTRODUÇÃO

Os *arrays* (pronuncia-se "arheis") nada mais são do que os vetores de matrizes.

Os vetores são chamados ***arrays* unidimensionais**, pois só possuem uma "dimensão", conforme já estudamos nos algoritmos.

As matrizes são chamadas ***arrays* bidimensionais**, por possuírem duas dimensões (linha e coluna), também já visto nos algoritmos.

Mesmo em uma linguagem de programação, alguns programadores mantêm o hábito de falar vetor ou matriz ao invés de *array*. Mesmo que a palavra, na linguagem, seja *array*, a estrutura em si não deixa de ser um vetor nem uma matriz.

2 DECLARAÇÃO E SINTAXE

A declaração de qualquer um dos tipos de *arrays* é praticamente igual. Nos algoritmos, quando declarávamos um vetor, utilizávamos a palavra "vetor" para tal declaração. O mesmo acontecia com matrizes; a palavra "matriz" era utilizada na declaração.

Aqui, como não há distinção de palavras entre vetor e matriz (as duas estruturas são chamadas de *array)*, a diferença está apenas na dimensão declarada.

Para visualizar a sintaxe, vamos declarar um *array* unidimensional chamado **vet** e um *array* bidimensional chamado **mat**.

Primeiro, vamos relembrar como era no algoritmo:

```
Vet : Vetor [1..10] de inteiro;
Mat : Matriz [1..4, 1..3] de real;
```

Quanto à palavra "de", utilizada na declaração, basta traduzir para o inglês: "of". As palavras "Vetor" e "Matriz", como dito há pouco, são convertidas para *"Array"*. Veja como fica:

```
Vet : Array [1..10] of Integer;
Mat : Array [1..4, 1..3] of Real;
```

Todas as demais estruturas relacionadas aos *arrays* permanecem iguais aos algoritmos, porém com a devida conversão para a linguagem (por exemplo, costuma-se percorrer um *array* com uma estrutura **For-Do**, assim como se costumava percorrer com **Para-faça** nos algoritmos).

Vamos a um exemplo, utilizando apenas a estrutura do tipo vetor:

Desenvolva um programa que leia 7 valores e mantenha-os armazenados. Após armazenados, o programa deverá percorrer os valores e exibir:

a) qual é o maior valor;
b) qual é o menor valor;
c) a média dos valores.

```
1   Program ExemploArrayUnidimensional;
2
3   const Qtd = 7;
4
5   var
6          Valores : Array [1..Qtd] of Real;
7          Maior, Menor, Soma : Real;
8          I : Integer;
9
10  Begin
11
12         For I := 1 to Qtd do
13         Begin
14                Write('Valor: ');
15                Readln(Valores[I]);
16 End;
17
18 {Assume o primeiro valor do vetor como sendo o maior e o menor}
19        Maior := Valores[1];
20        Menor := Valores[1];
21
22        For I := 1 to Qtd do
23        Begin
24               { Armazena o maior valor }
25               If (Valores[I] > Maior) Then
26               Begin
27                     Maior := Valores[I];
28               End
29               { Se não for o maior, verifica se é o menor }
```

```
30              Else If (Valores[I] < Menor) Then
31              Begin
32                  Menor := Valores[I];
33              End;
34
35              { Soma os valores para calcular a média }
36              Soma := Soma + Valores[I];
37
38        End;
39
40        { Exibe os resultados }
41        Writeln('Maior valor: ', Maior);
42        Writeln('Menor valor: ', Menor);
43        Writeln('Média: ', (Soma / Qtd));
44
45 End.
```

Agora, veremos um programa utilizando matriz.

Faça um programa que solicite valores ao usuário para alimentar duas matrizes de dimensão 3 x 2. Em seguida, o programa deverá realizar a soma das duas matrizes, armazenar a soma em uma terceira matriz e exibir esta matriz resultante na tela.

```
1 Program ExemploArrayBidimensional;
2
3 const
4        QtdLin = 3;
5        QtdCol = 2;
6
7 var
8    Mat1 : Array [1..QtdLin, 1..QtdCol] of Real;
9    Mat2 : Array [1..QtdLin, 1..QtdCol] of Real;
10   MatSoma : Array [1..QtdLin, 1..QtdCol] of Real;
11   L, C : Integer;
12
13 Begin
14
15        { Alimenta a matriz 1 }
16        For L := 1 to QtdLin do
17        Begin
18               For C := 1 to QtdCol do
19               Begin
20                      Write('Mat1[',L,',',C,']: ');
21                      Readln(Mat1[L,C]);
22               End;
23        End;
24
25        clrscr;
26
27        { Alimenta a matriz 2 }
28        For L := 1 to QtdLin do
```

```
29        Begin
30             For C := 1 to QtdCol do
31               Begin
32                     Write('Mat2[',L,',',C,']: ');
33                     Readln(Mat2[L,C]);
34               End;
35        End;
36
37        { Calcula a Matriz Soma }
38        For L := 1 to QtdLin do
39        Begin
40             For C := 1 to QtdCol do
41               Begin
42                     MatSoma[L,C] := Mat1[L,C] + Mat2[L,C];
43               End;
44        End;
45
46        { Exibe a Matriz Soma }
47
48        For L := 1 to QtdLin do
49        Begin
50             For C := 1 to QtdCol do
51               Begin
52                     Write(MatSoma[L,C],'       ');
53               End;
54               writeln;
55        End;
56
57  End.
```

Para exibir matrizes na tela, em um formato "matriz matemática", da forma que estamos acostumados a ver, é bem fácil. Veja as dicas a seguir.

Da linha 50 à 53 encontra-se a estrutura que mostra todos os valores de uma única linha. Podemos entender desta forma também: estrutura que mostra todas as colunas de uma linha. Perceba, na linha 52, que ao exibir cada valor armazenado em **MatSoma**, estamos exibindo junto um espaço. Isto fará com que cada valor exibido a seguir estará a "um espaço igual" de distância do valor anterior.

Após exibir todos os valores de uma linha, ou seja, após finalizar esta estrutura **For-Do**, será executada a linha 54. Resumidamente, o **writeln** irá quebrar uma linha na tela, pois não exibirá nada e, devido ao "**ln**", fará esta quebra.

Assim, quando a execução chegar à linha 55 e então retornar para a linha 48, onde iniciará a imprimir as colunas da linha seguinte, estes valores serão exibidos já na linha de baixo.

Isto faz com que cada linha da matriz seja exibida em uma linha na tela, e cada coluna esteja a um espaço igual de distância.

É importante saber que este espaço é feito através de um *"tab"*, ou seja, é utilizado o espaço proporcionado pela tecla *"tab"*.

Como no exemplo do programa anterior, a matriz é de dimensão 3 x 2, a exibição aparecerá semelhante ao *layout* a seguir (simulando alguns valores):

```
1     5
10    3
8     21
```

RESUMO DO TÓPICO 4

- Os *arrays*, na linguagem de programação, correspondem ao mesmo conceito de vetores e matrizes. Os vetores são *arrays* unidimensionais e as matrizes são *arrays* bidimensionais. Contudo, na prática, não é errado chamar um *array* unidimensional de vetor ou um bidimensional de matriz, mesmo em linguagens de programação.

- Da mesma forma que estávamos acostumados nos algoritmos, para percorrer um *array*, a estrutura **For-Do** é bastante recomendada.

- É interessante exibir matrizes na tela, em um formato visualmente semelhante ao que estamos acostumados a escrever na matemática.

- Para isto, há uma pequena dica que consiste em, a cada valor exibido, exibir, também, um espaço formado por um caractere *"tab"* e, após exibir todos os elementos de uma linha, quebrar para a próxima linha com auxílio de, por exemplo, um comando **writeln**.

AUTOATIVIDADE

1 Faça um programa que solicite e armazene 5 valores. Em seguida, o algoritmo deverá exibir os valores na ordem inversa em que foram lidos.

2 Desenvolva um programa que leia 10 números e armazene-os em um *array* do tipo vetor. Após a leitura destes números, o algoritmo deverá solicitar outro número para fazer uma busca.

O programa deverá percorrer os números lidos procurando pelo "número de busca". Ao final, o programa deverá exibir quantas vezes o número de busca está presente no *array* ou apresentar uma mensagem informando se o número não está no *array*.

3 Crie um programa que solicite 8 valores e armazene-os em um *array* (vetor). O programa, sem seguida, deverá ler mais dois valores (A e B) e apresentar a soma dos valores armazenados nos índices A e B do *array*.

4 Faça um programa que leia 10 valores e armazene-os em um *array*. O programa deverá ordenar (em ordem crescente) os valores, dentro do próprio *array*. Ao final, o programa percorrerá o *array* e exibirá os valores na tela, já ordenados.

5 Crie um programa que leia e armazene 10 valores em um *array* "A" e mais 10 valores em um *array* "B". O programa deverá preencher um terceiro *array* onde, nos índices pares, serão armazenados os valores do *array* "A". Nos índices ímpares, deverão ser armazenados os valores do vetor "B".

6 Desenvolva um programa que preencha uma matriz quadrada 5 x 5. O programa deverá exibir todos os valores que estão na diagonal principal da matriz.

7 Reescreva o mesmo programa anterior, porém o programa deverá exibir os valores que estão fora da diagonal principal.

8 Reescreva novamente o programa anterior, modificando-o para que exiba somente os valores que estão acima da diagonal principal.

9 Escreva um programa que consiga armazenar uma matriz de até 10 x 10. No início, o programa solicitará ao usuário que informe a dimensão da matriz que deseja trabalhar (se ultrapassar 10 x 10, o programa deverá alertar que não é permitido e solicitar novamente, até que o usuário informe um valor dentro do permitido). A matriz somente poderá ser quadrada. Após ler a dimensão, o programa deverá solicitar valores para preencher a matriz. Depois de preenchida, o programa deverá exibir todos os valores armazenados na diagonal secundária.

10 Faça um programa que leia o nome de, no mínimo, 3 cidades e armazene-os em um *array* (vetor). O programa deverá solicitar valores para alimentar uma matriz onde cada casa da matriz corresponde à distância entre duas cidades. Exemplo:

Distância em km ↘		Blumenau	Indaial	Timbó
Blumenau		0	20	25
Indaial		20	0	5
Timbó		25	5	0

Pode-se perceber que podemos fazer uma relação entre os índices do vetor e as linhas e colunas da matriz. Por exemplo: casa 1 x 3 da matriz, onde está o valor 25, corresponde à distância entre Blumenau (índice 1 do vetor) e Timbó (índice 3 do vetor).
A matriz contém somente os números correspondentes às distâncias. O vetor servirá para orientar o usuário, para ele saber qual é a distância que estará informando.
Quando o programa pedir uma distância, deverá exibir as cidades referentes a esta distância, por exemplo: "Informe a distância entre Blumenau e Indaial: ".

O programa não deverá:
a) Solicitar a distância entre uma mesma cidade (Blumenau x Blumenau)
b) Solicitar a distância entre duas cidades que já foram informadas. Exemplo: Após solicitar a distância entre Blumenau e Timbó, o programa não deverá solicitar a distância entre Timbó e Blumenau, pois a distância será a mesma.

No caso "a)", o programa deverá preencher, automaticamente, o valor 0 (zero).
No caso "b)", o programa deverá preencher, automaticamente, o valor já informado anteriormente, correspondente à distância entre estas duas idades.

Após a matriz preenchida, o programa deverá exibir:
a) Quais são as duas cidades mais próximas e qual é a distância entre elas.
b) Quais são as duas cidades mais distantes e qual é a distância entre elas.

FUNCTIONS E PROCEDURES

1 INTRODUÇÃO

As palavras *functions* e *procedures* já nos levam a deduzir que estamos falando dos subalgoritmos (funções e procedimentos), que aqui (em uma linguagem de programação) costuma-se chamar de subprogramas ao invés de subalgoritmos.

Neste caso a conversão também é muito simples, pois, como várias outras estruturas, esta conversão será apenas uma tradução para o inglês.

2 ESTRUTURA DE UMA *PROCEDURE*

Como já mencionado, é apenas uma tradução. Confira:

```
Procedure <nome> (<parâmetros>);
const
       <constantes>
var
       <variáveis>

Begin
       <instruções>
End;
```

As características a seguir não são obrigatórias, utilizam-se somente se necessárias:

- <parâmetros>
- Área de constantes
- Área de variáveis

A palavra "*procedure*" pronuncia-se "procídâr".

Veja alguns exemplos:

Esta *procedure* apenas exibe um menu de opções.

```
Procedure ExibeOpcoes;
Begin
       Writeln('1 - Incluir');
       Writeln('2 - Excluir');
       Writeln('3 - Alterar');
       Writeln('4 - Sair');
End;
```

Esta *procedure* recebe um valor e exibe este valor ao quadrado.

```
Procedure Quadrado(valor : Real);
Begin
       Write(valor * valor);
End;
```

Se observarmos à risca, a palavra *procedure* é do gênero masculino, pois é o procedimento (e não a procedimento). Porém, acabou-se popularizando, no Brasil, tratar a estrutura de Procedimento como se fosse do gênero feminino. Assim, falamos "a *procedure*" ou "uma *procedure*". Como acabamos de informar, é a forma popular de falar, mesmo não sendo a correta.

3 ESTRUTURA DE UMA *FUNCTION*

Esta estrutura também é apenas uma tradução. Vamos direto à sintaxe e, em seguida, a exemplos.

Sintaxe:

```
Function <nome> (<parâmetros>) : <tipo do valor de retorno>;
const
       <constantes>
var
       <variáveis>

Begin
       <instruções>
       <nome> := <valor de retorno>;
End;
```

Nós já sabemos, mas vamos reforçar: a característica principal, que faz com que um subprograma seja uma função, é o retorno de um valor.

Assim, é necessário na sintaxe existir o <tipo do valor de retorno> e também o próprio retorno de valor, que é feito ao final da *function*, atribuindo o valor que se deseja retornar para o nome da função.

Como exemplo, iremos rever a função que recebe dois valores numéricos, efetua a soma e devolve o resultado. Esta função estará junto a um programa que a chama para obter o resultado de dois valores solicitados ao usuário.

```
1  Program ExemploFunction;
2
3  var
4        n1, n2 : Real;
5
6  Function Soma(valor1, valor2 : Real) : Real;
7  Begin
8        { Faz a soma dos valores e já retorna o resultado
9  atribuindo ao nome da função }
10       Soma := valor1 + valor2;
11 End;
12
13 Begin
14
15       write('Primeiro número: ');
16       readln(n1);
17
18       write('Segundo número: ');
19       readln(n2);
20
21       write(n1, ' + ', n2, ' = ', Soma(n1,n2));
22
23 End.
```

Na linha 21, o **write** exibirá:

a) o valor de n1 (vamos simular 13)
b) o sinal de "+"
c) o valor de n2 (vamos simular 2)
d) o sinal de igual "="
e) o retorno da função Soma, ou seja, o resultado de 13 + 2, em nossa simulação

Isto aparecerá na tela:

```
13 + 2 = 15
```

4 PASSAGEM DE PARÂMETROS POR REFERÊNCIA

Até agora, o tipo de passagem de parâmetros que estudamos é conhecido como "passagem de parâmetros por valor" ou "passagem de parâmetros por cópia". Isto porque, quando passamos uma variável por parâmetro a um subprograma, a variável que está declarada na construção do subprograma recebe apenas o valor (ou uma cópia do valor) da variável enviada.

Exemplo: No programa anterior, quando chamamos **Soma(n1,n2)**, estamos passando apenas uma cópia do valor de **n1** e **n2** para **valor1** e **valor2** respectivamente. Assim, se **n1** tiver 18 e **n2** tiver 4, após a chamada da função **Soma**, teremos isto na memória:

```
n1 = 18            valor1 = 18
n2 = 4             valor2 = 4
```

É desta forma que trabalhamos até agora.

Já na passagem de parâmetros por referência, como o próprio nome já diz, a variável declarada no subprograma recebe não apenas uma cópia do valor mas, sim, uma referência à área de memória onde está declarada a variável que foi enviada para o subprograma.

Vamos representar visualmente:

```
n1 = 18   ← valor1
n2 = 4    ← valor2
```

Assim, sempre que formos utilizar a variável **valor1**, a execução ira buscar o conteúdo onde **valor1** está "apontando", que é a área de memória onde está armazenado o valor de **n1**.

Qual é a diferença básica?

A diferença é que **tudo o que acontecer com a variável referência, dentro do subprograma, acontecerá com a variável que foi enviada**. Utilizando nosso exemplo, **tudo o que acontecer com valor1 acontecerá com n1 e tudo o que acontecer com valor2 acontecerá com n2.**

Como se define que uma variável deve receber a referência e não só o valor? Simplesmente colocando a palavra **var** na declaração da variável, no subprograma.

Vamos a um exemplo:

```
1  Function Soma(var valor1 : Real; valor2 : Real) : Real;
2  Begin
3         { Instruções }
4  End;
```

Veja que somente o primeiro parâmetro será passado por referência, pois somente ele tem a palavra **var** antes. O segundo parâmetro será passado da forma como já estávamos utilizando até o momento, por cópia.

O programador pode passar quais e quantos parâmetros ele quiser por referência.

Vamos ver um exemplo simples para entender melhor o que acontece quando valores são passados por referência ou por cópia.

```
1  Program ExemploPassagemParametroReferencia;
2
3  var
4         n1, n2 : Real;
5
6  Procedure Executa(Param1 : Real; var Param2 : Real);
7  Begin
8         { Adiciona 1 à primeira variável }
9         Param1 := Param1 + 1;
10         {Adiciona 1 à segunda variável }
11         Param2 := Param2 + 1;
12 End;
13
14 Begin
15
16         n1 := 10;
17         n2 := 8;
18
19         Executa(n2, n1);
20
21         Writeln('O valor de n1 é ', n1);
22         Writeln('O valor de n2 é ', n2);
23
24 End.
```

O conteúdo a seguir será exibido após executar o programa:

```
O valor de n1 é 11
O valor de n2 é 8
```

Vamos entender por quê:

n2 é o primeiro parâmetro enviado para o subprograma. Quem recebe **n2** é **Param1,** porém **Param1** recebe apenas uma cópia do valor de **n2.**

n1, o segundo parâmetro, é enviado para o subprograma e quem o recebe é **Param2,** porém **Param2** recebe uma referência à área de memória onde está **n1.**

Vejamos como fica a memória:

```
n1 = 10    ← Param2
n2 = 8       Param1 = 8
```

Quando o subprograma executa a linha 9, a memória ficará assim:

```
n1 = 10    ← Param2
n2 = 8       Param1 = 9
```

Em seguida, o subprograma executará a linha 11, e a memória ficará com este estado:

```
n1 = 11    ← Param2
n2 = 8       Param1 = 9
```

Tudo o que acontece com uma variável, que recebeu uma referência dentro de um subprograma, acontece também com a variável cuja referência foi passada.

Em outras palavras, quando algo acontecer com **Param2**, acontecerá também com **n1**.

Assim, quando a execução do subprograma terminar, as variáveis ali contidas não farão mais parte da execução. Sendo assim, a memória estará assim:

```
n1 = 11
n2 = 8
```

Perceba que o valor de **n1** foi alterado através de **Param2** que estava dentro do subprograma.

Para finalizar, vamos ver um problema mais prático.

Vamos criar um programa que leia um salário e atualize este salário em 5,35 por cento. O programa contará com um subprograma que faz esta atualização.

```
1   Program Pzim ;
2
3   var
4          Salario : Real;
5
6   procedure AtualizaSalario(var valor : Real);
7   begin
8        {Coloca em "valor" o próprio valor, porém atualizado em 5,35%}
9          valor := valor + (valor * 5.35 / 100);
10  end;
11
12  Begin
13
14      Write('Salário: ');
15        Readln(Salario);
16
17        AtualizaSalario(Salario);
18
19        Writeln('O salário foi reajustado para: ', Salario);
20
21  End.
```

A variável **Salario** recebe um valor na linha 15 (vamos supor, mil reais). Na linha 17 é feita a chamada a um subprograma. No subprograma, a variável

valor recebe a referência da variável **Salario**. Quando, na linha 9, o conteúdo da variável **valor** é alterado, a variável **Salario** é alterada também.

A execução volta ao programa principal, para a linha 19. O conteúdo da variável **Salario** – que foi alterado pelo subprograma – é exibido.

RESUMO DO TÓPICO 5

- Os subprogramas do tipo *Function* e *Procedure* correspondem às mesmas estruturas já estudadas nos algoritmos, Funções e Procedimentos.

- Entre suas vantagens, podemos reforçar a modularização de ações, o reaproveitamento de código e a abstração de rotinas.

- Estudamos, neste tópico, um recurso muito interessante que é a passagem de parâmetros por referência. Este recurso permite que o conteúdo de uma variável possa ser alterado por dentro de um subprograma.

AUTOATIVIDADE

1 Desenvolva um algoritmo que leia o valor do salário bruto de uma pessoa e calcule o desconto do imposto de renda segundo os dados a seguir:

Faixa Salarial	Desconto (%)
até 500	Isento
de 500 até 1.500	10%
de 1.500 até 2.500	15%
acima de 2.500	25%

Enquanto o salário não for 0 (zero), o programa deverá continuar solicitando salários para calcular o desconto.

Crie um subprograma chamado **CalculaDesconto** que receba o salário bruto e retorne o salário líquido calculado.

2 Você faz uma aplicação de "A" reais à taxa de juros "J" constante por um período de "M" meses. Qual será o montante "T" após o término da aplicação? Considere o cálculo de juros compostos.

O programa deverá ler os valores A, J e M, enviar estes valores para um subprograma chamado **Calcula**, que calculará o montante e retornará o valor calculado. O programa principal deverá exibir o resultado.

3 Refaça o programa que lê três valores correspondentes aos possíveis lados de um triângulo e exibe se pode ser um triângulo ou não. Se puder, o programa deverá exibir qual é o tipo do triângulo. Se o triângulo for isósceles ou escaleno, o programa deverá verificar se ele também é um triângulo retângulo e exibir esta informação ao usuário.

O programa deverá utilizar os seguintes subprogramas:
Triangulo – Subprograma que recebe três medidas e verifica se as medidas formam um triângulo. O subprograma deve retornar um valor lógico indicando se as medidas formam um triângulo ou não.
Equilatero – Subprograma que recebe três medidas e verifica se as medidas formam um triângulo equilátero. O subprograma deve retornar um valor lógico indicando se as medidas formam um triângulo equilátero ou não.

Isosceles – Subprograma que recebe três medidas e verifica se as medidas formam um triângulo isósceles. O subprograma deve retornar um valor lógico indicando se as medidas formam um triângulo isósceles ou não.
Escaleno – Subprograma que recebe três medidas e verifica se as medidas formam um triângulo escaleno. O subprograma deve retornar um valor lógico indicando se as medidas formam um triângulo escaleno ou não.
Retangulo – Subprograma que recebe três medidas e verifica se as medidas formam um triângulo retângulo. O subprograma deve retornar um valor lógico indicando se as medidas formam um triângulo retângulo ou não.

4 Um clube fará um recadastramento de todos os seus sócios. Cada sócio que comparecer no recadastramento passará a pagar um valor fixo (R$ 300,00) por mês, mais alguns acréscimos, de acordo com o número de dependentes e o tempo que já é cliente.

Faça um programa que solicite, para cara sócio (não se sabe quantos sócios aparecerão) a quantidade de dependentes e o tempo que é cliente (em meses). A cada sócio, o programa deverá exibir a mensalidade a ser paga, a quantidade de dependentes e o tempo de cliente.

As regras para recálculo são:
Número de dependentes
Nenhum dependente – desconto de 5% (sobre o valor fixo)
1 dependente – acréscimo de R$ 5,00
2 ou mais dependentes – acréscimo de R$ 10,00
Tempo de cliente
12 meses ou menos – 1% de desconto
13 meses até 47 meses – 3% de desconto
48 meses ou mais – 8% de desconto
* Este desconto é aplicado sobre o valor da mensalidade já considerando o desconto ou acréscimo correspondente ao número de dependentes.

Para auxiliar o programa principal, crie os seguintes subprogramas:
Continua – subprograma utilizado para verificar se o usuário deseja ler mais dados ou parar a leitura. O subprograma deve perguntar se o usuário deseja continuar, deverá ler a decisão do usuário e retornar um valor indicando se o usuário deseja continuar ou não. O programa principal deverá utilizar-se dessa resposta (desse valor) para continuar a solicitar dados ou não.
Dependentes – subprograma que recebe um valor referente ao número de dependentes e o valor da mensalidade. O subprograma deverá alterar o valor da mensalidade de acordo com a regra.

Tempo – subprograma que recebe um valor referente ao tempo de cliente e o valor da mensalidade (já calculado com o número de dependentes). O subprograma deverá alterar a mensalidade de acordo com a regra.
Exibe – este subprograma recebe o valor da mensalidade calculado, o número de dependentes e o tempo que o usuário é cliente e exibe estes valores, semelhante a:

Valor a pagar: 285,2
Dependentes: 2
Tempo: 50 meses

5 Faça um programa que leia estas informações de vários funcionários:

Salário (utilizar subprograma LerSalario)
Tempo de serviço (utilizar subprograma LerTempo)
Regra 1 - Se o salário for menor que R$ 700,00, o programa deverá solicitar a quantidade de dependentes que o funcionário possui e deverá adicionar, ao salário, R$ 9,50 para cada dependente.
Regra 2 - O programa também deverá verificar o tempo de serviço. Se esse tempo for maior que 1 ano e menor que três anos, o salário deverá ser acrescido de 0,5%. Se o salário estiver entre 3 anos e menor que 5 anos, deverá ser acrescido de 1,5%. Se for maior que 5 anos, deverá ser acrescido de 3%.

Funcionamento:
O programa principal deverá chamar um subprograma para ler o salário e um subprograma para ler o tempo de serviço (ver especificação destes subprogramas mais abaixo).
Para saber o valor a ser calculado sobre o número de dependentes, o programa principal deverá chamar o subprograma **ValorDependentes**.
Para calcular o acréscimo baseado no tempo de serviço, o programa chamará o subprograma **AcrescimoTempo**.
A cada funcionário, o programa deverá exibir o salário acrescido do valor por dependentes, mais o acréscimo por tempo de serviço.

Definição dos subprogramas:
LerSalario – Subprograma que recebe uma variável para armazenar um salário, solicita o valor de um salário, lê um valor e armazena o valor lido na variável recebida.
LerTempo – Subprograma que recebe uma variável para armazenar um valor inteiro, solicita o valor de tempo em meses, lê um valor e armazena o valor lido na variável recebida.

ValorDependentes – Subprograma que recebe o valor correspondente ao salário de um funcionário e verifica se o valor é menor que R$ 700,00. Se for, o subprograma deverá solicitar a quantidade de dependentes. O subprograma deverá calcular R$ 9,50 para cada dependente e devolver esse valor calculado (Regra 1).

AcrescimoTempo – Subprograma que recebe um valor correspondente ao um tempo de serviço e a um salário, verifica o mesmo e retorna o valor de acréscimo conforme Regra 2.

TÓPICO 6

CONCEITOS AVANÇADOS

1 INTRODUÇÃO

Todos os conceitos que vimos até agora, na linguagem de programação Pascal, já havíamos estudado nos algoritmos, salvo algumas sutis diferenças.

Veremos, agora, algumas características mais avançadas que a linguagem proporciona. Você poderá encontrar na literatura alguns destes conceitos também em algoritmos, porém deixamos para estudá-los somente neste momento, pois agora já temos um conhecimento adequado que auxiliará a entender melhor estas estruturas.

2 TIPOS DE DADOS CRIADOS PELO PROGRAMADOR

Até agora, estudamos apenas os tipos de dados padrões da linguagem. No entanto, é possível criar tipos de dados personalizados.

Vamos rever o exemplo de soma de matrizes. No problema, tínhamos três variáveis *arrays* bidimensionais, que armazenam o mesmo tipo de dados e têm a mesma dimensão.

```
1  Program ExemploVetor;
2
3  const
4         QtdLin = 3;
5         QtdCol = 2;
6
7  var
8         Mat1 : Array [1..QtdLin, 1..QtdCol] of Real;
9         Mat2 : Array [1..QtdLin, 1..QtdCol] of Real;
10        MatSoma : Array [1..QtdLin, 1..QtdCol] of Real;
11        L, C : Integer;
```

Note que as variáveis **Mat1**, **Mat2** e **MatSoma** têm a mesma declaração, ou seja, são exatamente do mesmo tipo de dados.

Este é um dos casos para os quais podemos criar um tipo de dado que corresponda a essa declaração. Depois, basta declarar as três variáveis com este novo tipo.

Assim como existe uma área de variáveis (delimitada com a palavra **var**) e uma área de constantes (delimitada pela palavra **const**), podemos criar uma área de tipo de dados. Esta área será delimitada pela palavra **type**.

A sintaxe para se declarar um tipo de dados é:

```
<nome do tipo> = <tipo>;
```

<nome do tipo> é o identificador do tipo, ou seja, é você, o programador, que criará, assim como criamos variáveis, constantes etc.

<tipo> é propriamente o tipo de dado a ser declarado.

Vamos declarar um tipo chamado **Matriz**. Este tipo terá o mesmo formato das matrizes que precisamos declarar para o problema do nosso exemplo.

```
 1  Program ExemploDeclaracaoTipo;
 2
 3  const
 4         QtdLin = 3;
 5         QtdCol = 2;
 6
 7  type
 8         Matriz = Array [1..QtdLin, 1..QtdCol] of Real;
 9
10  var
11         Mat1, Mat2, MatSoma : Matriz;
12         L, C : Integer;
 .         .
 .         .
 .         .
```

A declaração dos tipos encontra-se depois da área de constantes, pois assim podem-se utilizar as constantes que já estão declaradas para, neste caso, definir a dimensão do *array*.

A área de variáveis vem depois, pois os tipos são para, justamente, declarar variáveis com tais formatos definidos por eles (os tipos).

O funcionamento de todo o resto do programa permanece exatamente igual. Neste caso, a criação de um tipo de dado foi para simplificar a escrita.

Quando você trabalhar com matrizes e existir mais de uma matriz com o mesmo tipo e dimensão, crie um tipo de dado com estas características.

3 REGISTROS

Imagine a necessidade de manter armazenadas as seguintes informações, para várias pessoas:

- Nome
- Idade
- RG
- Peso

Qual seria a solução para isto? Se fosse apenas uma informação por pessoa (um nome por pessoa, por exemplo), poderíamos guardar em um único *array*, concorda? Um *array* do tipo *string*, onde você armazena vários nomes, cada um em um índice.

Porém, precisamos armazenar quatro informações para cada pessoa. Assim, poderíamos utilizar quatro *arrays*, um para cada informação. Vamos representar visualmente como ficaria esta estrutura:

Array **Nome**

Índices →	1	2	3
Dados →	Edgar Scandurra	Jaime Caetano	Muriel Sagas

Array **Idade**

Índices →	1	2	3
Dados →	51	32	29

Array **RG**

Índices →	1	2	3
Dados →	12345678	98765432	14679528

Array **Peso**

Índices →	1	2	3
Dados →	75.5	82	78.7

Com esta estrutura, se pegarmos, por exemplo, os dados do índice 1 de cada um dos *arrays*, obteremos todos os dados de uma única pessoa.

Array no índice 1	**Informação armazenada**
Nome[1]	Edgar Scandurra
Idade[1]	51
RG[1]	12345678
Peso[1]	75.5

Vejamos, agora, como ficaria um programa que armazena estas informações, utilizando quatro *arrays*.

```
1   Program ExemploQuatroArrays ;
2
3   const
4          Qtd = 5;
5
6   type
7          ArrayString = Array [1..Qtd] of String;
8
9   var
10         Nome : ArrayString;
11         Idade : Array [1..Qtd] of Integer;
12         RG : ArrayString;
13         Peso : Array [1..Qtd] of Real;
14         I : Integer;
15
16  Begin
17
18         { Lendo as informações para "x" pessoas }
19         For I := 1 to Qtd do
20         Begin
21                Write('Nome: ');
22                Readln(Nome[I]);
23
24                Write('Idade: ');
25                Readln(Idade[I]);
26
27                Write('RG: ');
28                Readln(RG[I]);
```

```
29
30              Write('Peso: ');
31              Readln(Peso[I]);
32
33              writeln;
34       End;
35       clrscr;
36       { Exibindo as informações de todas as pessoas armazenadas }
37 For I := 1 to Qtd do
38       Begin
39              Writeln('Nome: ', Nome[I]);
40
41              Writeln('Idade: ', Idade[I]);
42
43              Writeln('RG: ', RG[I]);
44
45              Writeln('Peso: ', Peso[I]);
46
47              writeln;
48       End;
49
50 End.
```

Antes de prosseguir, vamos analisar um pouco o programa a fim de entendê-lo bem para termos bastante base para compreender o próximo conceito.

Primeiramente, como possuímos mais de uma variável do tipo *array* de *strings*, criamos um tipo para esta definição.

Poderíamos criar um tipo de dado para cada *array*, mas como praticamente cada variável *array* possui um tipo diferente, podemos declarar a variável diretamente com o tipo completo (linhas 11 e 13).

Você pode se perguntar por que a variável RG é do tipo *string*, se ela armazena apenas números. Esta é uma situação bastante interessante a ser analisada. Nós não utilizamos normalmente os números de um RG para cálculos. Não somamos o RG a nenhum valor, por exemplo. Portanto, este número, para nós, é apenas uma sequência de caracteres, por isso o tipo *string*. Além disso, se desejássemos armazenar os caracteres "ponto" do RG, o tipo continuaria igual, ou seja, poderíamos escolher armazenar "12345678" ou "12.345.678".

Vamos à leitura de dados através da estrutura **For-Do** que começa na linha 19 e termina na 34.

Para a linguagem, cada variável com os dados das pessoas é independente uma da outra, não há nenhum vínculo entre elas. Realmente não há.

Mas nós estamos criando um vínculo "lógico". Veja que todos os dados, de uma única pessoa, estão em um mesmo índice, mas em variáveis diferentes. Por exemplo, quando o valor da variável "`I`" for 3, o programa vai pedir o nome

de uma pessoa (linha 21). O usuário vai informar este dado (um nome qualquer) o qual será armazenado no índice 3 da variável Nome (linha 22). Em seguida, o programa solicitará a idade. O dado referente à idade desta pessoa também será armazenado no índice 3 (porém, na variável **Idade**), pois, das linhas 21 e 22 até as linhas 24 e 25, o valor da variável "`I`" não sofreu alteração. É dessa forma que criamos este vínculo.

Entendido o problema e a situação, vamos seguir adiante.

Existe um recurso, no qual podemos definir um registro de dados. Cada registro pode conter quantos dados forem necessários (em nosso exemplo: nome, idade, RG, peso).

Se definirmos um *array* cujo tipo seja este registro, cada posição do *array*, ao invés de armazenar uma única informação (que é o conceito que possuíamos até o momento), poderemos armazenar todas as informações definidas em um registro.

Vamos ver um exemplo gráfico de como ficaria esta estrutura:

Índices →	**1**	**2**	**3**
Registros → (Um em cada índice)	Nome := 'Edgar Scandurra' Idade := 51 RG := 12345678 Peso := 75.5	Nome := 'Jaime Caetano' Idade := 32 RG := 98765432 Peso := 82	Nome := 'Muriel Sagas' Idade := 29 RG := 14679528 Peso := 78.7

Representação de um *array* de registros

É necessária uma única variável para armazenar os dados e a estrutura fica mais simples.

Como e onde se declara um registro?

Um registro não deixa de ser um tipo de dados, portanto é declarado na área **Type**.

A sintaxe é a seguinte:

```
<nome do registro> = Record
       <campo_1> : <tipo>;
       <campo_2> : <tipo>;
       <campo_N> : <tipo>;
End;
```

Define-se o nome do registro (identificador), seguido de um sinal de igual e da palavra *"Record"*, que significa "Registro".

Entre o início (palavra *Record*) e o fim (*End;*) da estrutura, declaram-se todos os dados que se desejam armazenar no registro. Veja como ficaria o registro contendo os quatro dados das pessoas utilizados no exemplo anterior:

```
RegistroPessoa = Record
       Nome : String;
       Idade : Integer;
       RG : String;
       Peso : Real;
End;
```

Colocamos o nome do registro de **RegistroPessoa**. A letra "T" no início indica que aquele identificador é um tipo. Isto não é necessário, porém alguns programadores usam essa forma de escrita.

É comum chamar os dados de um registro de "campos". Por exemplo, o registro RegistroPessoa contém o campo "Nome".

Para declarar uma variável com este tipo, a sintaxe é padrão, conforme já vimos anteriormente. Exemplo:

```
Type
       RegistroPessoa = Record
              Nome : String;
              Idade : Integer;
              RG : String;
              Peso : Real;
       End;

var
       Pessoa : RegistroPessoa;
```

Para acessar cada campo da variável pessoa, utiliza-se tal sintaxe:

```
<variável>.<campo>
```

Desta forma:

```
Pessoa.Nome
Pessoa.RG
```

Vamos ver um programa simples que utiliza uma variável declarada com o tipo "Registro" que acabamos de ver.

```
 1  Program ExemploRegistro;
 2
 3  Type
 4         RegistroPessoa = Record
 5                Nome : String;
 6                Idade : Integer;
 7                RG : String;
 8                Peso : Real;
 9         End;
10
11  var
12         Pessoa : RegistroPessoa;
13
14  Begin
15
16         Write('Nome: ');
17         Readln(Pessoa.Nome);
18
19         Write('Idade: ');
20         Readln(Pessoa.Idade);
21
22         Write('RG: ');
23         Readln(Pessoa.RG);
24
25         Write('Peso: ');
26         Readln(Pessoa.Peso);
27
28         clrscr;
29
30         Writeln('Informações da pessoa: ');
31         Writeln('Nome: ', Pessoa.Nome);
32         Writeln('Idade: ', Pessoa.Idade);
33         Writeln('RG: ', Pessoa.RG);
34         Writeln('Peso: ', Pessoa.Peso);
35
36  End.
```

Se, em uma variável, conseguimos armazenar quatro valores, se declararmos um *array* de, por exemplo, 5 posições e cada posição for do tipo **RegistroPessoa**, em cada posição armazenaremos quatro dados, conforme já explicado anteriormente.

É isto que faremos no programa a seguir. A funcionalidade será a mesma do programa ExemploQuatroArrays.

```
1   Program ExemploArrayDeRegistros ;
2
3   const
4          Qtd = 5;
5
6   Type
7          RegistroPessoa = Record
8                 Nome : String;
9                 Idade : Integer;
10                RG : String;
11                Peso : Real;
12         End;
13
14  var
15         Pessoa : Array [1..Qtd] of RegistroPessoa;
16         I : Integer;
17  Begin
18
19         { Lendo as informações para "x" pessoas }
20         For I := 1 to Qtd do
21         Begin
22                Write('Nome: ');
23                Readln(Pessoa[I].Nome);
24
25                Write('Idade: ');
26                Readln(Pessoa[I].Idade);
27
28                Write('RG: ');
29                Readln(Pessoa[I].RG);
30
31                Write('Peso: ');
32                Readln(Pessoa[I].Peso);
33
34                writeln;
35         End;
36
37         clrscr;
38
39         { Exibindo as informações de todas as pessoas armazenadas }
40  For I := 1 to Qtd do
41         Begin
42                Writeln(I,'a. pessoa:');
43                Writeln('Nome: ', Pessoa[I].Nome);
44                Writeln('Idade: ', Pessoa[I].Idade);
45                Writeln('RG: ', Pessoa[I].RG);
46                Writeln('Peso: ', Pessoa[I].Peso);
47
48                writeln;
49         End;
50
51  End.
```

Podemos notar que, quando vamos trabalhar com um campo da variável **Pessoa**, esta variável sempre vem acompanhada por um índice (para o qual, no programa, é utilizada a variável "I"). Isto se deve ao fato que a variável é um *array*. Assim, o conceito é exatamente o mesmo dos *arrays* que já estudamos anteriormente. A única diferença é que, além de acessar um determinado índice do *array* (por exemplo, **Pessoa[I]** – lê-se: "pessoa no índice "I"), precisa-se especificar qual é a informação que desejamos, armazenada em tal índice. Isto, como já vimos, faz-se adicionando o campo, que queremos depois da variável, separado por um ponto (`Pessoa[I].Nome`, por exemplo).

Isto é visivelmente percebido através do quadro representação de um *array* de registros.

Outro conceito interessante é que, além de declararmos uma variável como um *array* de registros, como fizemos na linha 15 do programa anterior, podemos declarar mais um tipo que, ele próprio, já é um *array* de registros. Segue exemplo:

```
1   Program ExemploArrayDeRegistros ;
2
3   const
4          Qtd = 5;
5
6   Type
7          RegistroPessoa = Record
8                 Nome : String;
9                 Idade : Integer;
10                RG : String;
11                Peso : Real;
12         End;
13
14  ArrayRegistroPessoa = Array [1..Qtd] of RegistroPessoa;
15
16  var
17         TodasPessoas : ArrayRegistroPessoa;
18                Diretor, Gerente : RegistroPessoa;
```

Entendendo estas declarações:

Temos um tipo chamado **RegistroPessoa**. A variável que for declarada com este tipo poderá armazenar, ao mesmo tempo, quatro dados (nome, idade, RG e peso) de uma única pessoa.

Declaramos, também, um tipo chamado **ArrayRegistroPessoa**, que é formado por um *array* do tipo **RegistroPessoa**. A dimensão deste *array* é definida através da constante **Qtd**.

Isto quer dizer que uma variável, que for declarada com este tipo, poderá armazenar até 5 (valor de **Qtd**) registros, e cada registro poderá armazenar os mesmos quatro dados. Resumindo, poderemos armazenar os dados de 5 pessoas.

Assim, a variável **TodasPessoas** poderá armazenar os 4 dados para até 5 pessoas. Já as variáveis **Diretor** e **Gerente** poderão armazenar, cada uma, os 4 dados para uma única pessoa.

É com esta estrutura que faremos um próximo programa. O programa armazenará o nome, a idade e o salário de várias pessoas e, ao final, informará quem é a pessoa mais nova e quem é a pessoa que recebe o maior salário.

```
1   Program ExemploArrayDeRegistros ;
2
3   const
4          Qtd = 10;
5
6   Type
7          RegistroPessoa = Record
8                 Nome : String;
9                 Idade : Integer;
10                Salario : Real;
11         End;
12
13         ArrayRegistroPessoa = Array [1..Qtd] of RegistroPessoa;
14
15  var
16  TodasPessoas : ArrayRegistroPessoa;
17         PessoaComMaiorSalario, PessoaMaisNova : RegistroPessoa;
18         I : Integer;
19
20  Begin
21
22         { Solicitando e armazenando as informações }
23         For I := 1 to Qtd do
24         Begin
25                Write('Nome: ');
26                Readln(TodasPessoas[I].Nome);
27
28                Write('Idade: ');
29                Readln(TodasPessoas[I].Idade);
30
31                Write('Salário: ');
32                Readln(TodasPessoas[I].Salario);
33
34                writeln;
35         End;
36
37         { Coloca a primeira pessoa armazenada, como sendo a que tem
38  maior salário até o momento }
39         PessoaComMaiorSalario := TodasPessoas[1];
40         { Dados do array TodasPessoas no primeiro índice }
41
42       {Coloca a primeira pessoa armazenada, como sendo a mais nova}
```

```
43          PessoaMaisNova := TodasPessoas[1];
44          { Dados do array TodasPessoas no primeiro índice }
45
46          { Percorre as informações armazenadas }
47          For I := 1 to Qtd do
48          Begin
49              {Verifica se o salário que está passando agora é maior do que
50              o maior salário armazenado até o momento}
51                 If (TodasPessoas[I].Salario >
52              PessoaComMaiorSalario.Salario) Then
53                 Begin
54                        { Se o salário deste índice for maior,
55                    armazena-o como sendo o maior até o momento }
56                        PessoaComMaiorSalario := TodasPessoas[I];
57                 End;
58
59          { Verifica se a pessoa que está passando agora é mais nova
60     do que a mais nova até o momento }
61          If (TodasPessoas[I].Idade < PessoaMaisNova.Idade) Then
62           Begin
63                        { Se a pessoa deste índice for a maios nova,
64           armazena-a como sendo a mais nova até o momento }
65                        PessoaMaisNova := TodasPessoas[I];
66                 End;
67          End;
68
69    { Exibe as informações da pessoa que foi armazenada com o maior
70   salário }
71          Writeln('Pessoa que possui o maior salário: ', Pessoa-
72   ComMaiorSalario.Nome);
73          Writeln('Esta pessoa recebe: ', PessoaComMaiorSalarioSalario);
74
75   writeln;
76
77          {Exibe as informações da pessoa mais nova }
78          Writeln('A pessoa mais nova é o(a) ',PessoaMaisNova Nome,',
79          com ',PessoaMaisNova.Idade,' anos.');
80
81   End.
```

Criamos um registro que pode armazenar:

- um nome;
- uma idade;
- um valor que corresponde a um salário.

Criamos, também, um tipo (**ArrayRegistroPessoa**) que pode armazenar, no exemplo, até 10 pessoas. Para cada pessoa, poderão ser armazenados os três dados citados.

Declaramos a variável **TodasPessoas** com o tipo **ArrayRegistroPessoa**. Isto quer dizer que esta variável poderá armazenar a quantidade de pessoas determinada pela constante **Qtd** (e reforçando: para cada pessoa será possível armazenar Nome, Idade e Salário).

Declaramos duas variáveis (**PessoaComMaiorSalario**, **PessoaMaisNova**) com o mesmo tipo (**RegistroPessoa**). Isto significa que cada uma das variáveis pode armazenar os três dados (nome, idade e salário), porém de uma única pessoa para cada variável. Conseguiremos, então, armazenar os dados de duas pessoas específicas, uma em cada variável. Nada impede que aconteça de serem armazenados os dados de uma mesma pessoa em cada uma das variáveis. Se isto acontecer, os dados serão repetidos (um nome, idade e salário na variável **PessoaComMaiorSalario**, e o mesmo nome, idade e salário na variável **PessoaMaisNova**).

Porém, se esta for a situação, não há problema nenhum.

Das linhas 22 a 35, estamos solicitando informações de 10 pessoas e armazenando tudo na variável **TodasPessoas**.

Na linha 39, estamos inicializando a variável **PessoaComMaiorSalario** com os dados da primeira pessoa armazenada no *array*. Esta é uma pequena técnica de inicialização. Como o objetivo é comparar o conteúdo (neste caso, o salário) da variável **PessoaComMaiorSalario** com cada um dos salários armazenados em **TodasPessoas** e, para isto, a variável **PessoaComMaiorSalario** precisa ser inicializada, inicializa-se a variável, então, com os dados da primeira pessoa armazenada e, depois, segue comparando com as demais (isto é feito na linha 51). Se encontrar um salário maior, passará a armazenar os dados desta nova pessoa em **PessoaComMaiorSalario**; caso contrário, os dados que já estão na variável, continuarão lá.

A mesma coisa acontece com a variável **PessoaMaisNova**. Inicializa-se com os dados da primeira pessoa armazenada no *array* e, depois, segue comparando com as demais. Ao encontrar uma pessoa com idade menor do que a armazenada em **PessoaMaisNova**, passa a armazenar esta pessoa.

A diferença principal está no fato que agora, quando armazenamos os dados de uma pessoa, estamos armazenando todos os dados desta pessoa, de uma só vez.

Antes, precisávamos armazenar individualmente, em variáveis independentes, cada um dos dados.

LEITURA COMPLEMENTAR

MOTIVOS PARA ESTUDAR OS CONCEITOS DE LINGUAGENS DE PROGRAMAÇÃO

É natural que os estudantes imaginem como se beneficiarão do estudo dos conceitos de linguagens de programação. Afinal de contas, uma grande quantidade de outros tópicos da ciência da computação merece um estudo sério. O que apresentamos a seguir é o que acreditamos ser uma lista obrigatória dos benefícios potenciais relativos ao estudo de conceitos de linguagens.

- Aumento da capacidade de expressar ideias. Acredita-se que a profundidade de nossa capacidade intelectual seja influenciada pelo poder expressivo da linguagem em que comunicamos nossos pensamentos. Os que possuem uma compreensão limitada da linguagem natural são limitados na complexidade de expressar seus pensamentos, especialmente em termos de profundidade de abstração. Em outras palavras, é difícil para as pessoas conceberem estruturas que não podem descrever, verbalmente ou por escrito. Programadores inscritos no processo de desenvolver *softwares* veem-se similarmente embaraçados. A linguagem na qual desenvolvem o *software* impõe limites quanto aos tipos de estruturas de controle, de estruturas de dados e de abstrações que eles podem usar; assim, as formas de algoritmos possíveis de serem construídas também são limitadas. O conhecimento de uma variedade mais ampla de recursos de linguagens de programação reduz essas limitações no desenvolvimento de *software*. Os programadores podem aumentar a variedade de seus processos intelectuais de desenvolvimento de *software* aprendendo novas construções de linguagem. Pode-se argumentar que aprender as capacidades de outras linguagens não ajudará um programador obrigado a usar uma linguagem sem essas capacidades. Esse argumento não se sustenta, porém, porque frequentemente as facilidades da linguagem podem ser simuladas em outras linguagens que não suportam esses recursos diretamente.

Por exemplo, depois de ter aprendido as funções de manipulação de matrizes do FORTRAN 90 (ANSI, 1992), um programador C++ (STROUSTRUP, 1997) seria levado naturalmente a construir subprogramas para oferecer essas operações. O mesmo é verdadeiro em relação a muitas outras construções complexas [...]. O estudo dos conceitos das linguagens de programação forma uma apreciação dos recursos valiosos da linguagem e encoraja os programadores a usá-los. O fato de muitos recursos das várias linguagens poderem ser simulados em outras não diminui significativamente a importância de projetar linguagens com o melhor conjunto de recursos. Sempre é melhor usar um recurso cujo projeto foi integrado na linguagem do que usar uma simulação desse, o qual, muitas vezes, é menos elegante e mais desajeitado em uma linguagem que não o suporta.

- Maior embasamento para a escolha de linguagens apropriadas. Muitos programadores profissionais tiveram pouca educação formal em ciência da computação; ao contrário, aprenderam a programar sozinhos ou por meio de

programas de treinamento dentro da própria organização. Tais programas frequentemente ensinam uma ou duas linguagens diretamente pertinentes ao trabalho da organização. Muitos outros programadores receberam seu treinamento formal em um passado distante. As linguagens que aprenderam não são mais usadas, e muitos recursos agora disponíveis não eram amplamente conhecidos. O resultado disso é que muitos programadores, quando lhes é dada a possibilidade de escolha das linguagens para um novo projeto, continuam a usar aquela cama com a qual estão mais familiarizados, mesmo que ela seja pouco adequada ao novo projeto. Se tais programadores estivessem mais familiarizados com as outras linguagens disponíveis, especialmente com os seus recursos particulares, estariam em uma posição melhor para fazerem uma escolha consciente.

- Capacidade aumentada para aprender novas linguagens. A programação de computadores é uma disciplina jovem, e as metodologias de projeto, as ferramentas de desenvolvimento de *software* e as linguagens de programação ainda estão em um estágio de contínua evolução. Isso torna o desenvolvimento de *software* uma profissão excitante, mas também significa que a aprendizagem contínua é fundamental. O processo de aprender uma nova linguagem de programação pode ser extenso e difícil, especialmente para alguém que esteja à vontade com somente uma ou com duas linguagens e que jamais examinou os conceitos em geral. Assim que for adquirida uma completa compreensão dos conceitos fundamentais das linguagens, será mais fácil ver como esses estão incorporados ao projeto da linguagem aprendida.

Por exemplo, programadores que entendem o conceito de abstração de dados terão mais facilidade para aprender como construir tipos de dados abstratos em Java (GOSLING et al., 1996) do que aqueles que não estão absolutamente familiarizados com tal exigência. O mesmo fenômeno ocorre nas linguagens naturais. Quanto mais você conhece a gramática de sua língua nativa, mais fácil achará aprender uma segunda língua natural. Além disso, aprender uma segunda língua também tem o efeito colateral benéfico de ensiná-lo mais a respeito de seu primeiro idioma.

Por fim, é essencial que os programadores ativos conheçam o vocabulário e os conceitos fundamentais das linguagens de programação, para que possam ler e entender seus manuais e sua literatura de vendas de linguagens e de compiladores.

FONTE: SEBESTA, Robert W. **Conceitos de linguagens de programação**. 5. ed. Porto Alegre: Bookman, 2002. p. 16-17.

RESUMO DO TÓPICO 6

- Vimos neste último tópico que, além dos tipos de dados padrões da linguagem, o programador pode criar seus próprios tipos. Isto facilita quando há, por exemplo, mais de uma variável de um tipo mais complexo.

- Vimos, também, o recurso de registro. Este recurso permite criar variáveis nas quais se podem armazenar diferentes tipos de dados organizados em estruturas. Podem-se, ainda, criar vetores com estas estruturas, permitindo que, em uma única variável, possam-se armazenar várias estruturas com tipos de dados diferentes.

AUTOATIVIDADE

1 Cite uma vantagem de utilizar tipos de dados criados pelo programador. Dê um exemplo.

2 Faça um programa que armazene os dados de, no máximo, 50 pessoas. Tais dados são: Nome, Salário e Idade. Estes dados devem ser armazenados em uma estrutura de registro.

Após todos os dados armazenados, o programa deverá exibir:
a) o nome da pessoa mais nova
b) o nome da pessoa que possui o salário mais alto, porém somente das pessoas acima de 30 anos
c) o salário da pessoa mais velha

3 Crie um programa que possa armazenar, em um registro, o nome e três notas de um aluno. Crie um vetor que possa armazenar estas informações para até 30 alunos.

O programa deverá solicitar as informações dos alunos e, ao final, exibir, para cada aluno: o nome, a média e uma mensagem indicando se está aprovado ou não. A nota para aprovação é 6,5.

4 Desenvolva um programa que contenha um registro para armazenar as seguintes informações: CPF, Nome, Estado civil (1-Solteiro, 2-Casado, 3-Desquitado, 4-Viúvo) e Endereço. O campo Endereço é formado por um outro tipo de dados de Registro, cujos dados são: Rua, Número, Bairro e CEP.

O programa deverá ler e armazenar todos os dados para, no mínimo, 10 pessoas. Ao final, o programa deverá listar todas as pessoas e seus dados, no seguinte formato:

```
João Luiz Correa, CPF: 012.345.567-89, Casado, residente em: Rua
Antônio Blar, 135, bairro Centro, CEP 12654-321

Rogério Mello, CPF: 987.654.321-01, Solteiro, residente em: Rua
São Joaquim, 957, bairro Itoupava, CEP 89765-752
```

REFERÊNCIAS

BLAISE PASCAL. Disponível em: <http://pt.wikipedia.org/wiki/Blaise_Pascal>. Acesso em: 19 jul. 2012.

CURADO, Luiz Reginaldo A. F. PascalZIM. Disponível em: <http://pascalzim.tripod.com/>. Acesso em: 19 jul. 2012.

HOUAISS, Antônio. **O que é língua**. São Paulo: Brasiliense, 1991.

INDENTAÇÃO. Disponível em: <http://pt.wikipedia.org/wiki/Indenta%C3%A7%C3%A3o>. Acesso em: 28 jul. 2012.

KOCHANSKI, Djone; ANDRIETTI, Odilon José. **Algoritmos**. Indaial: Uniasselvi, 2005.

LA PASCALINE. Disponível em: <http://pt.wikipedia.org/wiki/La_pascaline>. Acesso em: 19 jul. 2012.

MONTEIRO, Rhycardo. Disponível em: <http://www2.unemat.br/rhycardo>. Acesso em: 2 set. 2012.

NÚMERO PRIMO. Disponível em: <http://pt.wikipedia.org/wiki/N%C3%BAmero_primo>. Acesso em: 16 jul. 2012.

POPKIN, Richard Henry; STROLL, Avrum. **Philosophy made simple**. [S.l.]: Random House Digital Inc., 1993.

SEPARADOR DECIMAL. Disponível em: <http://pt.wikipedia.org/wiki/Separador_decimal>. Acesso em: 25 jul. 2012.

SANTOS, M. F. dos. **Dicionário de filosofia e ciências culturais**. 3. ed. São Paulo: Matese, 1965.

VALDAMERI, Alexandre R. **Construção de algoritmos**. Indaial: Uniasselvi, 2003.