Fundação
CECIERJ
Consórcio cederj
Centro de Educação Superior a Distância do Estado do Rio de Janeiro

Computação I

Volume Único

Tiago Araújo Neves

GOVERNO DO
Rio de Janeiro

SECRETARIA DE CIÊNCIA,
TECNOLOGIA, INOVAÇÃO E
DESENVOLVIMENTO SOCIAL

UNIVERSIDADE
ABERTA DO BRASIL

MINISTÉRIO DA
EDUCAÇÃO

BRASIL
GOVERNO FEDERAL

Apoio:

FAPERJ
Fundação Carlos Chagas Filho de Amparo
à Pesquisa do Estado do Rio de Janeiro

Fundação Cecierj / Consórcio Cederj

www.cederj.edu.br

Presidente
Carlos Eduardo Bielschowsky

Vice-presidente
Marilvia Dansa de Alencar

Coordenação do Curso de Engenharia de Produção
CEFET – Diego Carvalho
UFF – Cecília Toledo Hernández

Material Didático

Elaboração de Conteúdo
Tiago Araújo Neves

Direção de Design Instrucional
Cristine Costa Barreto

Coordenação de Design Instrucional
Bruno José Peixoto
Flávia Busnardo da Cunha
Paulo Vasques de Miranda

Supervisão de Design Instrucional
Renatta Vittoretti

Design Instrucional
Bruna Canabrava
Cíntia Barreto
Renata Vittoretti

Coordenação de Produção
Fábio Rapello Alencar

Assistente de Produção
Bianca Giacomelli

Revisão Linguística e Tipográfica
Anna Maria Osborne
Beatriz Fontes
Flávia Saboya
José Meyohas
Licia Matos
Maria Elisa Silveira
Mariana Caser
Yana Gonzaga

Ilustração
Clara Gomes
Fernando Romeiro
Renan Alves
Vinicius Mitchell

Capa
Renan Alves

Programação Visual
Alexandre d'Oliveira
Camille Moraes
Cristina Portella
Deborah Curci
Filipe Dutra
Larissa Averbug
Maria Fernanda de Novaes
Mario Lima
Núbia Roma

Produção Gráfica
Patrícia Esteves
Ulisses Schnaider

N518

Neves, Tiago Araújo.
Computação I: volume único / Tiago Araújo Neves. – Rio de Janeiro: Fundação CECIERJ, 2015.
224 p.; il.19 x 26,5 cm.

ISBN: 978-85-458-0025-5

1. Título. I. Computação. II. Informática. III. Programação.

CDD: 004

Referências bibliográficas e catalogação na fonte, de acordo com as normas da ABNT.
Texto revisado segundo o novo Acordo Ortográfico da Língua Portuguesa.

Governo do Estado do Rio de Janeiro

Governador
Luiz Fernando de Souza Pezão

Secretário de Estado de Ciência, Tecnologia, Inovação e Desenvolvimento Social
Gabriell Carvalho Neves Franco dos Santos

Instituições Consorciadas

CEFET/RJ - Centro Federal de Educação Tecnológica Celso Suckow da Fonseca
Diretor-geral: Carlos Henrique Figueiredo Alves

FAETEC - Fundação de Apoio à Escola Técnica
Presidente: Alexandre Sérgio Alves Vieira

IFF - Instituto Federal de Educação, Ciência e Tecnologia Fluminense
Reitor: Jefferson Manhães de Azevedo

UENF - Universidade Estadual do Norte Fluminense Darcy Ribeiro
Reitor: Luis César Passoni

UERJ - Universidade do Estado do Rio de Janeiro
Reitor: Ruy Garcia Marques

UFF - Universidade Federal Fluminense
Reitor: Sidney Luiz de Matos Mello

UFRJ - Universidade Federal do Rio de Janeiro
Reitor: Roberto Leher

UFRRJ - Universidade Federal Rural do Rio de Janeiro
Reitora: Ricardo Luiz Louro Berbara

UNIRIO - Universidade Federal do Estado do Rio de Janeiro
Reitor: Luiz Pedro San Gil Jutuca

Sumário

Aula 1

Conceitos básicos de informática e programação

Tiago Araújo Neves

Meta

Expor um breve histórico da informática e da computação e os conceitos de programação e algoritmo.

Objetivos

Esperamos que, ao final desta aula, você seja capaz de:

1. diferenciar os conceitos-chave relacionados à computação;
2. identificar as características dos algoritmos;
3. construir um algoritmo.

Introdução

Com a disseminação do uso de computadores em praticamente todos os ambientes, tanto de trabalho como de lazer, nos dias de hoje, é vital o conhecimento sobre informática. A informática assumiu grande importância em ambientes profissionais por possibilitar a automação de múltiplas tarefas, fazendo com que as empresas ganhassem agilidade e competitividade. Nos ambientes de lazer, ela está presente nos jogos e em diversos aparelhos eletrônicos, que inclusive possuem acesso à internet atualmente.

Gregor Novak

Figura 1.1: Atualmente a educação também faz amplo uso da informática. Por exemplo, você mesmo pode estar lendo esta aula no monitor de um computador.

Fonte: http://www.sxc.hu/photo/509044

Apesar de o desenvolvimento da informática ter acontecido de forma acentuada a partir da Segunda Guerra Mundial, fato este que a torna uma ciência muito jovem, a computação assumiu tal importância na sociedade moderna que, hoje em dia, fica praticamente impossível pensar nesta mesma sociedade sem o uso de computadores em suas múltiplas formas, como *laptops*, celulares, *tablets*, equipamentos médicos de precisão etc.

Veja as diferenças entre *notebooks*, *netbooks*, *smartphones*, *tablets* e *e-readears*

Se há algum tempo atrás ter um computador portátil, o conhecido *notebook* era novidade, hoje os *smartphones*, os *tablets* e os *netbooks* suprem expectativas e necessidades muito maiores. Por serem equipamentos portáteis, conseguimos utilizá-los facilmente na rua, ônibus ou no trabalho. O mais interessante ainda é poder colocá-los no bolso e na mochila, visualizar e enviar *e-mails*, acessar a internet e as redes sociais a qualquer hora e em qualquer lugar.

As vantagens são muitas, mas você sabe a diferença de cada um?

Notebooks

Os *notebooks* podem fazer praticamente tudo o que um *desktop* (computador de mesa) faz, com a vantagem de ser portátil e consumir menos energia. Nele é possível criar e editar vídeos e fotos, assistir a filmes em alta definição e brincar com jogos para computador.

Netbooks

Os *netbooks* são menores que os *notebooks*, foram feitos para executarem tarefas simples, como checar *e-mails*, editar textos, navegar na internet e fazer apresentações leves. Suas vantagens são o peso e a resistência, um *netbook* costuma pesar apenas 1kg e sua bateria dura mais tempo, porque consome menos energia. Outra vantagem é que muitos aparelhos vêm com chip de telefonia móvel, o que possibilita acesso a redes 3G.

Smartphones

Smartphone é o nome dado ao tipo de telefone celular que incorpora tecnologias antes só vistas em computadores ou *notebooks*, como o acesso à internet, e-mail e sistema operacional. Com o *smartphone* é possível sincronizar dados com o computador pessoal, ter acesso à internet e a redes sociais, jogos e utilizar *bluetooth* e GPS. Os *smartphones* possuem teclado normal ou tela sensível ao toque. Muito prático, pode ser carregado no bolso, porém não foi feito para digitar grandes arquivos.

Tablets

Tablet é um computador em forma de prancheta eletrônica, sem teclado e com tela sensível ao toque. Todos os *tablets* vêm com conexão Wi-Fi e alguns também possuem conexão 3G. O foco

dos *tablets* é a navegação na internet, pois é difícil trabalhar em softwares mais pesados, como Photoshop. Com o *tablet*, além dos games, você pode utilizar simuladores de guitarra até programas educativos.

E-readers

E-reader é um leitor de mídias digitais, podemos utilizar como exemplo os livros digitais chamados de *e-books*. O *e-reader* possui tela *e-ink* que facilita a leitura, pois não deixa a vista tão cansada como numa tela normal de LCD. Os *e-readers* têm a vantagem do peso, são superleves e de fácil locomoção. Outra vantagem legal é que o *e-reader* possui ferramenta de anotações que permite sublinhar ou rabiscar o texto sem qualquer tipo de alteração ao livro original.

Agora que você conhece todas as diferenças e funcionalidades, é fácil escolher um aparelho que se adapte ao seu estilo e necessidade.

Disponível em: <http://www.univesp.ensinosuperior.sp.gov.br/preunivesp/2325/veja-as-diferen-as-entre-notebooks-netbooks-smartphones--tablets-e-e-readears.html>. Acesso em: 06 mai. 2015.

Mais informações acesse: https://www.youtube.com/watch?v=Izufk1Gayrc.

Nesse sentido, é fundamental para um profissional de qualquer área do conhecimento dominar, pelo menos, as funções básicas de um computador, como por exemplo a edição de textos e a navegação na internet. Contudo, para profissionais das áreas exatas, é necessário um conhecimento mais profundo a respeito das tecnologias envolvidas no processo de computação, uma vez que **soluções de prateleira** nem sempre atendem às necessidades destes usuários. Portanto, é necessário que estes profissionais possuam um conhecimento mais avançado para executar suas simulações, projetos e pesquisas, uma vez que as ciências destas áreas estão em constante aperfeiçoamento.

Solução de prateleira

Jargão técnico utilizado para se referir a soluções já existentes e que podem ser compradas com facilidade. Programas antivírus são um exemplo típico deste tipo de produto.

Dentro da área de exatas, os profissionais de diferentes carreiras possuem diferentes atribuições em relação à informática/computação. Aos engenheiros, por exemplo, cabe a concepção dos modelos para processos produtivos, cálculos estruturais, simulações e também a construção de protótipos de programas para estas e outras tarefas. Aos profissionais de computação cabem a análise de complexidade e também o aperfeiço-

amento dos mencionados protótipos de programas, através de técnicas computacionais. Estas diferentes atribuições em relação a uma mesma tarefa se dão pela competência de cada profissional. Um programador, por exemplo, não tem a competência para fazer um projeto de um navio, pois ele não possui em sua formação os conceitos físicos e matemáticos necessários e que são plenamente conhecidos por um engenheiro naval. O engenheiro naval por sua vez não possui conhecimentos de computação gráfica, bancos de dados, integração multiplataforma e multilinguagem entre outros que são necessários para a automatização do processo de construção de navios. Nesse sentido, a formação dos dois profissionais se complementam para a execução de um projeto e, para que o sucesso do projeto seja alcançado, é necessário o uso de uma forma de comunicação comum entre estes profissionais: os **algoritmos**.

Algoritmo

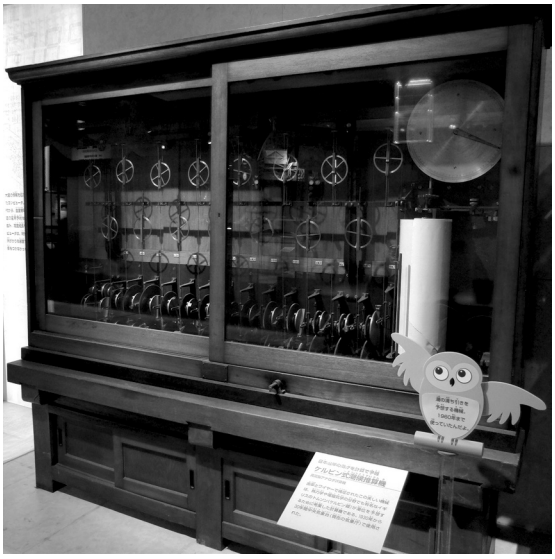
Conjunto finito de passos bem definidos para resolver um problema.

Nesta disciplina, serão estudados algoritmos em suas diversas formas, com o objetivo de promover o aperfeiçoamento e uma melhor comunicação entre os profissionais de diferentes áreas do conhecimento. O uso de estratégias e ferramentas necessárias para a criação de algoritmos eficientes para a resolução de problemas físicos, matemáticos e da engenharia também serão estudados. Por hora, vamos começar pelo básico: o conceito de computador.

Mas o que exatamente é um computador?

Ao lidar com o mundo da informática, o primeiro objeto que vem a mente é o computador. Um computador é uma máquina programável, ou seja, que pode fazer várias operações de acordo com a vontade do usuário. Segundo Farrer (1989), existem dois tipos de computadores:

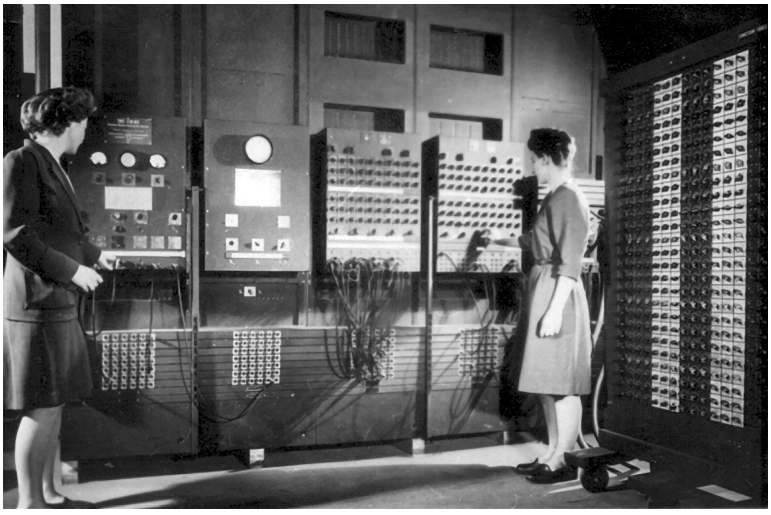
- Os analógicos, que lidam com grandezas físicas, como por exemplo a máquina de prever marés de Kelvin.



Momotarou2012

Figura 1.2: Máquina de prever marés de Kelvin.
Fonte: http://commons.wikimedia.org/wiki/File:Kelvin%27s_tide_predictor.jpg

- Digitais, que lidam com informações que representam grandezas físicas e/ou lógicas. Exemplo: Eniac.



Exército norte-americano

Figura 1.3: Eniac, considerado o primeiro computador digital eletrônico.
Fonte: http://commons.wikimedia.org/wiki/File:Two_women_operating_ENIAC.gif

Os computadores que encontramos no dia a dia são digitais, ou seja, utilizam informações para fazer suas operações. Estes computadores são essencialmente um conjunto de peças, também chamadas ***hardware*** no

Hardware

Nome dado aos componentes físicos (peças) dos computadores. Exemplo: processador, memórias, monitor etc.

Sistema operacional

Conjunto de programas responsável por gerenciar, proteger e prover abstração sobre o *hardware* bem como prover funcionalidades para outros aplicativos/programas. Alguns dos sistemas operacionais mais conhecidos são o Windows, o Android, o Linux e o Solaris.

jargão técnico. Para prover a interação entre estas diferentes peças, todo computador possui pelo menos um **sistema operacional** (S.O.), que é o responsável por gerenciar todos os recursos do computador.

A **Figura 1.4** mostra, em forma de camadas, como se dão as interações entre os usuários, os programas e o *hardware*. Note que os usuários comunicam-se com os programas por meio de janelas ou comandos de texto e os programas se comunicam com o S.O. O sistema operacional, por sua vez, se comunica com o *hardware* e com os programas. Logo, pode-se perceber que uma das funções do sistema operacional é servir de intermediário entre os programas e o *hardware*, escondendo detalhes de *hardware* para os programas em uso. Um bom exemplo disso é a gravação de um arquivo em disco. Ao baixar um arquivo da internet, para salvá-lo no disco, o usuário só precisa informar a pasta onde quer que o arquivo seja armazenado. Detalhes como o número de rotações por segundo, sentido de rotação dos discos ou posicionamento da cabeça de gravação são gerenciados pelo sistema operacional. É importante mencionar que a **Figura 1.4** expressa o processo de comunicação usual. Em alguns casos, é possível fazer com que programas acessem o *hardware* diretamente.

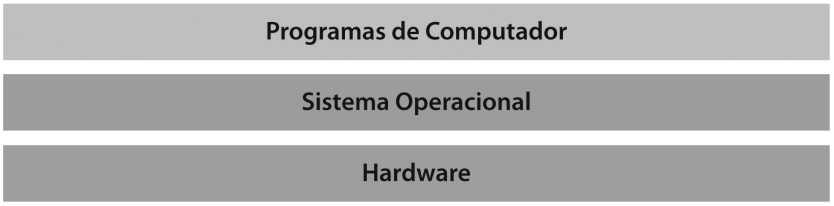
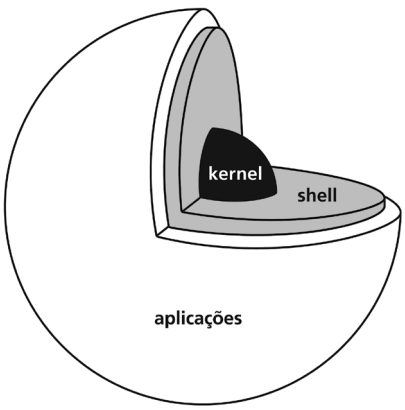


Figura 1.4: Comunicação entre programas, S.O. e o *hardware*.

Os sistemas operacionais podem ser divididos em duas camadas, chamadas *Kernel* e *Shell*. A camada que é responsável por interagir diretamente com o *hardware* ao *kernel*, ou núcleo, do sistema. Esta camada é responsável por gerenciar as peculiaridades de cada dispositivo para que isto fique transparente para o usuário. É nesta camada que ficam os *drivers* de dispositivos, que são programas que fazem acesso, controle, gerenciamento de cada dispositivo sendo, em muitos casos, fornecidos pelos fabricantes de cada componente. A camada *Shell* é responsável por fornecer ser-

viços e funcionalidades para o usuário e para outras aplicações. Existem basicamente dois tipos de *shell*, os de linha de comando e os de interface gráfica. Em ambos os tipos, as funções são praticamente as mesmas. A grande diferença é o modo de apresentação. Frequentemente, para melhor uso de sistemas operacionais e dos computadores, é necessário o uso simultâneo dos dois tipos de *shell*, visto que cada um deles é mais adequado para a realização de determinadas tarefas.



A imagem representa graficamente a ideia da divisão dos S.O. em camadas. A palavra *shell*, que significa concha em inglês expressa a ideia de algo que envolve e protege um núcleo, neste caso o *kernel* do sistema. Na imagem, também são mostradas aplicações que são construídas sobre o *shell*. Estas aplicações são programas de alto nível, navegadores de internet, editores de texto e organizadores de arquivos, que necessitam de serviços providos pelo shell para que possam executar suas tarefas.

Atividade 1

Atende ao Objetivo 1

Identifique os componentes de *hardware*, presentes na imagem a seguir:

Pela imagem você consegue identificar qual é o sistema operacional que o usuário está utilizando neste computador? Se sim, qual é?

Resposta Comentada

Não deve ter sido muito difícil para você identificar onde está o *hardware*, não é mesmo? No caso do computador que vemos na imagem, o hardware são os componentes físicos, ou seja, a torre, monitor, teclado, mouse e caixas de som. No monitor, vemos a imagem de um pinguim, que é o símbolo do sistema operacional Linux. Portanto, podemos supor que o sistema operacional em uso seja o Linux.

Agora que você já sabe o que é *hardware*, vamos estudar duas funções essenciais para o funcionamento do computador.

Gerenciamento de memórias e processadores

Como já vimos na seção anterior, os S.O. são responsáveis por gerenciar todos os recursos dos computadores. Porém, dois recursos devem ser destacados devido ao impacto direto no desempenho do computador e dos programas: processador(es) e memórias. Para entender o gerenciamento de processadores usaremos uma analogia. Primeiro, deve-se saber que um programa é constituído de partes menores, chamadas instruções. Todos os processadores possuem um limite máximo de instruções que podem executar por unidade de tempo. Nos processadores modernos, esta taxa chega a mais de 1 bilhão de instruções por segundo, ou seja, mais de 1 giga-hertz. Para que as instruções possam ser processadas, elas devem passar por vários estágios do processador. Logo, fazendo uma analogia de um processador com uma estrada, quando temos poucos carros (instruções), os carros podem desenvolver a velocidade máxima permitida. No entanto, quando temos um excesso de veículos, ocorre congestionamento. Por isso, quando um computador está aparentemente lento, ao fechar programas conseguimos um melhor desempenho. Isto ocorre porque, ao fechar programas, estamos retirando instruções (veículos) do processador (estrada).

Tóth István

Figura 1.5: Quando um número muito grande de instruções é passado para o processador ocorre um acúmulo de instruções, assim como ocorre um acúmulo de carros devido ao grande número de usuários nas estradas, caso da pista da direita.
Fonte: http://www.sxc.hu/photo/341664

O gerenciamento de memórias, principal e secundária, também é feito pelos sistemas operacionais. Nos computadores modernos, sempre existem dois tipos de memória:

- a principal, geralmente representada por memórias RAM;
- a secundária, geralmente representada pelos discos rígidos.

As características destas memórias são determinantes para a existência destes dois tipos dentro de um computador. A memória principal é muito mais rápida, porém é volátil, o que significa que seu conteúdo é apagado quando o fornecimento de energia é cortado. A memória secundária não é volátil, porém é muito mais lenta do que a principal. Todo o programa para ser executado deve, primeiro, estar em memória principal para depois ser passado para o processador. Entretanto, nem sempre um programa cabe na memória juntamente com os dados sobre os quais é executado.

Imagine um computador de 512 MB de memória RAM, e um usuário que quer ver um vídeo de alta definição baixado da internet. Estes tipos de vídeos frequentemente possuem mais de 1 GB de tamanho. Supondo que o programa de execução de vídeo do computador tenha 20 MB de tamanho e que o vídeo tenha 1.200 MB, seriam necessários 1.220 MB de memória RAM para que ele fosse completamente armazenado em memória principal. Como vários programas são executados ao mesmo tempo, dificilmente seria possível ter, em um computador, memória principal suficiente para rodar todos os programas juntamente com seus dados. A solução encontrada é chamada paginação.

Unidade	Sigla	Quantidade
Bit	b	uma informação binária (0 ou 1)
Byte	B	8 b
Kilobyte	KB	1.000 B
Megabyte	MB	1.000 KB
Gigabyte	GB	1.000 MB
Terabyte	TB	1.000 GB
Petabyte	PB	1.000 TB

A tabela mostra a relação entre as diferentes unidades de tamanho de arquivos utilizadas nos computadores. O *bit* é a unidade básica de informação. Um *bit* armazena um único valor binário, ou seja,

um *bit* pode armazenar o valor 0 ou o valor 1. Um *byte* é composto por 8 bits. Um *kilobyte* é composto por mil *bytes*. Da mesma forma, um *megabyte* é composto por mil *kilobytes*, um *gigabyte* é formado por mil *megabytes* e assim por diante.

A memória ocupada por um programa e seus dados é dividida em seguimentos, chamados páginas, e apenas algumas destas páginas (em geral as que foram usadas mais recentemente) são mantidas em memória principal. O restante das páginas é mantida em uma memória virtual, que nada mais é do que um espaço no disco rígido destinado exclusivamente para isso. Quando um programa precisa de uma informação que não está em memória principal, o sistema operacional faz uma troca de páginas substituindo uma página da memória principal pela página do disco que contém a informação solicitada. Apesar de ser uma solução simples e elegante, o uso de memória virtual não é eficiente. Como os discos rígidos são muito mais lentos que as memórias principais, o uso frequente de paginação pode acarretar impactos consideráveis no desempenho dos programas.

Atividade 2

Atende ao Objetivo 1

Explique o que aconteceria se os computadores não tivessem memória secundária.

Resposta Comentada

A memória secundária é utilizada para manter dados na ausência de energia. Assim, sem esta memória não seria possível salvar textos, fotos etc. Um exemplo da falta que este tipo de memória faria: imagine que você precise imprimir um texto. Você digita, manda imprimir e desliga o computador. Se algum tempo depois você precisar imprimir o mesmo texto sem a presença da memória secundária, você teria que obrigatoriamente digitá-lo novamente. Com a memória secundaria é bem mais simples. Basta salvar o arquivo digitado em uma pasta e abri-lo novamente quando necessário.

Com os conceitos de *hardware* e sistema operacional, você está pronto para entender a próxima seção.

Como fazer computação

A computação é feita através da execução de programas de computador capazes de analisar as informações, fazer as transformações necessárias, realizar as operações aritméticas e lógicas requeridas e mostrar os resultados. Nesse sentido, o conceito de programa de computador é fundamental para a realização da computação. Todavia, para entender o conceito de programa é necessário entender o conceito de algoritmo.

Não há consenso na literatura sobre a etimologia da palavra algoritmo. A hipótese mais aceita é a de que ela tenha origem do sobrenome Al-Khwarizmi, de um matemático persa do século IX. Outras vertentes dizem que deriva do árabe *Al-goreten*, que quer dizer "raiz" ou ainda do grego *arithmos*, que significa "número".

Selo da URSS em homenagem a Al-Khwarizmi.

Fonte:http://upload.wikimedia.org/wikipedia/commons/9/93/Abu_Abdullah_Muhammad_bin_Musa_al-Khwarizmi_edit.png

Algoritmo

Como mencionado anteriormente, um *algoritmo* é um conjunto finito de passos bem definidos para resolver um problema. Apesar de o conceito de algoritmo ser muito antigo, visto que há relatos de algoritmos dos tempos da Grécia antiga, este conceito só foi formalizado em 1936 pelo matemático britânico Alan Turing.

Para executar um algoritmo e obter o resultado desejado deve-se seguir, metodicamente, cada um de seus passos. Em alguns casos é necessário fornecer as entradas ao algoritmo que são informações externas necessárias para sua execução. Contudo, estas entradas nem sempre são necessárias.

A vida de Alan Turing, considerado o pai da informática, já foi filmada algumas vezes. Você pode conferir uma dessas adaptações, chamada *Breaking the Code* (1996) produzida pela rede BBC. Está prevista para 2014 uma nova adaptação cinematográfica sobre sua vida.

Fonte: http://upload.wikimedia.org/wikipedia/commons/5/57/Alan_Turing.jpg

A **Figura 1.6** mostra as relações de um algoritmo com suas entradas e saídas. As entradas são tudo aquilo que é necessário ao funcionamento do algoritmo. O algoritmo é a sequência finita de passos para resolver o problema/tarefa e a saída é a solução/conclusão do problema/tarefa.

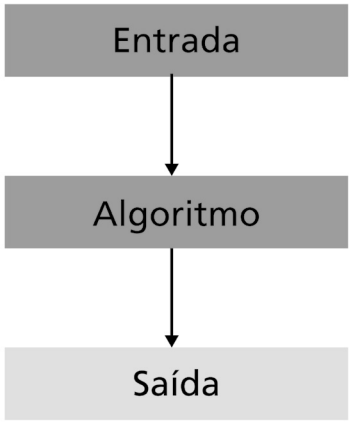


Figura 1.6: Relação entre um algoritmo, entrada e saída.

Apesar de ser um conceito pouco difundido, todos nós executamos/construímos algoritmos todos os dias. Um exemplo clássico é o deslocamento de um ponto a outro da cidade, utilizando um carro. Para fazer o deslocamento, é necessário traçar, mesmo que mentalmente, um trajeto. Para fazer o trajeto são necessárias algumas informações tais como: ponto

de origem, ponto de destino, condições do trânsito, sentido das ruas etc. Uma vez montado o trajeto, caso ele seja seguido passo a passo e não ocorram imprevistos, uma pessoa poderá chegar ao ponto de destino.

Outro exemplo muito citado são as receitas de cozinha. Para fazer uma musse de maracujá são necessários os seguintes ingredientes:

Receita de Musse de Maracujá

-1 lata de leite condensado,

-1 lata de creme de leite sem soro,

-1 lata (a mesma medida) de suco de maracujá concentrado,

-1 envelope de gelatina em pó sem sabor.

Tendo em mãos os ingredientes, basta bater no liquidificador o leite condensado, o creme de leite sem soro e o suco de maracujá. À parte, prepare a gelatina em pó, conforme instruções da embalagem. Quando amornar, coloque para bater juntamente com os demais ingredientes que estão no liquidificador. Unte uma forma de pudim com margarina e, antes de colocar o creme, passe água pela forma, como se fosse enxaguá-la. Coloque no congelador até estar no ponto para tirar da forma. Neste exemplo, a entrada são os ingredientes, o algoritmo são os passos da receita e a saída é a própria musse de maracujá.

Gilberto Santa Rosa

Figura 1.7: Você já tinha imaginado que o resultado de um algoritmo poderia ser tão saboroso?
Fonte: http://upload.wikimedia.org/wikipedia/commons/6/62/MouseDeMaracuja.jpg

Algoritmos e ambiguidades

Um problema recorrente encontrado em algoritmos como o da receita descrita é a ambiguidade de suas instruções. Várias sentenças do algoritmo podem ser consideradas ambíguas como, por exemplo, "Quando amornar, coloque para bater...". Nesta sentença, a expressão "amornar" não é precisa. Qual a temperatura que pode ser considerada morna? O que é considerado morno para uma pessoa também é considerado morno para todas as outras? A presença de expressões dúbias na linguagem escrita convencional a torna inadequada para escrever algoritmos que serão executados por computadores, visto que estes devem ser claros e não podem possuir ambiguidades.

Uma das formas mais utilizadas para representar algoritmos que serão executados por computadores são os pseudocódigos. Neste tipo de descrição, os algoritmos são executados linha a linha, de cima para baixo, e os comandos de cada linha são executados de acordo com sua precedência. Exemplos clássicos e modernos do uso deste tipo de representação podem ser encontrados em vários textos como Ascencio, Campos (2012) e Cormen et al. (2009).

A **Figura 1.8** mostra um exemplo de algoritmo simples escrito em pseudocódigo. Este algoritmo é concebido para escrever a mensagem "teste" na tela.

```
Algoritmo testeImpressao( )
Início
        Imprimir "teste"
Fim
```

Figura 1.8: Um algoritmo simples para imprimir na tela a palavra "teste".

Atividade 3

Atende ao Objetivo 2

Assinale as possibilidades a seguir com V para verdadeiro e F para falso em relação às características desejadas para um algoritmo.

( ) Claro

( ) Ambíguo

( ) Finito

( ) Possuir passos bem definidos

Resposta Comentada

Resposta: V, F, V, V

Dentre as características mencionadas, apenas a ambiguidade não é uma característica desejada para um algoritmo. Algoritmos devem ser claros, ou seja, sem ambiguidade, finitos, pois não podem ser executados indeterminadamente e possuir passos (instruções) bem definidas.

Agora que o conceito de algoritmo já esta definido, já é possível definir o conceito de *programa de computador*.

Programa de computador

Um programa de computador nada mais é do que a implementação de um algoritmo que pode ser executada pelo computador. Em outras palavras, um programa é um algoritmo convertido em uma linguagem que o computador possa compreender. Para solucionar um problema através do uso de programas de computador, são necessárias várias etapas.

A primeira é a elaboração de um algoritmo para solucionar este problema. Existem algoritmos piores ou melhores dependendo da situação. Normalmente, um algoritmo é dito melhor que outro se consegue resolver o mesmo problema em menos passos.

Existe toda uma fundamentação teórica dentro da Ciência da Computação dedicada à análise de complexidade de algoritmos. Pesquisadores se utilizam desta fundamentação para produzir algoritmos mais eficientes.

Linguagem de programação

Linguagem com sintaxe e semântica bem definidas que é usada na construção de programas de computador. Alguns exemplos de linguagens de programação são C++, criada na década de 1980 e utilizada para computação de alto desempenho, e Java, criada na década de 1990 e utilizada para programação web, de dispositivos móveis e multiplataforma.

Código fonte

Transcrição de algoritmo seguindo as regras sintáticas e semânticas de uma linguagem de programação.

A segunda etapa é escolher uma **linguagem de programação**, na qual o algoritmo será escrito. Linguagens de Programação são como ferramentas, dependendo do problema a ser resolvido deve-se utilizar uma ferramenta em detrimento de outra. Por exemplo, se o problema for tomar uma sopa, uma colher é a ferramenta mais indicada. Agora se o problema for cortar uma árvore, certamente existem ferramentas melhores do que uma colher. A escolha da linguagem de programação também deve ser feita em função do problema abordado. Se o problema requer alto desempenho computacional, uma boa sugestão é C++. Agora, se o problema envolve execução em múltiplas plataformas e/ou programação web, Java pode ser uma escolha mais adequada.

Uma vez escolhida a linguagem de programação, tem início a fase de codificação, que é a transformação do algoritmo em **código fonte.** Uma vez que o código fonte esteja concluído, o próximo passo é a compilação deste código fonte. Esta compilação transformará o código fonte em uma linguagem binária (composta por 0s e 1s), que é de fato a única linguagem que os computadores digitais atuais conseguem interpretar – veja Monteiro (2002) –, para mais detalhes, ou para uma breve explicação, o *link* http://www.oficinadanet.com.br/artigo/1347/. A tradução binária do código fonte é o que chamam Programa de computador. Logo, assim que o processo de compilação estiver concluído, o programa pode ser executado.

Apesar de o processo de codificação ser imprescindível para o processo de computação, este não é o foco desta disciplina por duas razões:

a) a concepção de algoritmos já é, por si só, um assunto complexo;

b) um algoritmo bem-feito pode ser transcrito para várias linguagens.

(a)

```
#include<iostream>

int main(int argc, char** argv )
{
  std::cout<<"teste\n";
}
```

(b)

```
program imprimirTeste
  write(*,*)"teste"
end program imprimirTeste
```

Figura 1.9: Transcrição do algoritmo da Figura 1.8 para duas linguagens: C++(a) e Fortran90 (b).

A **Figura 1.9** mostra a transcrição do algoritmo exposto na **Figura 1.8** para duas linguagens: C++ (1.9a) e Fortran90 (1.9b). O resultado dos dois programas é o mesmo: a mensagem "teste" é escrita na tela. Note que apesar

de o resultado do programa ser o mesmo, as linguagens são sintaticamente diferentes, o que mostra que o mesmo algoritmo pode ser escrito em várias linguagens através do uso da sintaxe específica de cada linguagem.

Portanto, a concepção de um algoritmo bem-feito é tão importante para a computação quanto a construção bem-feita de um programa. Nesse sentido, ao longo desta disciplina será mostrado como conceber algoritmos para diversas tarefas relacionadas à engenharia utilizando pseudocódigos, objetivando despertar o raciocínio lógico do leitor para que você possa, em um passo posterior, implementar seus algoritmos na linguagem de sua preferência.

Atividade Final

Atende aos Objetivos 1, 2 e 3

1. Refaça a receita de musse tentando retirar todas as ambiguidades existentes.

2. Monte um roteiro para se deslocar da sua residência até um local onde você tenha um compromisso (trabalho, escola etc.). Descreva bem os passos como, por exemplo, horário para sair de casa, em que direção caminhar, qual a linha de ônibus tomar etc. O objetivo deste exercício é que uma pessoa que não conheça o caminho consiga reproduzir o trajeto feito por você.

Resposta Comentada

1. Como comentado anteriormente, a ambiguidade na receita está presente nas expressões "amornar" e "ponto". Uma sugestão para retirar estas ambiguidades é substituí-las por expressões mais precisas como, por exemplo: quando a mistura estiver na temperatura de 25 graus Célsius, coloque no congelador por 15 minutos. Tome cuidado para não invalidar a receita ao fazer as mudanças.

2. Um ponto desafiador no segundo item é descrever um trajeto sem pontos de referência. Muitas vezes fazemos algoritmos que utilizam pontos de referência conhecidos como, por exemplo, "caminhe até o mercado popular e depois vire à esquerda." Este tipo de trajeto é relativamente simples para pessoas que conhecem a cidade, mas para alguém que não conheça o mercado popular, como um turista, por exemplo, pode ser difícil percorrer o trajeto sem ajuda ou informação de outras pessoas. Outro ponto crítico são as descrições que usam tempo. Por exemplo: "caminhe por 15 minutos". Deve-se considerar que cada pessoa tem um ritmo de caminhada diferente. Logo, o que um jovem percorre em 15 minutos, um idoso levará mais tempo. Seria mais fácil descrever o trajeto em termos de grandezas que não variam de pessoa para pessoa, como, por exemplo, "caminhe por 5 quadras".

Resumo

Nesta aula, você viu que a área de informática tem grande importância para todas as áreas do conhecimento e em especial para a área de exatas e que as diferentes profissões se relacionam de maneira diferente com esta área do conhecimento. Aprendeu conceitos-chaves da área, como os de computador, programa de computador e algoritmo. Também foram apresentados o processo de criação de programas, que um mesmo algoritmo pode ser escrito em várias linguagens, e os problemas que te-

mos ao descrever algoritmos utilizando a linguagem natural, bem como a solução deste último problema, que é o uso de pseudocódigo.

Informação sobre a próxima aula

Na Aula 2, serão introduzidos os conceitos iniciais e a sintaxe básica para a criação de algoritmos simples em pseudocódigo. Nas demais aulas, serão introduzidas novas ferramentas lógicas, que permitirão a concepção de algoritmos para resolver tarefas mais complexas.

Aula 2

Sintaxe básica

Tiago Araújo Neves

Meta

Expor os conceitos iniciais para a criação dos primeiros algoritmos.

Objetivos

Esperamos que, ao final desta aula, você seja capaz de:

1. criar algoritmos capazes de fazer operações matemáticas;
2. utilizar variáveis na construção de um algoritmo;
3. fazer com que os algoritmos interajam com o usuário.

Sintaxe, o que é isso mesmo?

Compare as frases a seguir:

Robert Nagy

Fonte: http://www.sxc.hu/photo/4243

O gato senta na cadeira.

senta O cadeira. na gato

Rose Ann

Fonte: www.sxc.hu/544853

Percebeu como a mudança na ordem dos componentes da frase fez com que ela se tornasse incompreensível? No estudo das línguas, a sintaxe é o conjunto de regras que ordena as palavras na estrutura da frase, tratando das relações de dependência entre as palavras.

Agora você deve estar se perguntando: mas o que isso tem a ver com a informática?

A informática, por definição, é a ciência da informação. Os computadores são máquinas que processam informação. Logo, faz parte do processamento de informação a comunicação, ou seja, transmitir e/ou trocar informações.

Para que a comunicação seja efetiva, aquele que transmite e aquele que recebe a informação devem ser capazes de se compreender mutuamente. Deste modo, padronizações na sintaxe dos algoritmos são necessárias para que outras pessoas possam compreender os algoritmos que você escreve.

Sintaxe básica

Para padronizar os algoritmos, devemos estabelecer uma estrutura para servir de base. Esta estrutura estará presente em todos os algoritmos descritos nesta disciplina; portanto, deve ser entendida com clareza.

A sintaxe de descrição para um algoritmo é mostrada na **Figura 2.1**.

```
Algoritmo nome( <lista de parâmetros> )
Início
  /*corpo*/
Fim
```

Figura 2.1: Sintaxe de um algoritmo.

Como mostrado na **Figura 2. 1**, todo algoritmo começa com a palavra reservada "Algoritmo", seguida de um nome para o algoritmo. Logo após o nome, está a lista de parâmetros de entrada envolvida por um par de parênteses obrigatórios (mesmo que a lista de parâmetros seja vazia). Note que a lista de parâmetros está envolvida pelos sinais <>. Isto indica que esta lista é opcional, já que podem existir algoritmos sem parâmetros de entrada. Na linha 2 do algoritmo, está a palavra reservada "Início", que indica o começo do algoritmo. Na linha 4, está a expressão "/*corpo*/". Na sintaxe utilizada nesta disciplina, tudo o que estiver entre as marcações "/*" e "*/" são comentários, ou seja, estão ali para dar uma breve explicação, mas não fazem parte do algoritmo. Comentários são muito usados para esclarecer pontos obscuros e ideias. Neste caso específico, o comentário foi colocado para mostrar onde ficará o chamado "corpo do algoritmo", ou seja, o conjunto de passos para resolver o problema. É importante mencionar que o corpo de um algoritmo pode ser composto de várias linhas. A última linha do algoritmo contém a palavra reservada "Fim".

Existem outras sintaxes que utilizam outras formas de introduzir comentários. Por exemplo, na linguagem C, também utilizam-se duas barras (//), e na Fortran, um ponto de exclamação (!).

Dentro da computação, existe um conjunto de regras utilizado para dar nomes aos algoritmos. Os nomes utilizados são os chamados **identificadores válidos**. Algumas linguagens de programação, C e Java entre elas, diferenciam letras maiúsculas de minúsculas. Por fins didáticos, esta diferenciação não será feita nesta disciplina.

Identificador válido

Conjunto de caracteres composto por letras, números e o caractere sublinhado (_). Este conjunto sempre começa com uma letra e não pode conter espaços (DEITEL; DEITEL; 2011).

Um ponto importante sobre identificadores válidos é que eles devem, na medida do possível, expressar claramente a finalidade para a qual serão utilizados. Por exemplo: suponha que você está criando um algoritmo para calcular a raiz quadrada de um número. Qualquer identificador válido pode ser utilizado para dar nome a este algoritmo, como a1b2, abcd ou ainda raizQuadrada. Contudo, é considerada uma boa prática, principalmente na computação, escolher identificadores que expressem com clareza a finalidade do algoritmo. Neste caso, o identificador raizQuadrada seria o mais indicado.

O conceito de identificador válido será sempre utilizado para dar nomes aos algoritmos, variáveis e funções. Portanto, é necessário sempre ter este conceito em mente.

Atividade 1

Atende ao Objetivo 1

Assinale V para verdadeiro ou F para falso, verificando se as seguintes cadeias de caracteres são identificadores válidos ou não.

( ) abcd

( ) a1b2

( ) 1a2b

( ) teste_1

( ) teste 1

Resposta Comentada

A terceira cadeia de caracteres e a última não respeitam as regras de formação de identificadores válidos. A terceira começa com um número e a última contém espaços. As demais opções satisfazem todas as regras, portanto são identificadores válidos.

Comando de impressão

Todo programa e, consequentemente, todo algoritmo, deve produzir pelo menos um resultado. Logo, é necessário um mecanismo para que os algoritmos possam exibir suas saídas. Um destes mecanismos é o comando de impressão. Este comando é utilizado para enviar mensagens simples para o usuário na forma de texto. Como mostrado na **Figura 1.5** da Aula 1, diferentes linguagens possuem diferentes comandos de impressão. Contudo, também foi visto que um mesmo algoritmo pode ser transcrito para diversas linguagens. Portanto, vamos definir um comando de impressão geral. Caso você queira utilizar o comando de impressão em alguma linguagem de programação, tudo o que deve fazer é procurar a sintaxe deste comando na linguagem de sua preferência.

O comando de impressão utilizado nesta disciplina será o comando "imprimir", e terá a seguinte sintaxe:

imprimir *expressão*

A *expressão* logo após o comando imprimir contém aquilo que será exibido para o usuário (normalmente, exibido na tela dos computadores). É importante mencionar que cada comando *imprimir* ocupa uma linha durante a exibição. Assim, se o algoritmo tem dois comandos *imprimir*, serão usadas duas linhas.

A expressão a ser exibida pode assumir várias formas: pode ser impresso um conjunto de caracteres, o resultado de uma expressão matemática, o valor de uma variável, dentre outras. Começaremos com impressão de expressões de texto. Toda expressão de texto (conjunto de caracteres) deve ser escrita entre aspas duplas (" "). Assim, se quisermos que um progra-

ma imprima a expressão “Bom Dia” para o usuário, podemos utilizar o seguinte algoritmo:

```
Algoritmo imprimirBomDia( )
Início
        Imprimir “Bom Dia!!!”
Fim
```

Figura 2.2: Uso do comando *imprimir* para a expressão *Bom dia!!!*.

Note que o algoritmo da **Figura 2.2** segue toda a sintaxe básica estabelecida na seção anterior. Ele começa com a palavra reservada *Algoritmo,* que é seguida de um nome sugestivo e de uma lista vazia de parâmetros (nas aulas posteriores, você aprenderá a utilizar entradas para algoritmos. Por enquanto, não utilizaremos este recurso). Nas linhas logo abaixo, estão as palavras reservadas *Início* e *Fim,* e entre elas está o *corpo do algoritmo*. Neste caso, o corpo é composto de apenas uma linha, que é o comando *imprimir* seguido da expressão “Bom Dia!!!”. Este algoritmo faz com que a mensagem “Bom Dia!!!” seja exibida para o usuário exatamente como o algoritmo da **Figura 1.4** da Aula 1 (e, consequentemente, os códigos fonte da **Figura 1.5**) exibe a mensagem “teste”.

Um outro exemplo de uso do comando *imprimir* pode ser visto na **Figura 2.3**. Neste caso, o algoritmo é feito para exibir o resultado final de uma expressão aritmética.

```
Algoritmo imprimirSoma( )
Início
        Imprimir 5 + 2
Fim
```

Figura 2.3: Imprimindo resultado de uma expressão.

No caso da **Figura 2.3**, o que é exibido para o usuário é o número 7, que é o resultado da avaliação da expressão 5 + 2. É importante mencionar que, apesar da expressão conter espaços entre os termos, estes espaços não são obrigatórios; portanto, a expressão poderia ser escrita 5+2, e o resultado seria o mesmo. Os espaços foram colocados simplesmente por questão de clareza. O resultado obtido por este algoritmo ocorre porque o comando imprimir é executado depois que todas as operações forem efetuadas. Normalmente, os comandos são executados da esquerda para a direita (do mesmo modo como escrevemos e lemos uma linha na cultura

Precedência (de operações)

Regra que estabelece em que ordem as operações são efetuadas. Um exemplo clássico é a diferente precedência entre a soma e a multiplicação na Matemática. A expressão 2+3x5 tem a operação de multiplicação executada primeiro, apesar de a referida multiplicação estar mais à direita da soma. Isto ocorre porque a precedência da multiplicação é maior que a da soma.

ocidental), exceto quando há **precedências** diferentes envolvidas. O comando imprimir tem a menor precedência em relação à soma, logo a soma é executada primeiro, e o resultado da soma é exibido para o usuário.

Um exemplo interessante do uso do comando imprimir para diferentes tipos de informação pode ser visto na **Figura 2.4**. Note que na figura há dois comandos de impressão. Como vimos, algoritmos podem conter mais de um comando em seu corpo. Nestes casos, a execução é feita sequencialmente de cima para baixo (exceto em casos específicos, que veremos em outra aula).

```
Algoritmo imprimirTextoSoma( )
Início
        Imprimir "5 + 2"
        Imprimir 5 + 2
Fim
```

Figura 2.4: Comando imprimir para diferentes tipos de informação.

Atividade 2

Atende aos Objetivos 1 e 3

Execute o algoritmo da **Figura 2.4** no espaço em branco a seguir. Qual é o resultado de sua execução? Ou seja, o que é exibido para o usuário?

Resposta Comentada

Em uma linha, é exibido o texto *5 + 2* e, na linha subsequente, é exibido o número 7. Como visto, cada comando imprimir utiliza uma linha ao exibir mensagens para o usuário. Portanto, se temos dois comandos imprimir, deveremos ter duas linhas exibidas. Na primeira, é exibido o texto 5+2, porque no algoritmo esta expressão aparece entre aspas, indicando que a expressão a ser impressa é uma expressão de texto. Na segunda linha, a ausência das aspas indica que expressão a ser impressa é o resultado de uma operação, neste caso, aritmética. Assim, a saída para o usuário é mostrada da seguinte maneira:

```
5 + 2
7
```

Diversas operações aritméticas podem ser utilizadas no comando imprimir. A **Tabela 2.1** mostra as operações aritméticas, seus respectivos operadores, exemplos de uso, significado e precedência. Quanto maior a precedência de um operador, maior a prioridade para sua execução. Note que vários operadores têm a mesma precedência. No caso de expressões compostas somente por operadores de mesma precedência, a expressão é avaliada da esquerda para a direita.

Tabela 2.1: Precedência de operadores

Operação	Operador	Exemplo de uso	Significado	Precedência
Potenciação	^	10^2	Dez elevado a dois	4
Inversão de sinal	–	–2	Menos dois	3
Multiplicação	*	2*3	Dois vezes três	2
Quociente	/	6/2	Quociente de seis dividido por dois	2
Módulo da divisão	%	4%3	Resto da divisão de quatro por três	2
Soma	+	2+2	Dois mais dois	1
Subtração	–	5–1	Cinco menos um	1

O operador *módulo da divisão* também é chamado *de resto* da *divisão ou modulus*. Quando fazemos uma divisão de dois números inteiros, obtemos dois resultados: um quociente e um resto. Para acessarmos o QUOCIENTE, usamos o operador "/". Para acessarmos

o RESTO, usamos o operador "%". É importante mencionar que o operador *modulus* só pode ser usado para divisão com números inteiros. A divisão entre dois números reais é sempre um número real; portanto, esta divisão não tem resto e o operador *modulus* perde o sentido. Logo, para divisão de números reais, deve-se usar somente o operador Quociente "/".

Em alguns casos, é necessário mudar a ordem de avaliação das expressões aritméticas. Para este fim, utilizam-se os parênteses. Assim, uma expressão do tipo (2+3)*5 tem a soma avaliada primeiro. É importante mencionar que, na computação, diferentemente da matemática, não existem outros operadores para mudar a precedência de avaliação. A matemática usa, além dos parênteses, colchetes e chaves. Ex.: 7x{[(2+3)*5]−1 }. Na computação, como existe apenas o operador parênteses, a expressão anterior ficaria do seguinte modo: 7*(((2+3)*5)−1). Neste caso, os parênteses mais internos são resolvidos primeiro. Assim, a primeira operação é a soma 2+3. O resultado desta soma é então multiplicado por 5. Do resultado da multiplicação, é subtraída uma unidade e, por último, é feita a multiplicação por 7.

Na computação, números reais normalmente são expressos com um ponto (.) separando a parte inteira da parte decimal.

Atividade 3

Atende ao Objetivo 1

1. Avalie o resultado das seguintes expressões:

1+2*−1

1*2/3

1+4*5−2

1%2−1

2. Indique o resultado das seguintes expressões. Considere que somente números inteiros são utilizados:

2+3*4

(2+3)*4

2+3/4−1

2+3/(4−1)

(4%(3−2))*1

4%3−2*1

3. Coloque parênteses na expressão a seguir, de modo que o resultado final seja 71:

3+6*8−3/7−4

Resposta Comentada

1. −1, 0, 19 e 0. Na primeira expressão, o −1 é avaliado primeiro; em seguida, a multiplicação por 2 e, por último, a soma com 1. No segundo caso, a multiplicação é feita primeiro, resultando em 2. Em seguida, é tomado o quociente da divisão de 2 por 3, que resulta em zero. Na terceira expressão, a multiplicação é feita primeiro; em seguida, a soma e, por último, a subtração. No quarto caso, o módulo da divisão é resolvido primeiro. Como o resto da divisão de 1 por 2 resulta em 1, ao subtrairmos 1, obtemos zero como resultado final.

É importante mencionar que o operador de resto da divisão só pode ser utilizado para números inteiros, de modo que 1/2 resulta em 0 e 1%2 resulta em 1. Se estes números forem reais, o resultado é um número real. Deste modo, 1.0/2.0 resulta em 0.5.

2. 14, 20, 1, 3, 0, −1

3. ((3+6)*8)−3/(7−4)

Variáveis

Variáveis são posições de armazenamento (memória) associadas a um identificador (nome) que são utilizadas para guardar informações (valores). Em linguagens fortemente tipadas (Java e c++, entre elas), as variáveis são também associadas a um tipo, que exprime qual a natureza da informação presente naquela posição de memória (número inteiro, número real, texto, etc.). A sintaxe de declaração de variáveis utilizadas são as seguintes:

tipo nome

tipo nome1, nome2, … , nomen

A primeira é utilizada para declarar uma única variável de um determinado tipo. A segunda declara várias variáveis de um mesmo tipo

Dentre os tipos que serão utilizados, estão:

- inteiro (para números inteiros);
- real (para números reais);
- lógico (para valores lógicos/booleanos);
- texto (para conjuntos de caracteres).

O tipo lógico, também chamado de booleano, pode assumir dois valores: verdadeiro ou falso. São comumente utilizados em estruturas de controle, como desvio (Aula 3) e repetição (Aula 5), para verificar a veracidade de certas sentenças.

Deste modo, uma declaração de uma variável inteira para armazenar a idade de uma pessoa em anos pode ser feita da seguinte forma: *Inteiro idade*. De modo semelhante, a declaração de duas variáveis reais para expressar o salário de uma pessoa e sua altura em metros pode ser feita do seguinte modo: *Real salario, altura*.

Na computação, uma variável é uma posição de armazenamento (uma posição de memória) e o valor da informação armazenado nesta posição pode mudar ao longo da execução de um programa. Neste sentido, as variáveis computacionais são semelhantes às variáveis utilizadas em funções na matemática, visto que nas funções, as variáveis podem assumir diferentes valores.

Para modificar os valores das variáveis, é utilizada a operação de atribuição, para a qual utilizaremos o operador "=". A atribuição pode ser feita no momento da criação de uma variável ou em qualquer outro ponto do algoritmo. Assim, se em alguma parte de um algoritmo escrevermos

```
...
Inteiro Idade =3
Imprimir Idade
...
```

ou

```
…
Inteiro Idade
```

```
Idade=3
Imprimir Idade
```

…

o resultado será o mesmo. O que será exibido é o valor da informação contido na variável idade. Neste caso, *3*.

Diferentes linguagens de programação possuem diferentes tipos e diferentes operadores. É preciso ficar atento com detalhes de sintaxe no momento de transcrever algoritmos em código fonte.

É possível utilizar variáveis em expressões aritméticas e para compor textos mais complexos. A **Figura 2.5** mostra um algoritmo que faz uso de variáveis em expressões matemáticas e em construção de textos compostos.

```
Algoritmo testeVariaveis( )
Início
        Inteiro a=3,b=3,c
        Texto t1= "A soma de a e b é: "
        c=a+b
        Imprimir t1+c
Fim
```

Figura 2.5: Expressões com variáveis numéricas e textuais.

Na primeira linha do corpo do algoritmo, são declaradas três variáveis inteiras: *a*, *b* e *c*. Para *a* e *b,* são atribuídos valores iniciais. Na segunda linha, é criada uma variável textual que contém a expressão "*A soma de a e b é: "*. Na terceira linha, é calculada a soma entre os valores contidos nas variáveis *a* e *b,* e o resultado desta soma é armazenado em *c*. Isto ocorre porque o operador de atribuição tem precedência menor que o operador de soma, logo a soma é feita primeiro. Na última linha do corpo, está o comando *Imprimir t1+c,* que exibe a concatenação do texto contido

em *t1* com o valor contido em *c*. Assim, o que é exibido para o usuário é a expressão *A soma de a e b é: 6* . Note que, apesar de a operação de concatenação também utilizar o operador +, o que está sendo feito não é a soma entre um texto e um número, e sim a justaposição de um texto com um número. A operação de concatenação pode ser utilizada tanto entre números e texto quanto entre textos. O fato de um mesmo operador (neste caso, o +) poder ser utilizado para mais de uma operação se deve à **sobrecarga de operadores**.

Sobrecarga de operador

Capacidade que um operador tem de poder ser utilizado para mais de uma operação. O operador –, por exemplo, pode ser utilizado para duas operações, de acordo com a **Tabela 2.1**: a subtração e a inversão de sinal. O contexto indica à qual operação o operador se refere em um trecho de código.

A **Figura 2.6** mostra um algoritmo cujo valor de uma variável é modificado. Na primeira linha do corpo do algoritmo, é criada uma variável chamada *a* e atribuída a esta variável o valor 3. Em seguida, o valor de *a* é exibido. A terceira linha modifica o valor de *a* para 4. Por último, o novo valor de *a* é exibido.

```
Algoritmo mudandoValores( )
Início
        Inteiro a=3
        Imprimir a
        a=4
        Imprimir a
Fim
```

Figura 2.6: Algoritmo que faz mudança no valor de uma variável.

Assim, o que é exibido pelo algoritmo da **Figura 2.6** são as seguintes linhas:

```
3
4
```

Além da operação de concatenação, outra operação comum em textos é o acesso aos caracteres. Exemplo: suponha que temos uma variável de texto *t1* com a expressão *abcde*. Para termos acesso somente à letra *a*, utilizamos a sintaxe t1[0]. Para acessarmos a letra *b*, utilizamos t1[1], e assim por diante até t1[4], que retorna à letra *e*. Na sintaxe que utilizaremos ao longo das aulas, a posição inicial dos conjuntos de caracteres será zero.

Na computação, existem linguagens em que a contagem de caracteres de texto começa em 0 (c++, Java) e em 1 (Pascal). É preciso atenção para adaptar algoritmos para as características da linguagem utilizada.

A **Figura 2.7** mostra um algoritmo que faz uso do acesso a caracteres. Neste exemplo, é exibido para o usuário primeiro a palavra *casa*. Em seguida, é exibida a última letra da palavra (*a*). Logo após, o último caractere da palavra é alterado e é exibida a palavra *caso*.

```
Algoritmo acessoCaracteres( )
Início
        Texto t= “casa”
        Imprimir t
        Imprimir t[3]
        t[3]= “o”
        Imprimir t
Fim
```

Figura 2.7: Algoritmo que faz uso do acesso aos caracteres de uma variável de texto.

Note que o operador de atribuição pode ser usado para modificar caracteres individuais (linha 4 do corpo do algoritmo da **Figura 2.7**) de uma variável textual, bem como para mudar todo o conjunto de caracteres (linha 1 do corpo do algoritmo da **Figura 2.7**).

Atividade 4

Atende aos Objetivos 2 e 3

1. Faça um algoritmo para exibir o seu nome.

2. Faça um algoritmo que imprima em uma linha o seu nome normalmente e em outra linha imprima como é feito na língua inglesa. Use variáveis de texto. Exemplo: para o nome *João Silva*, o algoritmo deve imprimir em uma linha *João Silva* e na outra *Silva, João*.

Resposta Comentada

Como a primeira questão é parte da segunda, apresentaremos apenas a solução para a segunda atividade.

```
Algoritmo imprimirNome( )
Início
        Texto nome, sobrenome, espaco
        nome = "João"
        sobrenome = "silva"
        espaco = " "
        Imprimir nome+espaco+sobrenome
        Imprimir sobrenome+","+espaco+nome
Fim
```

Neste algoritmo, três variáveis textuais são utilizadas: uma para guardar o nome, uma para guardar o sobrenome e outra para guardar o espaço. Note que existem várias maneiras de se resolver esta questão. Uma delas seria usar a vírgula também como variável. Outra possível seria não utilizar nenhuma variável, como no exemplo a seguir:

```
Algoritmo imprimirNome2( )
Início
        Imprimir "João Silva"
        Imprimir "Silva, João"
Fim
```

Como os dois algoritmos produzem o mesmo resultado final, a impressão de "João Silva" em uma linha e "Silva, João" na linha subsequente, ambos são soluções válidas para o problema.

Comando de leitura

Muitas vezes, é necessário interagir com os algoritmos, dando a eles dados para sua execução. Um exemplo comum disso é o fornecimento do endereço de um site que queremos visitar para o navegador. Para este fim, utilizaremos o comando Ler, que possui a seguinte sintaxe:

Ler *nomeVariavel*

Este comando permite que o usuário altere o valor de uma variável, especificada pelo nome, em tempo de execução. Ou seja, em um algoritmo em que exista uma variável inteira *idade*, e encontramos um comando *Ler idade*, o algoritmo exige que o usuário insira uma informação e armazenará a informação dada na variável *idade*. Quando falamos de computadores, um dos modos de se implementar o comando *Ler* em um programa é permitir que o usuário digite a informação utilizando o teclado. Como estamos lidando com algoritmos, deixemos simplesmente que o comando faça uma interação com o usuário, a fim de obter informação, independentemente de como essa informação é inserida (teclado, janelas, comando de voz, etc.). É importante mencionar que o comando *Ler* é bloqueante, ou seja, enquanto o usuário não inserir a informação, o algoritmo não prossegue sua execução.

A **Figura 2.8** mostra um exemplo de uso do comando *Ler*. Neste algoritmo, duas variáveis têm seu valor alterado por comandos *Ler*. Em seguida, é calculada e exibida a soma dos valores digitados.

```
Algoritmo testeLer( )
Início
        Inteiro a,b
        Ler a
        Ler b
        Imprimir a+b
Fim
```

Figura 2.8: Exemplo de uso do comando *Ler*.

Note que, a princípio, não é possível determinar o que vai ser exibido pelo algoritmo da **Figura 2.8**. Neste caso, a resposta exibida depende de quais informações serão inseridas pelo usuário. Caso sejam inseridos para *a* e *b* os valores *2* e *3* respectivamente, o algoritmo irá exibir como saída o valor *5*. Caso sejam fornecidos os valores *4* e *8,* será exibido o valor *12*, e assim sucessivamente.

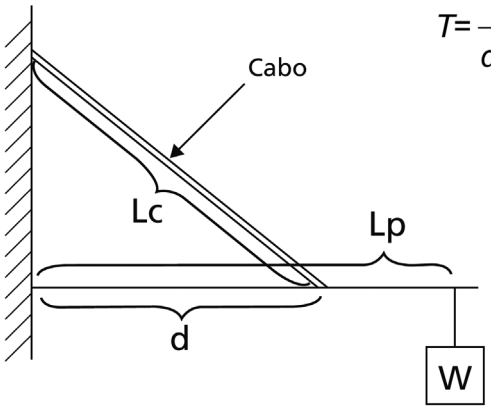
Atividade Final

Atende aos Objetivos 1, 2 e 3

1. Faça um algoritmo que leia um número real x e imprima o oposto dele, ou seja, –x**.**

2. A tensão em um cabo que suporta um peso é dada pela equação a seguir, onde T é tensão; w, o peso; d, Lp e Lc são distâncias. A figura seguinte mostra como posicionar as distâncias e o peso. Faça um algoritmo que leia todos os dados necessários e calcule a tensão no cabo. Suponha que os valores digitados serão sempre válidos, ou seja, nenhum valor negativo será passado, nem catetos maiores que hipotenusas, etc.

$$T = \frac{W \cdot Lp \cdot Lc}{d \cdot \sqrt{Lc^2 - d^2}}$$



Resposta Comentada

1. O modo mais simples de resolver esta questão é armazenar o valor lido em uma variável e fazer a inversão no próprio comando de impressão. Uma possível resposta para o problema é apresentada a seguir.

```
Algoritmo oposto( )
Início
        Inteiro x
        Imprimir "Digite um número"
        Ler x
        Imprimir "o oposto é" + (-x)
Fim
```

Note que há dois operadores na expressão que segue o comando imprimir: o operador de concatenação (+) e o operador de inversão de sinal (−). Como não foi definida a relação de precedência entre os operadores aritméticos e operadores de texto, foi utilizado o operador parênteses para indicar qual operação deve ser feita primeiro. Neste caso, a primeira a ser feita é a inversão de sinal, depois a concatenação. O resultado da concatenação é então impresso.

2. Apesar de a informação dada na segunda questão sobre valores negativos e a relação entre as distâncias ser óbvia, muitas vezes, o tratamento destas questões acaba por roubar o foco dos programadores. De fato, um programador que trabalha em produção de *software* passa mais tempo tentando impedir o usuário de fazer coisas erradas do que programando o núcleo do *software*. Como o objetivo desta disciplina é introduzir você no mundo dos algoritmos, as técnicas usadas para evitar estes erros não serão abordadas. Para as atividades desta disciplina, você pode assumir que o usuário dos algoritmos não fará nada que seja inválido.

No que tange à resolução do exercício, a única dificuldade que existe é transformar a equação da tensão da forma matemática, como ela foi apresentada, para uma forma computacional. Uma das possíveis soluções para este problema seria o seguinte algoritmo:

```
Algoritmo calculoTensao( )
Início
        Real w, lp,lc,d
        Imprimir "Digite as distâncias d, lc, lp e o peso w respectivamente"
        Ler d,lc,lp,w
        Real tensao = w*lp*lc/(d*(lc^2.0-d^2.0)^(1.0/2.0))
        Imprimir tensao
Fim
```

Note que a raiz quadrada da expressão é transformada em uma exponenciação por uma fração. Um fato importante a ser comentado é a presença dos números reais na expressão "1.0/2.0". Estes números são necessários porque, se utilizarmos números inteiros, como "1/2", estaremos, na verdade, utilizando o resultado do quociente da divisão de 1 por 2. Como já vimos antes, "1/2" tem quociente zero e resto 1, e o que queremos na expressão é utilizar "0.5" como expoente, logo o uso de números reais é necessário para obter o "0.5" exigido pela equação.

Resumo

Nesta aula, você aprendeu a sintaxe básica de algoritmos, inserir comentários e a lidar com a precedência de operações. Também aprendeu o conceito de variável, como declarar uma variável e como alterar seus valores. Por fim, aprendeu a utilizar os comandos de leitura e escrita, que possibilitam escrever algoritmos que interagem com o usuário.

Informação sobre a próxima aula

Na próxima aula, você verá como lidar com situações em que devemos executar trechos de algoritmos dependendo de certos resultados e situações. Para isso, serão introduzidas as estruturas de desvio de fluxo.

Aula 3

Desvio condicional: parte I

Tiago Araújo Neves

Meta

Expor os conceitos de desvio condicional e as estruturas usadas para esse fim.

Objetivos

Esperamos que, ao final desta aula, você seja capaz de:

1. identificar o conceito e o uso de variáveis e expressões lógicas simples;
2. fazer algoritmos que utilizem expressões lógicas simples desvios condicionais.

Introdução

Muitas vezes, necessitamos executar certas ações em função de algumas condições. Alguns exemplos do cotidiano: devemos lavar a louça, se há louça suja; levamos o guarda-chuva, se o dia está chuvoso; usamos casaco, se está frio; se temos alface e tomate, preparamos uma salada, senão, saímos para comprá-los.

Essa necessidade de executar ações na presença de certas condições também existe dentro da computação. Para analisar tais condições, são utilizadas estruturas condicionais, também chamadas de estruturas de desvio de fluxo. O uso destas estruturas é frequente e seu domínio indispensável para a construção de algoritmos. Nesta aula, você verá dois tipos de estrutura de desvio: a estrutura de desvio simples e a estrutura de desvio composto.

Variáveis e expressões lógicas

Uma *expressão lógica* nada mais é do que um conjunto de operações que pode resultar em dois possíveis valores: *verdadeiro* ou *falso*. Exemplos de expressões lógicas simples são expressões que usam os operadores relacionais como o operador > (maior que). A **Tabela 3.1** mostra os operadores relacionais e os operadores de igualdade como comumente vistos na Matemática, seus correspondentes em pseudocódigo, exemplos de uso e seus significados.

Tabela 3.1: Operadores matemáticos e seus correspondentes em pseudocódigo

Operador matemático	Operador em pseudocódigo	Exemplo	Significado
=	==	a==b	a é igual a b?
≠	!=	a!=b	a é diferente de b?
>	>	a>b	a é maior que b?
<	<	a<b	a é menor que b?
≥	>=	a>=b	a é maior ou igual a b?
≤	<=	a<=b	a é menor ou igual a b?

Um exemplo de avaliação de um operador relacional pode ser *a*<*b*, tendo a variável *a* o valor 1 e a variável *b* o valor 6. Neste caso, a expressão é traduzida para 1<6, que resulta em *verdadeiro*. Agora, se mantivermos os valores das variáveis e mudarmos o operador para >, temos a expres-

são *a>b*, que é traduzida para 1>6, que resulta em *falso*. Para simplificar a notação utilizada, usaremos *V* para denotar *verdadeiro* e *F* para *Falso*.

É importante mencionar que o resultado de operações de igualdade e relacionais podem ser armazenadas em variáveis do tipo lógico, isto é, ilustrado pelo algoritmo da **Figura 3.1**, que irá imprimir *V* ou *F*, dependendo dos valores digitados para as variáveis. É importante mencionar que variáveis do tipo lógico só podem assumir dois valores: *V* ou *F*.

```
Algoritmo variavelLogica()
Início
        Inteiro a,b
        Ler a,b
        Logico teste = a>b
        Imprimir teste
Fim
```

Figura 3.1: Utilizando variáveis lógicas para armazenar resultados de expressões relacionais.

Os operadores relacionais e de igualdade podem ser utilizados com expressões mais elaboradas como, por exemplo, 3+2>5−0. Neste caso, as expressões aritméticas são resolvidas primeiramente, e somente então são avaliados os operadores relacionais e de igualdade. Isto ocorre devido à precedência desses operadores.

Atividade 1

Atende ao Objetivo 1

Avalie o resultado das seguintes expressões como verdadeiro ou falso:

a) sendo a=2 e b=3, a expressão a%b!=b%a

b) sendo a=3 e b=1, a expressão a−b=−1*(b−a)

c) sendo a=1 e b=2, a expressão a+b!=b+a

Resposta Comentada

Analisaremos a primeira expressão. Como as expressões são resolvidas primeiro, e sempre da esquerda para a direita, 2%3 é resolvida primeiro. Como o resto da divisão de *a* por *b* é 2, então, temos 2!=3%2. Em seguida,

é resolvido 3%2, que resulta em 1. Assim, a expressão fica 2!=1. Como 2 é diferente de 1, a expressão resulta em verdadeiro.

De maneira análoga, a segunda expressão resulta em verdadeiro, e a terceira em falso.

Estrutura de desvio simples

A estrutura de desvio simples, também chamada *Estrutura Se*, é utilizada quando devemos executar certas ações, caso uma condição seja atendida. Esta estrutura tem a seguinte sintaxe:

Se (*Expressão Lógica*) Então
  Instruções
Fim Se

Figura 3.2: Sintaxe da *Estrutura Se*.

A **Figura 3.2** mostra a sintaxe da *Estrutura Se*. Como palavras reservadas da estrutura, temos: *Se, (,), Então e Fim Se.* A palavra *Se* indica o início da instrução condicional. O símbolo "(" marca o início da expressão lógica que expressará a condição que queremos avaliar. O símbolo ")" marca o fim da expressão lógica. A palavra *Então* marca o início das ações. Ela indica que, na próxima linha, estará a primeira das ações que devemos executar, caso a condição seja satisfeita. A expressão *Fim Se* marca o fim da estrutura de desvio. Note que entre o *Então* e o *Fim Se* existem *instruções.* É neste ponto que estarão expostas todas as ações que devem ser executadas, caso a *Expressão Lógica* seja satisfeita. O conjunto de instruções contido entre o *Então* e o *Fim Se* é executado somente se o resultado da *Expressão Lógica* for *verdadeiro.* Caso o resultado da *Expressão Lógica* seja *falso*, o conjunto de instruções entre o *Então* e o *Fim S*e é ignorado.

Imagine a execução de um algoritmo como um fluxo que segue de cima para baixo. A **Figura 3.3** mostra um exemplo deste fluxo. Nesta figura, existem três instruções genéricas. Como o fluxo de execução segue de cima para baixo, a primeira instrução a ser executada é a *instrução 1*; em seguida, a *instrução 2* e, por último, a *instrução 3*.

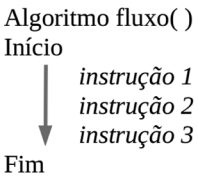


Figura 3.3: Percurso do fluxo de execução.

O uso de estruturas de condicionais permite desviar o fluxo de execução, e por isso estas estruturas também são chamadas de estruturas de desvio. A **Figura 3.4** mostra o desvio de fluxo de uma instrução, se quando o resultado da expressão lógica é *verdadeiro*. O fluxo inicialmente passa pelo conjunto de instruções que vem antes do *Se* e chega até a estrutura condicional. Neste ponto, a expressão lógica é avaliada; caso o resultado seja *verdadeiro*, o fluxo passa pelo conjunto de instruções dentro da estrutura condicional até chegar ao *Fim Se*. Por fim, o fluxo segue a partir do *Fim se*, passa pelas instruções posicionadas após a estrutura condicional até chegar ao fim do algoritmo. Note que o fluxo de programa passa por dentro da instrução *Se* – o que permite que o conjunto de instruções entre o *Então* e o *Fim Se* sejam executadas.

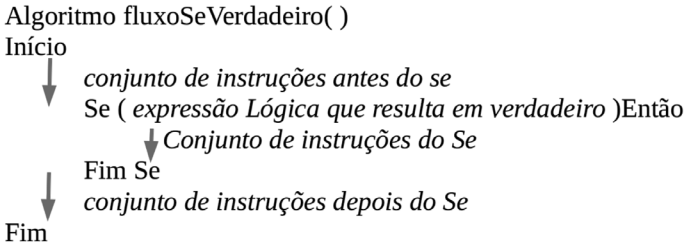


Figura 3.4: Fluxo de uma instrução *Se* com resultado da expressão lógica *verdadeira*.

A **Figura 3.5** mostra o fluxo de execução de uma estrutura *Se* quando o resultado da expressão lógica é *falso*. O fluxo inicialmente passa pelo conjunto de instruções que vem antes do *Se* e chega até a estrutura condicional. Neste ponto, a expressão lógica é avaliada, caso o resultado seja *falso,* o fluxo é desviado para o *Fim Se*. Por fim, o fluxo segue a partir do *Fim Se*, passa pelas instruções posicionadas após a estrutura condicional até chegar ao fim do algoritmo. Note que o fluxo é desviado do *Se* para o *Fim Se* e que as instruções entre o *Então* e o *Fim Se* não são executadas.

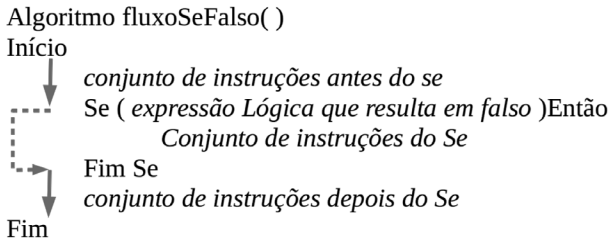


Figura 3.5: Fluxo de uma instrução *Se* com resultado da expressão lógica *falso*.

Atividade 2

Atende aos Objetivos 1 e 2

1. Faça um algoritmo que calcule o valor absoluto de um número. A equação do valor absoluto é descrita abaixo:

$$x = \left\{ \begin{array}{c} x, se x = 0 \\ -x, c.c. \end{array} \right\}$$

2. Faça um algoritmo que leia um número inteiro e verifique se tal número pertence ao conjunto dos números naturais.

3. Faça um algoritmo que verifique se um determinado número, informado pelo usuário, é múltiplo de 2.

4. Faça um algoritmo que verifique se um determinado número *x* é múltiplo de outro número *y*. Os valores de *x* e *y* devem ser informados pelo usuário.

Resposta Comentada

1. De acordo com a equação, o valor absoluto de um número *x*, também chamado módulo de *x*, é o próprio *x*, se *x* for maior ou igual a zero, ou $-x$, em caso contrário. Desse modo, devemos fazer um algoritmo que imprima o valor de *x* multiplicado por −1, caso *x* seja menor que zero. Um possível algoritmo para este problema é o seguinte:

```
Algoritmo modulo( )
Início
        Real x, res
        Imprimir "Digite um número"
        Ler x
        res = x
        Se (x < 0)Então
                res = -1*res
        Fim Se
        Imprimir "O módulo deste número é "+res
Fim
```

Nesse algoritmo, são usadas duas variáveis: *x*, para leitura do número, e *res*, para o resultado do valor absoluto. O primeiro passo é ler o número do qual se deve calcular o módulo. Em seguida, colocamos em *res* o valor do próprio número lido (res=x). O passo crítico deste algoritmo é a estrutura de desvio. Neste caso, se *x* for maior ou igual a zero, nada precisa ser feito, pois em *res* está o valor do próprio *x*, o que atende à equação. Caso *x* seja menor que zero, o valor contido em *res* é multiplicado por −1, e o resultado é armazenado na própria variável *res* (res=−1*res). Isso ocorre porque os operadores de inversão de sinal e multiplicação têm precedência maior que o de atribuição, logo o lado direito da expressão é avaliado e só depois a operação de atribuição é executada. Note que isto atende à definição da equação, uma vez que, se *x* é menor que zero

e *res* possui o mesmo valor de *x*, ao multiplicarmos *res* por −1, encontramos o valor absoluto de *x*. Como o resultado é armazenado em *res*, e *res* é justamente onde queremos armazenar o valor absoluto, então, o algoritmo atende à equação.

2. O conjunto dos números naturais está contido no conjunto dos números inteiros. Assim, um número natural necessariamente é inteiro, mas o contrário não é verdadeiro. Os números naturais são compostos de todos os números inteiros positivos mais o zero. Assim, para verificar se um número inteiro pertence ao conjunto dos números naturais, basta verificar se ele é maior ou igual a zero.

3 e 4. Como a atividade 3 é um caso particular da atividade 4, apenas uma solução para essas a atividades será apresentada. Para verificar se um número *x* é múltiplo de outro número *y*, deve-se verificar se a divisão de *x* por *y* é exata, ou seja, tem resto igual a zero. Para uma possível solução, primeiramente, são criadas duas variáveis com a mesma nomenclatura do problema. A seguir, estas variáveis são lidas e o resto da divisão entre elas é analisado. Caso o resto seja zero, isto implica que *x* é um múltiplo de *y*. Caso contrário, *x* não é múltiplo de *y*.

Estrutura de desvio composto

Em muitos casos, devemos tomar cursos de ação diferentes, de acordo com uma condição. Um exemplo do cotidiano é decidir como pagar o restaurante na hora do almoço. Se temos dinheiro suficiente, pagamos em dinheiro; senão, pagamos com cartão. Outro exemplo é decidir qual caminho tomar para ir ao trabalho. Se o caminho mais curto não está congestionado, utilizamos este; senão, utilizamos outro caminho.

Você conhece os programas que auxiliam o motorista na escolha do melhor caminho? Veja na reportagem a seguir:

Trânsito agora: chegue mais rápido com Google Maps, Waze, Copilot e Here

Por Lú Faveiro

> Aplicativos de GPS como Waze, Google Maps, Copilot e Here ajudam os motoristas a encontrarem os melhores caminhos para chegarem aos seus destinos. Com diversas funções, como orientação por voz e possibilidade de compartilhamento, os navegadores prometem descomplicar o trânsito.
>
> […]
>
> "GPS colaborativo" que também pertence ao Google, o Waze é definido como uma comunidade de mapeamento de trânsito em tempo real. Mais de 40 milhões de usuários alimentam o aplicativo, que sugere ao usuário a melhor opção de rota. O programa indica barreiras policiais, engarrafamentos e o que mais for compartilhado no caminho.
>
> Disponível em: <http://www.techtudo.com.br/dicas-e-tutoriais/noticia/2014/01/transito-agora-chegue-mais-rapido-com-google-maps-waze-copilot-e-here.html>. Acesso em: 30 jan. 2014.

A estrutura condicional composta, também chamada de estrutura *Se-Senão*, é utilizada para resolver problemas deste tipo, onde dependendo do resultado de uma expressão devemos fazer ações diferentes. A sintaxe da estrutura composta é mostrada na **Figura 3.6**.

```
Se ( Expressão Lógica ) Então
  Instruções do se
Senão
  Instruções do Senão
Fim Se
```

Figura 3.6: Sintaxe da estrutura *Se-Senão*.

Note que a estrutura *Se* e a estrutura *Se-Senão* têm muitos elementos em comum. A diferença é que depois das instruções do *Se*, existe a palavra reservada *Senão* e um segundo conjunto de instruções. O primeiro conjunto de instruções, mostrados na figura como "instruções do Se", fica localizado entre o *Então* e o *Senão*. O segundo conjunto, mostrado na figura como "instruções do Senão", fica localizado entre o *Senão* e o *Fim Se*. É importante mencionar que não há um "Fim Senão". Isso ocorre porque o *Se* e o *Senão* são duas partes de uma mesma estrutura. A primeira

parte da estrutura *Se-Senão* começa no *Se* e termina logo antes do *Senão*. A segunda parte começa no *Senão* e vai até o *Fim Se*.

A semântica da estrutura *Se-Senão* é um pouco mais elaborada que a da estrutura *Se*. Nesta estrutura, caso a expressão lógica resulte em *verdadeiro*, o conjunto de instruções do *Se* (entre o *Então* e o *Senão*) é executado e o conjunto de instruções do *Senão* (entre o *Senão* e o *Fim Se*) é ignorado. Caso a expressão lógica resulte em *falso*, o conjunto de instruções do *Se* é ignorado e o conjunto de instruções do *Senão* é executado. Note que, independente do resultado da expressão, sempre haverá um conjunto de instruções que é executado e outro que é ignorado. O resultado da expressão é que determina qual conjunto será executado e qual será ignorado.

A **Figura 3.7** ilustra o percurso do fluxo de execução de um algoritmo com uma estrutura *Se-Senão* quando o resultado da expressão lógica é *verdadeiro*.

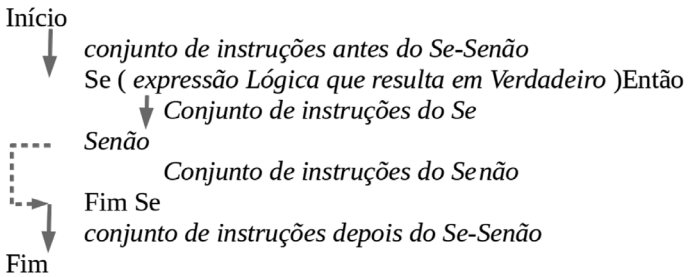


Figura 3.7: Fluxo do *Se-Senão* com condição *verdadeira*.

O fluxo segue do início, passa por todas as instruções antes do *Se-Senão* e chega à estrutura de desvio. Neste momento a expressão lógica é avaliada e, caso seu resultado seja *verdadeiro*, as instruções do *Se* (entre o *Então* e o *Senão*) são executadas. Ao chegar no *Senão* o fluxo é desviado para o *Fim Se*, ignorando as instruções do *Senão* (entre o *Senão* e o *Fim Se*). Após o *Fim Se*, o fluxo segue pelas instruções após o *Se-Senão* até o fim do algoritmo.

Já a **Figura 3.8** mostra o percurso do fluxo de execução de um algoritmo com uma estrutura *Se-Senão* quando o resultado da expressão lógica é *falso*.

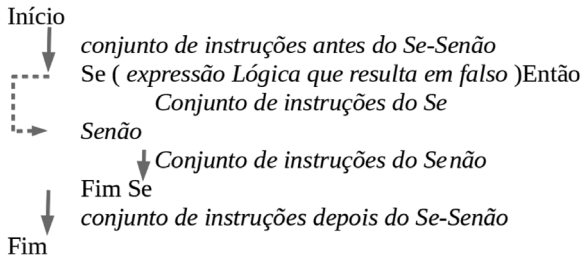


Figura 3.8: Fluxo do *Se-Senão* com condição *falsa*.

O fluxo segue do início, passa por todas as instruções antes do *Se-Senão* e chega à estrutura de desvio. Neste momento, a expressão lógica é avaliada e, caso seu resultado seja *falso*, o fluxo é desviado para o *Senão*, ignorando as instruções do *Se* (entre o *Então* e o *Senão*). Ao chegar ao *Senão*, o fluxo passa pelas instruções do *Senão* (entre o *Senão* e o *Fim Se*) até chegar ao *Fim Se*. Após o *Fim Se*, o fluxo segue pelas instruções após o *Se-Senão* até o fim do algoritmo.

Exemplo de uso: fazer um algoritmo que determine se uma equação do segundo grau possui ou não raízes reais.

Uma equação do segundo grau é comumente representada por $ax^2+bx+c=0$; possui raízes reais quando b^2-4ac é maior ou igual a zero. Assim, uma possível solução para o problema é a seguinte:

```
Algoritmo raizesEquacao()
Início
        Real a,b,c
        Imprimir "Digite os 3 coeficientes da equação"
        Ler a,b,c
        Se(b*b-4*a*c>=0) Então
                Imprimir "A equação possuí raízes reais"
        Senão
                Imprimir "A equação NÃO possuí raízes reais"
        Fim Se
Fim
```

Figura 3.9: Algoritmo para verificar se uma equação do segundo grau possui raízes reais.

Atividade 3

Atende aos Objetivos 1 e 2

1. Complete o exemplo anterior, inserindo as instruções necessárias para calcular quais as raízes da equação informada, caso a equação possua raízes reais.

2. Faça um algoritmo que determine se um determinado número informado pelo usuário é par ou ímpar.

3. Faça um algoritmo que leia a idade de um indivíduo e determine, de acordo com a legislação em vigor atualmente, se este indivíduo é maior ou menor de idade.

4. O Índice de Massa Corporal (IMC) é uma medida utilizada pelos profissionais de saúde para avaliar se um indivíduo está ou não acima do peso. Para calcular este índice, dividimos a massa do indivíduo, chamada de m (obtida em Kg por uma balança) pelo quadrado da sua altura, chamada de h (medida em metros), ou seja, $IMC=m/h^2$. Para os homens, caso o IMC seja maior do que 26,4, o indivíduo é considerado acima do peso. Para as mulheres, esse limiar é 25,8. Faça um algoritmo para calcular o IMC de pessoas do mesmo sexo que você. O algoritmo deve ler as informações necessárias (massa e altura), calcular o IMC e exibir uma mensagem informando se o indivíduo está ou não acima do peso ideal.

5. Em uma determinada fábrica de hastes de metal, uma haste é considerada dentro do padrão de qualidade se a diferença entre o tamanho da haste e um gabarito não ultrapassa um limiar. Este limiar varia de acordo com o tipo de material utilizado na fabricação da haste. Faça um algoritmo que, dados o comprimento da haste, o gabarito e o limiar, determina se a haste está ou não dentro do padrão de qualidade.

6. Faça um algoritmo que leia dois números reais e um número inteiro. O número inteiro é uma opção que representa uma operação a ser feita. Os números reais são os operandos da operação em questão. Caso a opção seja 1, o algoritmo deve calcular e imprimir a soma dos dois números reais. Caso 2, a subtração; caso 3, a divisão, e o resultado deve ser a multiplicação para a opção 4.

7. Faça um algoritmo que receba uma quantidade de horas trabalhadas em um mês por um funcionário, o valor a ser pago por hora e o número de horas previstas no contrato do funcionário. O algoritmo deve calcular quanto o funcionário irá receber. As horas extras (horas trabalhadas além do contrato) devem ter seu valor aumentado em 50%.

Resposta Comentada

1. Para determinar as raízes de uma equação do segundo grau, devemos utilizar a fórmula de Bhaskara, que diz que a primeira raiz, chamada x^1, é dada por

$x_1 = \frac{-b+\sqrt{\Delta}}{2a}$ e a segunda raiz, denominada x², é dada por $x_2 = \frac{-b+\sqrt{\Delta}}{2a}$, onde delta é dado por $\Delta = b^2 - 4ac$. Para fazer uma raiz quadrada, basta fazer uma potenciação fracionária, ou seja, $\sqrt{b} = b^{1/2}$. Lembre-se de que a potenciação tem precedência maior que a divisão, logo é necessário utilizar parênteses para exprimir um expoente fracionário, bem como é necessário o uso de constantes reais.

2. Para verificar se um número é par ou ímpar, basta verificar se ele é múltiplo de dois, utilizando o operador modulos. A mesma estrutura pode ser usada para a atividade 3, mudando apenas a condição e as mensagens a serem impressas.

4. O primeiro passo para resolver esta questão é perceber que os valores de massa e altura e do limiar de IMC são números reais. Assim, uma possível solução para a questão, considerando-se o sexo masculino, é dada a seguir. Uma solução para o sexo feminino é facilmente obtida trocando a mensagem inicial e substituindo o valor 26,4 para 25,8.

```
Algoritmo IMC()
Início
        Real massa,altura
        Imprimir "Digite a massa e a altura de um homem"
        Ler massa,altura
        Se(massa/altura^2 > 26.4 ) Então
                Imprimir "Acima do peso"
        Senão
                Imprimir "Não está acima do peso"
        Fim Se
Fim
```

5. Neste caso, o que temos que fazer é calcular o módulo da diferença entre os tamanhos (da haste e do gabarito) e verificar se esta diferença é menor que o limiar. Uma possível solução é dada a seguir.

```
Algoritmo padraoQualidade()
Início
        Real tam, gab, limiar
        Ler tam, gab, limiar
        real dif = tam – gab
        Se (dif <0) Então
                dif = -1 * dif
        Fim Se
        Se(dif <= limiar) Então
                Imprimir “A peça está de acordo com o padrão”
        Senão
                Imprimir “A peça está fora do padrão de qualidade”
        Fim Se
Fim
```

6. Uma possível solução para o problema é mostrada a seguir.

```
Algoritmo operacoes()
Início
        Real a,b
        Inteiro op
        Imprimir “Informe os operando e o código da operação”
        Ler a,b,op
        Se(op==1) Então
                Imprimir a + b
        Fim Se
        Se(op==2) Então
                Imprimir a - b
        Fim Se
        Se(op==3) Então
                Imprimir a / b
        Fim Se
        Se(op==4) Então
                Imprimir a * b
        Fim Se
Fim
```

7. Neste caso, é necessário calcular o volume de horas excedentes, que é dado pela subtração entre a quantidade de horas trabalhadas e a quantidade de horas estipuladas em contrato. Caso o volume de horas excedentes seja maior do que zero, isto significa que o funcionário fez horas extras, e ele deve ser remunerado a mais por estas horas. Para sabermos o valor extra, devemos multiplicar as horas extras pela metade do valor da hora e somar este resultado ao pagamento anterior do funcionário, obtendo assim um novo pagamento. Caso o volume de horas extras seja menor ou igual a zero, não devemos alterar o pagamento calculado. Uma possível solução é dada a seguir:

```
Algoritmo horaExtra()
Início
        Real valor
        Inteiro trab, contrato
        Imprimir “Digite o número de horas trabalhadas e as horas de contrato”
        Ler trab, contrato
        Imprimir “Digite o valor da hora”
        Ler valor
        real pagamento = trab*valor
        inteiro extra = trab - contrato
        Se(extra>0) Então
                pagamento = pagamento + extra*valor/2
        Fim Se
        Imprimir “O funcionário deve receber: ”+pagamento
Fim
```

Resumo

Nesta aula, você aprendeu a lidar com expressões lógicas (que retornam verdadeiro ou falso) simples e a utilizar estas expressões em estruturas de desvio condicional, simples e compostas. Também foi visto que, dependendo do resultado de uma expressão lógica, um determinado conjunto de instruções pode ser executado ou ignorado, dependendo da estrutura de desvio utilizada.

Informação sobre a próxima aula

Na próxima aula, você verá como lidar com expressões lógicas mais elaboradas e como construir estruturas de desvio encadeadas através de aninhamento.

Aula 4

Desvio condicional: parte II

Tiago Araújo Neves

Meta

Expor os conceitos avançados de desvio condicional.

Objetivos

Esperamos que, ao final desta aula, você seja capaz de:

1. reconhecer o conceito e o uso de expressões lógicas avançadas;
2. criar algoritmos capazes de fazer desvios condicionais que utilizem expressões como condição;
3. fazer algoritmos que utilizam aninhamento.

Expressões lógicas utilizando conjunção, disjunção e negação

Até o momento, os algoritmos propostos têm utilizado expressões simples nas estruturas de desvio. Contudo, em muitos problemas da vida real, são necessárias análises de múltiplos valores e múltiplas variáveis para se tomar uma decisão. Um exemplo de análise de múltiplos valores, que utilizamos no cotidiano, é a obrigatoriedade do voto no Brasil, em relação à idade. De acordo com nossa legislação atual, um indivíduo é obrigado a votar, caso tenha mais de 18 e menos de 70 anos.

Figura 4.1: As pessoas entre 16 e 18 e as com mais de 70 anos também podem votar, embora não sejam obrigadas a isso.

No caso das múltiplas variáveis, um exemplo é determinar se um aluno regularmente matriculado em um curso presencial está ou não aprovado. Tendo por base o regimento e o regulamento de cursos de graduação da Universidade Federal Fluminense, um aluno de curso presencial, para ser considerado aprovado, precisa ter nota maior ou igual a 6 e frequência maior ou igual a 75%.

Para montar expressões que lidam com múltiplos valores e múltiplas variáveis, utilizam-se as operações lógicas. Existem várias operações lógicas; entretanto, três são essenciais e estão presentes em qualquer linguagem de programação. São elas a conjunção, a disjunção e a negação.

Negação

Por ser a mais simples das três operações lógicas básicas, a Negação será apresentada primeiro. Esta operação, que tem como operador a exclamação (!) e utiliza a sintaxe *!operando*, simplesmente inverte o valor lógico de um operando. Assim, se o operando tem valor verdadeiro, o resultado da negação deste operando é falso. De maneira análoga, se o operando tem valor falso, o resultado da negação deste operando é verdadeiro. A **Tabela 4.1** mostra a tabela verdade da operação de negação para os possíveis valores de operandos e o resultado da negação destes operandos. Caso uma variável lógica chamada *a* deva ser negada, a expressão montada deve ter a forma *!a* e deve ser lida como *não a*.

Tabela 4.1: Tabela verdade para a operação de negação

Operando	!Operando
Verdadeiro	Falso
Falso	Verdadeiro

Uma possível analogia com operadores matemáticos convencionais seria relacionar a operação de negação com a operação de inversão de sinais. A operação de inversão transforma números positivos em números negativos e vice-versa. Se formos considerar os valores positivos como verdadeiros e os negativos como falsos, a operação de negação tem a mesma função que a inversão de sinais.

Note que estas operações são feitas para tipos diferentes. A inversão de sinal é feita para números e a negação para valores lógicos. De nenhuma maneira o operador *!* pode ser substituído pelo operador – em uma expressão lógica, assim como o operador – não pode ser substituído pelo operador *!* em uma expressão matemática. O objetivo da analogia é apenas fazer um paralelo entre um conhecimento que você está adquirindo agora com algo que já conhece.

Também é importante notar que a operação de negação é uma **operação unária.**

Operação unária

Possui apenas um operando. Uma operação unária muito conhecida é a inversão de sinal na matemática, que é feita através do operador "–".

Um exemplo de uso deste operador é a seguinte expressão: *!a* . Neste caso, o resultado da expressão depende do valor da variável *a*. Caso esta variável tenha o valor verdadeiro, o resultado da expressão *!a* é falso. Analogamente, se o resultado da variável for falso, a expressão *!a* resulta em verdadeiro. A expressão *!a* é lida como "não a" ou "a negado".

Conjunção

A conjunção, também chamada *operação e*, tem por finalidade unir operandos de modo que uma expressão resulte falso toda vez que um operando for falso. Esta operação é **binária**, tem como operador o *e* comercial (&) e sua resolução é sempre feita da esquerda para direita. A conjunção de dois operandos *a* e *b* pode ser feita de dois modos: 1) a & b (lê-se *a e b*); 2) b & a (lê-se *b e a*).

Operação binária

Utiliza dois operandos, ou seja, são necessários dois valores lógicos para se fazer uma conjunção. Várias operações conhecidas são binárias. As mais comuns são soma, subtração, multiplicação e divisão. Para realizar uma soma, por exemplo, é necessário ter dois valores para somar.

A **Tabela 4.2** mostra a tabela verdade da conjunção entre dois operandos. Observe que a conjunção só retorna verdadeiro quando os dois operandos são verdadeiros. Assim, caso um dos operandos seja falso, a conjunção sempre retorna falso.

Tabela 4.2: Tabela verdade da conjunção para dois operandos

Operando 1	Operando 2	Operando 1 & Operando 2
Falso	Falso	Falso
Falso	Verdadeiro	Falso
Verdadeiro	Falso	Falso
Verdadeiro	Verdadeiro	Verdadeiro

A ordem dos operandos não altera o resultado final da conjunção, embora possa afetar a velocidade com que esta operação encontra seu resultado. Suponha que se deva fazer a conjunção de três operandos lógicos, chamados *a, b* e *c* (note que um operando lógico pode ser uma variável lógica ou uma expressão que resulte em verdadeiro ou falso como, por exemplo, 3>1 ou x==0. Para fazer isso, monta-se a seguinte expressão: *a & b & c*. A avaliação de expressões lógicas, assim como as aritméticas, é feita da esquerda para a direita. Caso o valor do operando *a* seja *falso* e os valores de *b* e *c* sejam *verdadeiros*, o resultado da expressão anterior é

falso. Contudo, basta analisar o operando *a* para obter este resultado, logo os operandos *b* e *c* não são analisados, visto que, na conjunção, apenas um operando falso é necessário para determinar o resultado da expressão. Agora, se a expressão fosse *b & c & a*, os três operandos deveriam ser avaliados para determinar o resultado da expressão. Assim, primeiro seria avaliado o resultado de *b & c*. Como os dois são verdadeiros, o resultado é *verdadeiro*. Em seguida, seria avaliado *verdadeiro & a*, que resulta em *falso*. Logo, a ordem dos operandos não influencia no resultado da expressão, mas influencia no número de análises que devem ser feitas para se chegar ao resultado final.

Uma possível analogia com operações matemáticas convencionais seria relacionar a conjunção com a multiplicação. Considere falso como sendo zero e verdadeiro como sendo um. Assim, o resultado da multiplicação de dois valores só resultará em 1 (verdadeiro) se todos os operandos forem 1 (verdadeiros). Caso qualquer um dos operandos seja 0 (falso), o resultado da multiplicação será 0 (falso).

Novamente, cabe lembrar que as operações de conjunção e multiplicação não são equivalentes, logo não se pode substituir o * por um *&* em uma expressão aritmética e nem substituir o *&* por * em uma expressão lógica.

Disjunção

A disjunção tem por finalidade unir operandos de modo que, se um deles for verdadeiro, toda a expressão resulta em verdadeiro. Esta operação é binária, tem como operador a barra vertical (|), e sua resolução é sempre feita da esquerda para a direita. A disjunção de duas variáveis *a* e *b* pode ser feita de dois modos:

1) a | b (lê-se *a ou b*);

2) b | a (lê-se *b ou a*).

A **Tabela 4.3** mostra a tabela verdade da operação de disjunção entre dois operandos. Note que, se um dos operandos é verdadeiro, a disjunção retorna verdadeiro, logo o único modo de uma disjunção retornar falso é quando todos os operandos são falsos.

Tabela 4.3: Tabela verdade da disjunção para 2 operandos

Operando 1	Operando 2	Operando 1 | Operando 2
Falso	Falso	Falso
Falso	Verdadeiro	Verdadeiro
Verdadeiro	Falso	Verdadeiro
Verdadeiro	Verdadeiro	Verdadeiro

Assim como na conjunção, a ordem dos operandos não altera o resultado final da disjunção, embora possa afetar a velocidade com que esta operação encontra seu resultado. Caso algum valor verdadeiro seja encontrado entre os termos de uma disjunção, não é necessário avaliar os demais, uma vez que apenas um valor verdadeiro é suficiente para determinar o resultado da disjunção.

Embora algumas vezes sejam feitas analogias entre a disjunção e a operação soma da matemática convencional, é necessário ter cuidado. Para que esta analogia possa ser feita, é necessário estabelecer que o conjunto utilizado é o dos números naturais, o valor zero corresponde a falso, e qualquer coisa maior que zero corresponde a verdadeiro. Assim, podemos verificar que o único modo de a soma de dois operandos resultar em 0 (falso) é quando os dois operandos são 0 (falsos). Para qualquer outra combinação de valores, o resultado desta soma é maior do que zero (verdadeiro).

Observe que, apesar de poder estabelecer analogias entre as operações lógicas e as operações aritméticas, as três analogias são diferentes. A primeira é definida no conjunto dos números inteiros, exceto zero; a segunda é definida no conjunto {0,1} e a terceira é feita no conjunto dos números naturais. Na primeira, os números negativos são considerados falsos, e os positivos, verdadeiros. Na segunda, 0 é falso e 1 verdadeiro. Na terceira, 0 é falso e todos os valores maiores que 0 são verdadeiros. Estas diferenças entre os domínios só evidencia que as operações não são equivalentes. Logo, devem-se aplicar operações lógicas em contextos lógicos e operações aritméticas em contextos matemáticos. Embora algumas

vezes utilizemos operadores lógicos e aritméticos juntos, é importante ressaltar que os tipos de dados com que cada operação trabalha é diferente, assim como a precedência destas operações. Por exemplo: vamos avaliar a expressão 3+4>6 & 2*2!=1.

A resolução da expressão é feita da esquerda para a direita, porém a precedência dos operadores também deve ser observada. Operadores aritméticos têm precedência maior que operadores relacionais, que têm precedência maior que os operadores de igualdade, que, por sua vez, têm precedência maior que operadores lógicos. A exceção é o operador unário de negação (!), que tem a segunda maior precedência vista até o momento. Assim, as expressões aritméticas são avaliadas primeiramente. Logo, a expressão anterior resulta em *3+4>6 & 4!=1* e, na próxima avaliação, em *7>6 & 4!=1*. Em seguida, são resolvidas as expressões relacionais; logo, a expressão resulta em *Verdadeiro & 4!=1*. Quando as operações de igualdade são resolvidas, a expressão resulta em *Verdadeiro & Verdadeiro*. Por último, a operação lógica é efetuada. Como os dois operandos são verdadeiros, o resultado da expressão é verdadeiro. É importante mencionar que assim como nas expressões aritméticas, os parênteses podem ser utilizados para mudar a ordem em que as operações são executadas.

A **Tabela 4.5** apresenta todos os operadores estudados até o momento, seus tipos e sua precedência relativa. Operadores na mesma linha possuem a mesma precedência e operadores com precedência maior são executados primeiro. Note que o operador de atribuição é o operador que tem a menor precedência e, devido a isso, todas as expressões são avaliadas antes de se fazer uma atribuição.

Tabela 4.5: Precedência dos operadores vistos até o momento

Operador	Tipo	Precedência
^	Binário	Mais alta
–,!	Unário	.
*,/,%	Binário	.
+,–	Binário	.
>,>=,<,<=	Binário	.
==,!=	Binário	.
&	Binário	.
|	Binário	.
=	Binário	Mais baixa

Atividade 1

Atende aos Objetivos 1 e 2

1. Avalie o resultado das seguintes expressões:

a) a–2>0 & b → sendo a=6 e b=Falso

b) a & b–3!=0 | Falso → para a=Verdadeiro e b=0

c) a>3 & !(3>4) & Verdadeiro → sendo a=5

2. Faça um algoritmo que lê três números e ainda verifica se estes números representam os lados de um triângulo. Para saber se esses três números representam um triângulo, use a propriedade que a soma de quaisquer dois lados de um triângulo é sempre maior que o terceiro.

3. Faça um algoritmo que receba as notas de três provas e a quantidade de faltas de um aluno. O algoritmo também deve ler o número total de aulas ministradas. O algoritmo deve calcular a média e a frequência do aluno em porcentagem. Caso a média seja inferior a 6 ou a frequência menor que 75%, o algoritmo deve imprimir que o aluno está reprovado. Caso contrário, deve imprimir que o aluno está aprovado.

Resposta Comentada

1. a) b tem o valor falso e a conjunção de falso com qualquer expressão resulta em falso.

b) como b vale zero, a expressão aritmética *b−3* resulta em *−3*, que é diferente de zero; como *a* é verdadeiro, logo a expressão é avaliada como *Verdadeiro & Verdadeiro | Falso*. Como a conjunção tem precedência maior, a expressão se reduz para *Verdadeiro | Falso*, que resulta em verdadeiro.

c) sendo 5 o valor de *a*, a expressão é avaliada como *Verdadeiro &!(Falso) & Verdadeiro*. A negação tem maior precedência, logo a expressão fica *Verdadeiro & Verdadeiro & Verdadeiro,* que resulta em verdadeiro.

2. Sendo *a*, *b* e *c* os lados do triângulo, devemos analisar as condições *a+b >c*, *a+c>b* e *b+c>a*. Como a propriedade diz que a soma de quaisquer dois lados tem que ser maior que o terceiro, as três condições devem ser verdadeiras. Assim, devemos usar conjunções para unir as três condições. Uma possível solução é dada a seguir.

```
Algoritmo triangulo()
Início
        Real a,b,c
        Ler a,b,c
        Se ( a+b>c & a+c>b & b+c>a ) Então
                Imprimir "Os três números formam um triângulo"
        Senão
                Imprimir  "Os números não formam um triângulo"
        Fim Se
Fim
```

3. A primeira coisa a ser feita nesta questão é ler as 5 informações definidas pelo enunciado. Uma vez feita a leitura, calcula-se a média e a porcentagem de frequência usando a regra de três. Feito isso, deve ser montada uma expressão lógica contendo as regras de aprovação. Uma possível solução é mostrada a seguir.

```
Algoritmo notaFrequencia()
Início
  Real n1,n2,n3
  Inteiro presenca, aulas
  Imprimir "Digite as três notas, quantas aulas foram ministradas e em quantas o aluno veio"
  Ler n1,n2,n3,aulas,presenca
  Real media = (n1+n2+n3)/3.0
  Real frequencia = (presenca*100.0)/aulas
  Se(media>=6.0 & frequencia >= 75.0)Então
     Imprimir "Aprovado"
   Senão
     Imprimir "Reprovado"
  Fim Se
Fim
```

Aninhamento

Em muitos casos, é necessário analisar múltiplas condições de maneira encadeada. Por exemplo: para verificar se um triângulo é equilátero (três lados iguais), escaleno (três lados diferentes) ou isósceles (quaisquer dois lados iguais), devemos analisar a relação entre os três lados. Caso os três lados sejam iguais, o triângulo é equilátero; porém, caso isto não ocorra, devemos fazer uma segunda avaliação condicional para determinar se o triângulo é escaleno ou isósceles. Esta segunda análise requer uma segunda estrutura condicional, que deve ser feita somente no caso em que a primeira falhe. Ou seja, é necessária uma estrutura *Se* dentro do *Senão* da primeira. Esta construção pode ser verificada na **Figura 4.6**.

```
Algoritmo aninhamentoSe()
Início
        Real a,b,c
        Imprimir "Digite os 3 lados de um triângulo"
        Ler a,b,c
        Se(a==b & b==c) Então
                Imprimir "Equilátero"
        Senão
                Se(a!=b & b!=c & a!=c) Então
                        Imprimir "Escaleno"
                Senão
                        Imprimir "Isósceles"
                Fim Se
        Fim Se
Fim
```

Figura 4.2: Estruturas *Se* aninhadas para classificar um triângulo em relação aos lados.

Este tipo de construção, com uma estrutura dentro de outra, é chamado de aninhamento e pode ser usado dentro de diversos tipos de estrutura, como veremos mais adiante. Para compor o aninhamento corretamente, é necessário colocar uma estrutura dentro da outra, o que significa que, tendo duas estruturas condicionais, uma interna e uma externa, é necessário que todos os elementos da estrutura interna estejam dentro da estrutura externa.

Observe a **Figura 4.3**. O algoritmo é o mesmo da **Figura 4.2**, porém com as linhas numeradas e as instruções da estrutura aninhada marcadas com um *. Note que todos os * (linhas 9 a 13) estão dentro do *Senão* que começa na linha 8 e termina na linha 14. Isto significa que o *Se* aninhado começa na linha 9 e termina na linha 13. Logo, a instrução *Fim Se* da linha 13 pertence à estrutura *Se-Senão* que começa na linha 9. Consequentemente, o *Fim Se* da linha 14 pertence à estrutura *Se-Senão* que começa na linha 6.

```
01 Algoritmo aninhamentoSe()
02 Início
03      Real a,b,c
04      Imprimir "Digite os 3 lados de um triângulo"
05      Ler a,b,c
06      Se(a==b & b==c) Então
07              Imprimir "Equilátero"
08      Senão
09              *Se(a!=b & b!=c & a!=c) Então
10              *       Imprimir "Escaleno"
11              *Senão
12              *       Imprimir "Isósceles"
13              *Fim Se
14      Fim Se
15 Fim
```

Figura 4.3: Aninhamento destacado.

Note que, com o uso de aninhamento, uma estrutura fica completamente contida dentro de outra. Assim, se encontramos um *Fim Se*, este certamente pertence à estrutura de desvio (*Se* ou *Se-Senão)* mais interna, que ainda está aberta.

Atividade 2

Atende aos Objetivos 1, 2 e 3

1. Faça um algoritmo que, dados três ângulos de um triângulo, classifique este triângulo como acutângulo, retângulo ou obtusângulo, e imprima o tipo correspondente.

2. Faça um algoritmo para dizer se uma equação do segundo grau expressa como ax^2+bx+c possui ou não raízes reais. Caso o valor de a na equação seja zero, o algoritmo deve imprimir "a equação passada não é do segundo grau".

Caso a equação possua raízes reais, o algoritmo deve imprimir "não possui raízes reais" e, caso a equação não possua raízes, ele deve imprimir "possui raízes reais ".

3. Faça um algoritmo que leia três informações de um indivíduo: sexo, massa e altura. O algoritmo deve calcular o Índice de Massa Corporal (IMC) e verificar e imprimir se o indivíduo está abaixo do peso, no peso ou acima do peso. Para calcular o IMC, dividimos a massa do indivíduo em Kg pelo quadrado da sua altura em metros, ou seja, $IMC=m/h^2$. Para os homens, caso o IMC esteja entre 20,7 e 26,4 o indivíduo está dentro do peso. Caso o IMC esteja na faixa abaixo deste intervalo, ele está abaixo do peso; se

estiver acima deste intervalo, ele está acima do peso. Para as mulheres, esse intervalo correspondente é 19,1 a 25,8.

4. Faça um algoritmo que receba a idade de uma pessoa e diga se ela não pode votar, pode votar ou é obrigada a votar, de acordo com a legislação eleitoral brasileira. Pela legislação atual, um indivíduo é obrigado a votar se tiver entre 18 e 70 anos. Ele pode votar se tiver mais de 70 e se estiver entre 16 e 18. Com menos de 16, um indivíduo não pode votar.

Resposta Comentada

1. Para verificar se um triângulo é retângulo, basta verificar se um dos ângulos é igual a 90 graus. Para verificar se ele é acutângulo, todos os ângulos devem ser menores do que 90 graus. Para verificar se ele é obtusângulo, basta verificar se um dos ângulos é maior do que 90.

2. Para resolver esta questão, o único cuidado especial a ser tomado é o aninhamento das instruções *Se*. O cálculo do valor de delta e as impressões de "tem raízes" e "não tem raízes" devem ser executados somente quando *a* for diferente de zero.

3. Uma possível solução é mostrada a seguir. Note que ao invés de avaliar todos os quesitos em uma mesma instrução de desvio, foi feita uma divisão das avaliações em duas instruções de desvio. A primeira determina quais os limiares de IMC a serem utilizados em função do sexo do indivíduo. A segunda faz a avaliação do IMC com os limiares definidos na primeira.

```
Algoritmo indiceMassaCorporal()
Início
   Real altura,massa, limiar1,limiar2
   Texto sexo
   Imprimir "Digite a altura a massa e o sexo do indivíduo"
   Ler altura,massa,sexo
   Real imc = massa/(altura^2)
   Se (sexo == "feminino") Então
      limiar1 = 19.1
      limiar2 = 25.8
   Senão
      limiar1 = 20.7
      limiar2 = 26.4
   Fim Se
   Se ( imc >= limiar1 & imc <= limiar2) Então
      Imprimir "Dentro do peso"
   Senão
      Se (imc < limiar1 ) Então
         Imprimir "Abaixo do peso"
      Senão
         Imprimir "Acima do peso"
      Fim Se
   Fim Se
Fim
```

4. Há vários modos de montar as condições para resolver esta questão. Uma possível solução é mostrada a seguir.

```
Algoritmo IdadeParaVotar()
Início
        Inteiro idade
        Ler idade
        Se ( (idade>=16 & idade <18 ) | idade >70 ) Então
                Imprimir “Pode Votar”
        Senão
                Se (idade>=18 & idade <=70) Então
                        Imprimir “Obrigado a votar”
                Senão
                        Imprimir “Não pode Votar”
                Fim Se
        Fim Se
Fim
```

Resumo

Nesta aula, você aprendeu a montar e interpretar expressões lógicas utilizando conjunção, disjunção e negação. Viu que essas operações podem ser utilizadas para expressar condições referentes a múltiplos valores e a múltiplas variáveis. Também foi mostrado como fazer o aninhamento de expressões condicionais e que este aninhamento é utilizado quando devemos analisar uma condição dependendo do resultado de outra condição.

Informação sobre a próxima aula

Na próxima aula, você verá como lidar com situações em que devemos repetir tarefas muitas vezes sem que isso implique replicação de trechos de um algoritmo.

Aula 5

Repetição: parte I

Tiago Araújo Neves

Meta

Expor os conceitos de repetição e uma das estruturas usadas para esse fim.

Objetivos

Esperamos que, ao final desta aula, você seja capaz de:

1. compreender o conceito de repetição e fazer uso de estruturas de repetição com contador;
2. fazer algoritmos que combinam estruturas de repetição e de desvio para resolver problemas.

Introdução

Em muitos casos da vida real, precisamos repetir várias vezes uma mesma tarefa. Como exemplo prático, suponha que você quer comprar um carro e já escolheu o modelo, cor, potência etc. O único ponto que falta é decidir em qual concessionária comprar. Para isso, você dispõe de uma lista contendo os telefones de todas as concessionárias que possuem o modelo que você deseja. Você deve, então, ligar para cada uma delas e perguntar preço e prazo de entrega. Ao final de cada ligação, anotará as respostas de cada concessionária e só depois de ligar para todas as lojas da lista, decidirá onde comprar o carro.

Outro exemplo corriqueiro seria o de um professor lançando as notas dos alunos no sistema da universidade. Para cada aluno, o professor deve conferir o número de matrícula e lançar a nota. Se existirem sessenta alunos, serão sessenta matrículas para conferir e sessenta notas para lançar.

Um modo ingênuo de fazer repetição de tarefas em algoritmos seria a replicação de trechos de código. Imagine que você tem uma tarefa a ser repetida e que o trecho que faz essa tarefa possui oito linhas. Se a tarefa precisa ser repetida sete vezes, no final, você tem 56 linhas no seu algoritmo. Esse tipo de abordagem é pouco eficiente, visto que aumenta muito o tamanho dos algoritmos e dificulta a sua legibilidade. Além disso, essa abordagem mostra-se ineficaz, quando não é possível determinar o número de vezes que a tarefa precisa ser repetida. Exemplos: um sistema de *login* de fórum eletrônico não pode fazer nenhuma suposição sobre quantos usuários vão se logar em um dia. Um sistema que faz simulações de financiamento bancário não pode supor quantos usuários farão simulação nem quantas simulações cada usuário fará. Um *player* de músicas não pode supor quantas músicas o usuário vai querer ouvir, dentre outros.

Para tratar desses problemas relacionados à repetição de tarefas, a computação usa estruturas de repetição, que evitam a replicação do código, tornando-o mais legível e elegante, e que realizam as tarefas, sendo o número de repetições conhecido ou não. Nesta aula, será apresentada a estrutura de repetição **Para**, também chamada de estrutura de repetição com contador, utilizada quando o número de repetições é conhecido.

Estrutura de repetição com contador

A primeira estrutura de repetição que será mostrada é a estrutura Para, conforme apresenta-se na **Figura 5.1**:

```
Para (Inicialização; Condição; Passo)Faça
        Instruções a serem repetidas
Fim Para
```

Figura 5.1: Sintaxe da estrutura Para.

Tal estrutura começa com a palavra reservada **Para** e termina com a palavra reservada **Fim Para**. Após o **Para**, entre os parênteses, estão a inicialização, condição e passo da contagem. A inicialização diz a partir de qual valor a contagem deve começar, a condição estabelece até quando a contagem deve continuar e o passo determina como a contagem é feita. Após os parênteses, está a palavra reservada **Faça**. Nessa estrutura, as instruções a serem repetidas estão entre o **Faça** e o **Fim Para**.

A semântica da estrutura **Para** pode ser descrita pelos seguintes passos:

1. A primeira coisa a ser feita nessa estrutura é a inicialização, que é realizada uma única vez no começo da repetição.
2. Após a inicialização, é feita a avaliação da condição. Se o resultado da análise for **Verdadeiro**, as instruções do **Para** (entre o **Faça** e o **Fim Para**) são executadas. Se não, o fluxo é desviado diretamente para o **Fim Para** e a repetição termina.
3. O passo da contagem é feito, em seguida, retorna-se ao passo 2.

Algumas observações importantes:

Iteração

Execução completa de um ciclo do conjunto de instruções de uma estrutura de repetição. Na estrutura **Para**, uma iteração compreende a avaliação da condição, a execução das instruções do corpo da estrutura e o passo.

Cada execução dos passos 2 e 3 da estrutura é chamada de **iteração**.

- Apesar de estar dentro da estrutura de repetição, a inicialização não é repetida; o que é repetido são a condição, as instruções entre o **Faça** e o **Fim Para** e o passo.
- Apesar de estar descrito no começo da estrutura, o passo é a última coisa a ser feita na repetição.

```
Algoritmo testePara()
Início
        Inteiro i
        Para (i=1;i<=10;i=i+1)Faça
                Imprimir i
        Fim Para
Fim
```

Figura 5.2: Exemplo de uso da estrutura Para.

A **Figura 5.2** mostra um exemplo de uso da estrutura **Para**. Nele a variável *i* é inicializada com *1*. Em seguida, é feita a comparação do valor de *i* com a constante *10*. Como *i* no início tem valor *1*, a expressão *i<=10* retorna **Verdadeiro**.

O próximo passo é executar as instruções do **Para**. Nesse caso, só há uma instrução, que é a de imprimir o valor de *i*. Após essa impressão, é feito o passo da contagem: *i* recebe o seu próprio valor acrescido de uma unidade, ou seja, *i* passa a valer *2*. Novamente, é feita a comparação do valor de *i* com a constante *10*. Como *i* vale *2*, a comparação *i<=10* novamente retorna **Verdadeiro**.

Em seguida, deve-se imprimir o valor de *i*, ou seja, imprimir *2*. O passo faz com que o valor do *i* seja modificado para *3* e, assim, o algoritmo prossegue até que *i* assuma o valor *10*. Nesse momento, a comparação *i<=10* retorna **Verdadeiro**, pois *10<=10*. O valor *10* é impresso e o passo é executado, fazendo com que *i* assuma o valor *11*. A comparação *i<=10* falha nesse momento, e o fluxo do algoritmo é desviado para o **Fim Para**, terminando a repetição.

Como não há nenhuma instrução após a repetição, o algoritmo termina sua execução. Logo, esse algoritmo imprime os números de 1 a 10, um em cada linha. Note que, se a mesma tarefa fosse feita através da replicação de código, seriam necessários 10 comandos "imprimir". Com o uso de uma estrutura de repetição, apenas três linhas foram necessárias.

É importante mencionar que a combinação entre inicialização, condição e passo é que determina quantas vezes as tarefas serão repetidas. Exemplo: suponha que uma determinada tarefa deva ser realizada dez vezes. O modo mais simples de se fazer isso é contar de 1 em 1 (passo 1), começando em 1 (inicialização 1) e indo até 10 (condição <=10). Porém, podemos realizar a mesma tarefa 10 vezes contando de inúmeras formas. Podemos começar em 10 (inicialização), contar até 1 (condição >=1), contando de −1 em −1 (passo −1). Podemos contar de 2 até 20 contando

de 2 em 2, de 10 até 100 contando de 10 em 10 e, assim, sucessivamente. Logo, o que determina a quantidade de vezes que uma tarefa é repetida não é a inicialização, nem a condição nem o passo, mas sim o contexto formado por esses três argumentos.

```
Algoritmo loop()
Início
        Inteiro i
        Para (i=1;i<=10;i=i-1)Faça
                Imprimir i
        Fim Para
Fim
```

Figura 5.3: Algoritmo com repetição em *loop*.

A **Figura 5.3** mostra um mau exemplo do uso de uma estrutura de repetição. Note que a contagem começa em *1* e prossegue enquanto a variável de contagem for menor do que *10*. O problema é o passo da estrutura. Observe que, a cada iteração, a variável de contagem é decrescida de uma unidade. Isso faz com que a variável seja sempre menor que *10*, independentemente do número de iterações da estrutura. Ou seja, na primeira iteração, será impresso o valor *1*; na segunda, *0*; na terceira, *–1*, e, assim, indefinidamente.

Como o conjunto de números inteiros é infinito, a estrutura de repetição nunca termina, o que, no jargão técnico da informática, é chamado de *loop* infinito ou, em alguns casos, simplesmente *loop*. O uso de estruturas de repetição que executam indefinidamente acabam por gerar algoritmos mal formados, uma vez que, por definição, um algoritmo deve ser um conjunto finito de passos para resolver um problema.

Atividade 1

Atende ao Objetivo 1

Nesta atividade, você colocará em prática o que estudamos de algoritmos até aqui:

a) Faça um algoritmo que imprima os números de 10 a 1 em ordem decrescente.

b) Faça um algoritmo que calcule a soma de 10 números reais digitados pelo usuário.

c) Faça um algoritmo que calcule a média aritmética de 10 números digitados pelo usuário.

d) Faça um algoritmo para calcular x^n em que x é um número real e n um número inteiro. Ambos os números devem ser informados pelo usuário.
Obs.: não use o operador ^ .

e) Faça um algoritmo que calcule o fatorial de um número qualquer digitado pelo usuário.

Resposta Comentada

a) O modo mais simples de resolver esta questão é fazer um algoritmo similar ao da **Figura 5.2**, alterando o controle da repetição, para que ele faça a contagem regressivamente. A solução é mostrada a seguir:

```
Algoritmo contagemRegressiva()
Início
        Inteiro i
        Para (i=10;i>=1;i=i-1)Faça
                Imprimir i
        Fim Para
Fim
```

b) Para calcular uma soma de vários números, pode-se utilizar a propriedade de que a soma total pode ser feita por partes. Exemplo: suponha a soma de quatro números 2+4+5+9. Essa operação pode ser resolvida da seguinte maneira: (2+4)+5+9, que, por sua vez, é igual a ((2+4)+5)+9. Ou seja, para resolver a soma do conjunto, basta resolver uma soma de cada vez, usando como operadores um elemento e o resultado da soma dos elementos anteriores.

O problema com essa estratégia é a primeira soma. Por definição, a soma é uma operação binária, ou seja, precisa de dois operadores. Logo, é necessário saber dois números para executar a primeira soma. Múltiplas estratégias podem ser utilizadas para resolver esta questão. A mais simples é utilizar a propriedade matemática do elemento neutro da soma. Ou seja:

2+4+5+9=0+2+4+5+9. Isso faz com que uma soma já possa ser calculada ao receber o primeiro número do conjunto.

Um algoritmo que usa a estratégia de solução proposta é mostrado a seguir. Neste algoritmo, a variável soma é utilizada para acumular os valores das somas das parcelas. Note que essa variável é inicializada com zero para que a soma possa ser feita de maneira bem-sucedida. A variável *x* é utilizada para ler o número que vai ser somado, e a variável *cont* utilizada para contar quantos números foram utilizados.

```
Algoritmo somaNum()
Início
        Inteiro cont
        Real soma=0,x
        Para (cont=0;cont<10;cont=cont+1)Faça
                Ler x
                soma=soma+x
        Fim Para
        Imprimir soma
Fim
```

c) Para resolver a questão, utilizamos um algoritmo muito similar ao da solução do subitem b. Para fazer uma média, devemos calcular uma série de somas e depois fazer uma divisão. A seguir, é apresentada uma possível solução. A diferença entre este algoritmo e o do subitem b é a divisão feita para calcular a média.

```
Algoritmo media()
Início
        Inteiro cont
        Real soma=0,x
        Para (cont=0;cont<10;cont=cont+1)Faça
                Ler x
                soma=soma+x
        Fim Para
        Imprimir "a media é: " + (soma/10)
Fim
```

d) A estratégia é basicamente a mesma do subitem b, tomando os devidos cuidados para que o elemento neutro utilizado seja o da multiplicação. Uma possível resposta é dada a seguir.

Note que a variável acumuladora é *result* e tem o valor inicial configurado em 1 (que é o elemento neutro da multiplicação). Para calcular a potência, devemos fazer um série de multiplicações de um mesmo número. A potência 5^3, por exemplo, pode ser descrita como *5*5*5*. Observe que a parcela *x* da

multiplicação é sempre a mesma. Isso ocorre porque *x* representa a base da potenciação. O resultado das sucessivas multiplicações é armazenado na variável *result*. Na primeira iteração, *result* vale *1* e seu valor será modificado para o resultado da multiplicação de *result*x=1*5=5*. Na segunda iteração, *result* já vale *5* e seu valor será alterado para *result*x=5*5=25*. Na terceira iteração, *result* vale 25 e seu valor será alterado para *result*x=25*5=125*.

```
Algoritmo potencia()
Início
        Inteiro cont, n
        Real result=1, x
        Imprimir "digite uma base e um expoente"
        Ler x,n
        Para (cont=1;cont<=n;cont=cont+1)Faça
                result = result*x
        Fim Para
        Imprimir "o resultado é: "+result
Fim
```

e) Para calcular o fatorial de um número *num*, devemos fazer as multiplicações *num*(num-1)*...*2*1=1*2*…*num*. Portanto, tanto faz multiplicar na ordem normal ou na ordem inversa. A solução apresentada calcula o resultado da multiplicação na ordem normal. O primeiro elemento (*1*) já está armazenado em *result*, logo, só é necessário fazer as multiplicações de 2 até *num*. Por isso, a variável de contagem *cont* é inicializada em *2* e finalizada com num na repetição. Como a variável *cont* é inicializada com *2*, finalizada com *num* e incrementada de 1 em 1, ela assumirá todos os valores necessários para o cálculo do fatorial. Ou seja, na primeira iteração, ela assumirá *2*, na segunda *3*, na terceira *4* e assim sucessivamente até *num*. Logo, é possível utilizá-la como termo da multiplicação no cálculo do fatorial.

```
Algoritmo fatorial()
Início
        Inteiro cont, result=1, num
        Imprimir "digite um número para calcular o fatorial"
        Ler num
        Para (cont=2;cont<=num;cont=cont+1)Faça
                result = result*cont
        Fim Para
        Imprimir "o fatorial é: "+result
Fim
```

Para entender melhor este exemplo, acompanhe a tabela a seguir. Nela, estão expostos os valores de cada variável no final de cada iteração para um exemplo em que o usuário informa querer saber o fatorial de 5:

Variável	Antes da repetição	Iteração 1	Iteração 2	Iteração 3	Iteração 4
num	5	5	5	5	5
cont	2	3	4	5	6
result	1	2	6	24	120

Note que a variável *num*, que contém o valor *5*, permanece constante. A variável *cont,* que é controlada pela estrutura de repetição, aumenta seu valor de *1* em *1*. A variável *result* utiliza o valor da variável *cont* para multiplicar o próprio valor, fazendo com que contenha o valor do fatorial. Assim, durante a iteração *2*, por exemplo, *result* receberá o próprio valor, que é o mesmo do final da iteração *1*(*2*) multiplicado pelo valor de *cont,* que também é o mesmo do final da iteração *1*(*3*). Logo, no final da iteração *2*, *result* receberá o resultado da multiplicação *2*3,* que resulta em *6*.

Após essa multiplicação, o passo da repetição é executado e o valor de *cont* é alterado para *4*. Do mesmo modo, na iteração *3*, *result* receberá o resultado da multiplicação *6*4* e, assim, sucessivamente. O algoritmo para quando *cont* atinge *6*, uma vez que *num* tem valor *5,* e *6* não é menor ou igual a *5*.

Combinando estruturas de repetição, desvio e declaração de variáveis

Como dito anteriormente, dentro do corpo de estruturas de repetição, podem existir múltiplas instruções. Alguns casos especiais merecem destaque, como o uso de desvios, repetições e declaração de variáveis dentro de repetições. Observe o exemplo da **Figura 5.4**, que expõe dois algoritmos:

(a)

```
Algoritmo criarVariavelRepticao1()
Início
        Inteiro i
        Inteiro j=1
        Para (i=1;i<=10;i=i+1)Faça
                Imprimir j
                j=j+1
        Fim Para
Fim
```

(b)

```
Algoritmo criarVariavelRepticao2()
Início
        Inteiro i

        Para (i=1;i<=10;i=i+1)Faça
                Inteiro j=1
                Imprimir j
                j=j+1
        Fim Para
Fim
```

Figura 5.4: Dois algoritmos: (a) equivale à Figura 5.2 e (b) faz criação de variável dentro da repetição.

Os dois algoritmos estão corretos. O da **Figura 5.4(a)** é equivalente ao algoritmo da **Figura 5.2**, ou seja, é um algoritmo que imprime números de *1* a *10*.

Já o algoritmo da **Figura 5.4(b)** imprime dez linhas com o mesmo valor, ou seja, é um algoritmo que imprime dez vezes no número *1*. Isso ocorre porque a variável *j* é criada e inicializada dentro da repetição, fazendo com que a cada iteração uma nova variável *j* seja criada. Isso é perfeitamente possível, visto que os dois algoritmos estão sintaticamente certos, porém deve-se observar que a criação de variáveis dentro das repetições faz com que valores armazenados dentro dessas variáveis sejam perdidos.

Logo, decidir se devemos criar ou não uma variável dentro de uma estrutura de repetição depende do tipo de problema que queremos resolver. Se valores armazenados em variáveis não precisam ser levados de uma iteração para outra, caso da **Figura 5.4(b)**, essas variáveis podem ser criadas dentro das repetições. No entanto, se os valores de iterações anteriores são importantes para as iterações seguintes, caso da **Figura 5.4(a)**, essas variáveis devem ser criadas fora da repetição.

```
Algoritmo variavelRepticao()
Início
        Inteiro i
        Para (i=1;i<=10;i=i+1)Faça
                Inteiro aux=i^2
                Imprimir aux
        Fim Para
        Imprimir aux
Fim
```

Figura 5.5: Algoritmo errado, que usa uma variável criada dentro de uma repetição fora dessa repetição.

Outra característica importante a mencionar é que variáveis criadas dentro de uma repetição só existem dentro dessa repetição. A **Figura 5.5** mostra um exemplo de algoritmo falho, que faz uso de uma variável criada dentro da repetição fora desta repetição. Note que a variável *aux*, criada para armazenar o valor do contador *i* elevado ao quadrado, é utilizada dentro da repetição. Isso está correto e é, inclusive, considerado uma boa prática.

O problema está em utilizar a variável *aux* fora da repetição, o que é feito logo após o **Fim Para**. Como variáveis criadas dentro de repetições só existem dentro das repetições, ao terminar a repetição, a variável deixa de existir. Logo, o erro está em tentar utilizar uma variável que não existe mais.

Caso você precise criar uma variável para auxiliar em cálculos ou em outras tarefas, e ela só for utilizada em uma estrutura (dentro de um desvio ou repetição, por exemplo), recomenda-se que tal variável seja declarada dentro da estrutura. Isto torna o código mais elegante e legível.

Para combinar estruturas condicionais e de repetição, também vale a regra do aninhamento, mostrada na Aula 4. Essa regra estabelece que, se uma estrutura *a* começa dentro de uma segunda estrutura *b,* então *a* deve terminar antes de *b.* Ou seja, *a* deve estar completamente contida dentro de *b*.

Exemplo: Suponha que queiramos ler dez números digitados pelo usuário e contabilizar quantos são pares. A **Figura 5.6** mostra uma possível solução para esse problema.

```
Algoritmo contarPares()
Início
        Inteiro i, cont=0,x
        Para (i=10;i<=100;i=i+10)Faça
                Imprimir “digite um número”
                Ler x
                Se (x%2==0)Então
                        cont = cont+1
                Fim Se
        Fim Para
        Imprimir “Foram digitados ”+cont+ “ números pares”
Fim
```

Figura 5.6: Exemplo de aninhamento de uma estrutura **Para** e uma estrutura **Se.**

A figura mostra um algoritmo composto por duas estruturas, uma de repetição e uma de desvio. Note que a estrutura de desvio começa e termina dentro da estrutura de repetição.

O algoritmo procede da seguinte maneira: inicialmente, a variável de contagem *i* é inicializada com o valor *10*. A seguir, é verificado se o valor de *i* é menor ou igual a *100*. Como *i* vale *10*, a condição é verdadeira e a repetição prossegue. Em seguida, é impressa uma mensagem para o usuário, pedindo para que ele digite um número. Na sequência, o número é lido e o algoritmo verifica se o número digitado é múltiplo de *2*, ou seja, se ele é par. Caso o número seja par, a variável *cont* é acrescida de uma unidade. Como última instrução da repetição, é feito o passo da contagem, fazendo com que *i* assuma o valor *20*. O algoritmo prossegue até que *i* assuma um valor maior que *100*. Por último, é impressa uma mensagem para o usuário, dizendo quantos dos números digitados foram pares.

Outro caso que deve ser mencionado é o aninhamento de estruturas de repetição. Considere duas estruturas de repetição: uma estrutura *a* externa e uma estrutura *b* posicionada dentro de *a*. Considere também que *a* está preparada para fazer *n* iterações e *b* para fazer *m* interações. Dado o posicionamento dessas estruturas, para cada iteração que *a* fizer, *b* fará *m* iterações. A **Figura 5.7** mostra um exemplo de aninhamento entre estruturas de repetição.

```
Algoritmo repeticoesAninhadas()
Início
   Inteiro i, j
   Para (i=1;i<=3;i=i+1)Faça
      Para (j=1;j<=3;j=j+1) Faça
         Imprimir "i vale "+i+" j vale "+j
      Fim Para
   Fim Para
Fim
```

Figura 5.7: Exemplo de aninhamento de repetições.

Na **Figura 5.7**, é possível observar que existem duas estruturas de repetição: uma que controla a variável *i* e outra que controla a variável *j*. Também observamos que a estrutura que controla a variável *j* está inteiramente posicionada dentro da estrutura que controla a variável *i*, o que caracteriza o aninhamento. Essa disposição faz com que, para cada interação da estrutura externa, todas as interações da estrutura interna sejam executadas. Ou seja, a repetição externa tem por finalidade repetir a estrutura de repetição interna. Desse modo, a saída que esse algoritmo produz é a seguinte:

i vale *1 j* vale *1*

i vale *1 j* vale *2*

i vale *1 j* vale *3*

i vale *2 j* vale *1*

i vale *2 j* vale *2*

i vale *2 j* vale *3*

i vale *3 j* vale *1*

i vale *3 j* vale *2*

i vale *3 j* vale *3*

Atividade 2

Atende ao Objetivo 2

Nesta atividade, você praticará algoritmos que combinam estruturas de repetição e de desvio para resolver problemas.

a) Faça um algoritmo que leia dez números informados pelo usuário e conte quantos deles foram pares e quantos foram ímpares.

b) Faça um algoritmo para determinar se um número qualquer, informado pelo usuário, é primo.

c) Sabendo que a série de Fibonacci pode ser descrita da seguinte maneira $f(1)=1$, $f(2)=1$ e $f(n)=f(n-1)+f(n-2)$ faça um algoritmo para calcular os termos da série. O número de termos n deve ser fornecido pelo usuário. Todos os termos da série de 1 até n devem ser mostrados.

d) Faça um algoritmo para resolver a seguinte série:

$$sen(x)=x-x^3/3!+x^5/5!-x^7/7!\ldots$$

Nessa série, *x* é um ângulo dado em radianos e o número de termos utilizados deve ser informado pelo usuário.

Resposta Comentada

a) Para resolver esta questão, basta modificar o algoritmo da **Figura 5.6** para que, ao invés de contar os números pares, ele conte os números ímpares.

b) Para resolver esta questão, primeiro é preciso relembrar o conceito de número primo. Um número primo é um número que só possui dois divisores: *1* e ele próprio. Baseado nessa propriedade, um algoritmo simples, para determinar se um número é primo ou não, usa a estratégia de contar quantos divisores esse número tem. Se existirem mais que dois divisores, ele não é primo. A solução é mostrada a seguir. Nela, a variável *n* é o número que será verificado. A variável *i* controla os valores dos possíveis divisores e a variável *cont* é responsável por armazenar quantos foram os divisores de *n*.

```
Algoritmo primo()
Início
   Inteiro i, n,cont=0
   Imprimir "Digite um número"
   Ler n
   Para (i=1;i<=n;i=i+1)Faça
      Se (n%i==0) Então
         cont=cont+1
      Fim Se
   Fim Para
   Se (cont ==2)Então
      Imprimir "O número "+n+ " é primo"
   Senão
      Imprimir "O número "+n+ " não é primo"
   Fim Se
Fim
```

Obs.: Existem vários modos para tornar esse algoritmo mais eficiente. Contudo, por questões didáticas, a eficiência de algoritmos não será abordada neste momento.

c) Para fazer um algoritmo para a série de Fibonacci, primeiro devemos compreender seu comportamento. A notação *f(1)* indica o primeiro termo da série, e a notação *f(2)* indica o segundo termo. O número de termos da série indica quantos serão os números gerados. Logo, se o número de termos é *5*, queremos gerar os *5* primeiros termos da série de Fibonacci, ou seja, *f(1)*, *f(2)*, *f(3)*, *f(4)* e *f(5)*. Os dois primeiros números já são dados *f(1)=1* e *f(2)=1*. Para achar os demais números, devemos utilizar a relação de recorrência exposta *f(n)=f(n−1)+f(n−2)*.

Assim, temos que *f(5)=f(4)+f(3)*. Portanto, para encontramos o valor de *f(5)*, precisamos antes encontrar *f(4)* e *f(3)*. Ainda temos que *f(4)=f(3)+f(2)*. O valor de *f(2)* é conhecido. Logo *f(4)=f(3)+1*. Já *f(3) = f(2)+f(1)*. Como os dois primeiros termos são conhecidos *f(3)=f(2)+f(1)=1+1=2*, portanto, *f(3)=2*. Com o valor de *f(3)*, podemos calcular *f(4)=f(3)+1=2+1=3*. Com o valor de *f(4)* e *f(3),* podemos calcular *f(5)=f(4)+f(3)=3+2=5*.

Note que não é possível calcular um valor de *f(n)* qualquer sem conhecer *f(n−1)* e *f(n−2)*. Assim, o modo mais conveniente de resolver esta questão é utilizar o *f(1)* e o *f(2)* para calcular o *f(3)*. Utilizar o *f(2)* e o *f(3)* para calcular o *f(4)* e, assim, sucessivamente.

Um algoritmo que utiliza essa estratégia para resolver o problema é mostrado a seguir. Nesse algoritmo, a variável *n* é utilizada para armazenar o número de termos a ser trabalhado. O comando *imprimir 1* na sexta linha é colocado porque se assume que a série terá obrigatoriamente,

pelo menos, um termo. Caso o número de termos seja maior ou igual a dois, imprimi-se o segundo termo (linha 8) e calcula-se o restante dos termos (quando necessário). Note que a repetição das linhas 9 a 14 calcula do terceiro termo em diante. Logo, se o número de termos pedido pelo usuário for igual a dois, essa repetição não será executada.

Dentro da repetição, foi criada uma variável *auxiliar* para calcular a soma dos dois termos anteriores, onde *t1* representa *f(n−1)* e *t2* representa *f(n−2)*. A soma é impressa e os valores dos termos trocados. Note que, para calcular o próximo termo, deve-se descartar o valor de *t2* e utilizar somente os valores de *t1* e o atual valor da variável *auxiliar*. Para fazer isso de maneira recorrente, o que devemos fazer é substituir o valor de *t2* por *t1* e substituir o valor de *t1* por *auxiliar*. Isso faz com que, na próxima iteração, os valores de *t1* e *t2* estejam atualizados – o que, por sua vez, faz com que o próximo termo da série seja calculado corretamente.

```
Algoritmo fibonacci()
Início
   Inteiro i, n,t1=1,t2=1
   Imprimir "Digite o número termos da Série"
   Ler n
   Imprimir 1 /* Primeiro termo da Série*/
   Se (n>=2) Então
   Imprimir 1 /* Segundo termo da Série*/
   Para (i=3;i<=n;i=i+1)Faça
      Inteiro auxiliar = t1+t2
      Imprimir auxiliar
      t2=t1
      t1=auxiliar
   Fim Para
Fim
```

Obs: Lembre-se de que o que está entre /* e */ é um comentário, logo, não faz parte do algoritmo.

d) Nesta série, *x* é um ângulo dado em radianos e o número de termos utilizados deve ser informado pelo usuário. Para resolver este problema, primeiro é preciso entender sua notação. As reticências no final da equação indicam que ela é uma série infinita, ou seja, tem um número de termos infinito. Como não é correto fazer um algoritmo com infinitos passos, devemos estabelecer um critério de parada para o cálculo da série.

O próprio enunciado já determina que o critério de parada seja o número de termos informado pelo usuário. Um termo da série é compreendido como um valor e um sinal. Assim, o primeiro termo desta série é $+x$. O segundo é $-x^3/3!$. O terceiro é $+x^5/5!$. E, assim, sucessivamente. Desse

modo, se o usuário determinar que o cálculo da série deve ser feito com dois termos, então retornaremos o resultado de $x - x^3/3!$.

Ainda sobre os termos da série, é importante mencionar que o primeiro termo é $x = x^1/1!$ e que, a cada termo, ocorrem duas mudanças: 1) o expoente do ângulo e o parâmetro do fatorial no denominador são acrescidos de duas unidades e 2) o sinal do termo é invertido. Assim, para obtermos o quinto termo, utilizamos como base o quarto, aumentamos o expoente do ângulo e o parâmetro do fatorial no denominador em duas unidades e invertemos o sinal. Como o quarto termo é $-x^7/7!$, o quinto fica $+x^9/9!$, e, assim, sucessivamente.

Por último, note que a série é apresentada como *sen(x)=série*. Esse tipo de notação significa que o valor do *sen(x)* é aproximado pela série. Logo, o valor do *sen(x)* será encontrado por meio da série, e não o contrário. Também é importante mencionar que, quanto maior o número de termos da série, mais próximo do valor do *sen(x)* o resultado estará.

Tendo em vista as explicações mostradas, uma possível solução é apresentada a seguir, na qual a resposta final é calculada termo a termo, começando em *x1/1!* e indo até o *n*-ésimo termo. A variável expoente é utilizada para armazenar o valor do expoente do ângulo e do parâmetro do fatorial no termo atual. Como esses valores são acrescidos de dois em dois na série, o algoritmo também deve fazer isso. Para fazer a inversão de sinal, a solução apresentada utiliza uma variável que é multiplicada pelo valor do termo. Como a cada termo o sinal é invertido, essa variável é multiplicada por *–1*, o que faz com que seu valor se alterne entre *1* e *–1*.

```
Algoritmo seno()
Início
   Inteiro i, n, expoente=1
   Real x,soma=0,sinal=1
   Imprimir "Digite o número termos da Série e o ângulo em radianos"
   Ler n,x
   Para (i=1;i<=n;i=i+1)Faça
      Inteiro fat = 1,j
      Para (j=2;j<=expoente;j=j+1)faça
         fat = fat * j
      Fim Para
      soma = soma+ sinal* x^expoente/fat
      expoente = expoente +2
      sinal = -1*sinal
   Fim Para
   Imprimir "O resultado é: "+soma
Fim
```

Obs.: Também existem outros modos para fazer este exercício. Um modo muito utilizado é separar as somas das subtrações. Nesta série, os termos ímpares são todos positivos, e os pares, negativos. Assim, calcula-se a soma dos termos ímpares e a soma dos termos pares, separadamente. No final, basta subtrair a soma dos pares da soma dos ímpares. Note, porém, que os expoentes, nesse caso, devem ser acrescidos de modo diferente.

Resumo

Nesta aula, você aprendeu a lidar com estruturas de repetição com contador e a compor algoritmos com aninhamento de estruturas de desvio e repetição. Observamos que a quantidade de repetições é determinada pela combinação de três componentes da estrutura de repetição (inicialização, condição e passo) e que, quando mal elaborada, essa combinação pode levar à construção de algoritmos mal formados. Também estudamos a criação de variáveis dentro de estruturas de repetição e mostramos que esse tipo de variável não pode ser utilizado fora da estrutura.

Informação sobre a próxima aula

Na próxima aula, você verá como lidar com a repetição de tarefas quando não é possível determinar o número de vezes que a tarefa será repetida.

Aula 6

Repetição: parte II

Tiago Araújo Neves

Meta

Expor os conceitos de repetição sem contador e as duas estruturas usadas para este fim.

Objetivos

Esperamos que, ao final desta aula, você seja capaz de:

1. definir e aplicar o conceito de estruturas de repetição sem contador;
2. fazer algoritmos que combinam estruturas de desvio e repetição, com e sem contador, para resolver problemas.

Introdução

Até agora, você aprendeu que, para construir algoritmos que repetem múltiplas vezes uma mesma tarefa, é necessário fazer uso de estruturas de repetição. Entretanto, foram apresentados somente casos em que é possível determinar o número de vezes que a tarefa será repetida. Existem casos nos quais não é possível determinar o número de vezes a repetir uma mesma tarefa e, nestas situações, o uso de uma estrutura de repetição com contador se torna inadequado.

Exemplo: Faça um algoritmo que leia números positivos que o usuário digita e os imprima na tela. O algoritmo deve parar sua execução assim que o usuário digitar um número negativo ou zero. Este é um caso simples, em que não é adequado utilizar uma estrutura de repetição com contador. Para usar a estrutura com contador, seria necessário perguntar ao usuário quantos números, no total, ele quer digitar. Entretanto, nem sempre o usuário sabe, *a priori*, quantas vezes fará uma tarefa. Um exemplo típico é a compra de itens pela internet. Nem sempre o usuário compra todos os itens que planejava comprar, ou ainda é possível que ele aproveite alguma promoção e até compre mais do que planejava. Supor um número fixo de itens de compra, neste caso, pode não ser uma boa estratégia.

Com esses exemplos, fica claro que repetição com contadores é uma opção que traz certas limitações. Em alguns casos, é necessário repetir tarefas sem pressupor o número final de repetições. Para esses casos, usaremos uma estrutura de repetição sem contador.

Código eficiente

É importante mencionar que tudo o que pode ser feito com uma estrutura de repetição com contador também pode ser feito com uma estrutura de repetição sem contador. Então surge uma pergunta: por que estudar dois tipos de estrutura de repetição, se a estrutura de repetição sem contador resolve todos os casos? A resposta é: eficiência. As estruturas de repetição com contador podem ser otimizadas para executar mais rapidamente nos computadores. Além disso, para problemas com necessidade de contadores, a estrutura com contador proporciona um algoritmo mais simples.

Nesta aula, você aprenderá não só a lidar com duas estruturas de repetição sem contador, mas também a combinar estruturas com/sem contador, juntamente com estruturas de desvio condicional para compor algoritmos. Além disso, veremos casos em que será analisada a necessidade de estruturas sem contador e quando devemos aplicar cada uma delas.

Estrutura de repetição sem contador e controle no início

A **Figura 6.1** mostra a sintaxe da estrutura *Enquanto*. Esse tipo de repetição sempre começa com a palavra reservada *Enquanto* e termina com *Fim Enquanto*. As instruções a serem repetidas encontram-se entre as palavras reservadas *Faça* e *Fim Enquanto*. Entre o *Enquanto* e o *Faça***,** encontra-se uma expressão lógica entre parênteses. A semântica da estrutura *Enquanto* é muito simples. As instruções contidas dentro do *Enquanto* serão repetidas enquanto a expressão lógica for verdadeira. Quando a expressão for avaliada como falsa, a repetição é encerrada.

Note que uma das instruções dentro da estrutura de repetição deve cuidar para que, em algum momento, a expressão lógica resulte em falso, pois, caso contrário, a expressão sempre resultará em verdadeiro e, consequentemente, a estrutura de repetição não será interrompida.

```
Enquanto ( Expressão Lógica ) Faça
  Instruções
Fim Enquanto
```

Figura 6.1: Sintaxe da estrutura *Enquanto*.

Agora que você conhece a estrutura *Enquanto*, vejamos seu uso em um exemplo simples. A **Figura 6.2** mostra o uso da estrutura *Enquanto*. A ideia deste exemplo é similar a de alguns jogos de perguntas e respostas presentes na rede. Nesse tipo de jogo, os candidatos têm de responder a várias perguntas. Toda vez que uma resposta errada é dada, a pergunta é refeita. Quando a resposta certa é dada, o jogo passa para a próxima pergunta. Ganha quem terminar as perguntas em menos tempo. No nosso exemplo, simularemos apenas uma pergunta. Enquanto a resposta for errada, o algoritmo exigirá uma outra resposta. Quando a resposta certa for inserida, o algoritmo informará que a resposta está certa e encerrará sua execução.

```
Algoritmo testeEnquanto()
Início
   Real x
   Imprimir "Digite o valor do perímetro de um quadrado de lado 2"
   Ler x
   Enquanto (x!=8)Faça
      Imprimir "Resposta incorreta. Tente novamente."
      Ler x
   Fim Enquanto
   Imprimir "Resposta correta"
Fim
```

Figura 6.2: Exemplo de uso da estrutura *Enquanto*.

O algoritmo procede da seguinte maneira: no início, cria-se uma variável *Real x*. Nesse exemplo, *x* é utilizada para fazer a leitura dos números. Em seguida, é exibida uma mensagem para o usuário e é feita uma leitura para o valor de *x*. Nesse caso, a primeira leitura é feita fora da estrutura de repetição porque o usuário pode digitar a resposta certa na primeira tentativa. Em seguida, é avaliada a condição. Caso a expressão lógica resulte em verdadeiro, ou seja, se *x* for diferente de oito, as instruções dentro da repetição serão executadas. Caso contrário, o fluxo é desviado para o *Fim Enquanto*, a mensagem "resposta correta" é exibida e o algoritmo é encerrado.

Dentro da repetição, é feita uma nova interação com o usuário. Como o valor de *x* é lido novamente dentro da repetição, a condição é avaliada novamente. Caso ela continue verdadeira, as instruções serão repetidas novamente. Caso ela se torne falsa, o fluxo é desviado para o *Fim Enquanto*, a mensagem "resposta correta" é exibida e o algoritmo se encerra. Note que a condição de parada é expressa em função do valor de *x*. Isso faz com que as instruções dentro da repetição sejam executadas enquanto o valor de *x* for diferente de oito.

```
Algoritmo testeEnquanto()
Início
  Real x
  Imprimir "Digite o valor do perímetro de um
quadrado de lado 2."
  Ler x
  Enquanto(x!=8)Faça
    Imprimir "DResposta incorreta.
Tente novamente."
    Ler x
  Fim Enquanto
  Imprimir "Resposta correta"
Fim
```

Cria-se a variável *Real x*, que é utilizada para fazer a leitura da resposta ao problema.

Mensagem para o usuário apresentando o problema.

Após a primeira leitura, é avaliada a condição. Caso a expressão lógica resulte em verdadeiro, ou seja, se *x* for diferente de 8, as instruções dentro da repetição serão executadas: o usuário receberá uma mensagem de erro e terá uma nova chance de solucionar o problema. Caso contrário, se *x=8*, o fluxo é desviado para *Fim Enquanto* e o usuário receberá uma mensagem de sucesso.

Figura 6.3: Exemplo de uso da estrutura *Enquanto* detalhado.

Estrutura de repetição sem contador e controle no Fim

A **Figura 6.4** mostra a sintaxe da estrutura de repetição *Faça-Enquanto*. Essa estrutura começa com a palavra reservada *Faça* e termina com *Enquanto*, seguido de uma expressão lógica. A semântica é similar à da estrutura *Enquanto*. As instruções dentro da estrutura *Faça* são repetidas enquanto a expressão lógica resultar em verdadeiro. A diferença entre as duas estruturas é que as instruções da estrutura *Faça* são executadas pelo menos uma vez, ou seja, independentemente do valor da expressão, as instruções são executadas na primeira vez. Dependendo do valor da expressão, haverá uma segunda iteração e, assim, sucessivamente.

```
Faça
  Instruções
Enquanto ( Expressão Lógica )
```

Figura 6.4: Sintaxe da estrutura *Faça-Enquanto*.

É importante mencionar que as duas estruturas (Faça-Enquanto e Enquanto) podem ser intercambiadas com as devidas adaptações. Existem casos onde, por simplicidade, é mais adequado utilizar uma ou outra estrutura. Exemplo: Faça um algoritmo para calcular a raiz quadrada de um número real positivo. O algoritmo deve se certificar de que o usuário digitará um número positivo antes de calcular a raiz. A **Figura 6.5** mostra uma possível solução para o problema. Note que fazer algo similar com a estrutura Enquanto requer pelo menos uma instrução a mais, que seria uma inicialização da variável c. Neste exemplo em particular, qualquer instrução extra implica um aumento de mais do que 10% no número de instruções do algoritmo. Como em alguns casos o número de instruções (tamanho) de um programa (implementação real de algoritmos) tem impacto direto no tempo de execução (programas que cabem dentro da memória cache dos computadores executam muito mais rápido), usar mais instruções que o necessário pode não ser vantajoso.

```
Algoritmo testeFaca1()
Início
   Real c;
   Faça
      Imprimir "Digite um número maior que zero"
      Ler c
   Enquanto (c<=0)
   Imprimir "raiz é: "+ (c^(1.0/2.0))
Fim
```

Figura 6.5: Exemplo de uso da estrutura *Faça-Enquanto.*

Vale a pena ressaltar que eficiência de algoritmos não é o foco desta disciplina. O objetivo, neste momento, é aprender a compor algoritmos corretos, capazes de fornecer respostas corretas para os problemas abordados. Nesse sentido, você pode utilizar a estrutura que preferir, desde que seu algoritmo esteja correto.

Atividade 1

Atende aos Objetivos 1 e 2

1. Faça um algoritmo que leia vários textos e os imprima na tela. O algoritmo deve parar a leitura quando for digitado o texto "!@#$". O algoritmo também deve calcular e mostrar quantos textos foram digitados. Obs: o texto "!@#$" não deve ser contabilizado.

2. O algoritmo da **Figura 6.2** também pode ser feito por meio do uso combinado da estrutura *Enquanto* com uma estrutura de desvio aninhada. Faça um algoritmo que utilize uma estrutura de repetição sem contador e uma estrutura de desvio para evitar que a primeira leitura seja feita fora da repetição.

3. Faça um algoritmo que calcule a média aritmética de uma turma em que o número de alunos não é informado. O algoritmo deve parar de calcular a média quando uma nota negativa for digitada.

4. Um determinado material radioativo perde metade de sua massa a cada *x* segundos. Faça um algoritmo para calcular o tempo, em horas, minutos e segundos para que uma determinada massa *y* deste material seja menor ou igual a um determinado limiar de massa *z*, sendo *y>z*. Os parâmetros *x, y, z* devem ser informados pelo usuário.

5. Faça um programa que encontre o máximo divisor comum entre dois números *a* e *b,* de acordo com o seguinte critério:

a) Calcule o resto *c* da divisão de *a* por *b.*

b) Se o resto *c* for zero, *b* é o m.d.c.

c) Se o resto não for zero, substitua *a* por *b* e *b* por *c*, e execute novamente o passo 1.

Resposta Comentada

1. Para resolver esta questão, podemos usar como base o algoritmo da **Figura 6.2**. O que deve ser feito a mais é uma estrutura condicional para contabilizar a quantidade de textos digitados. Uma possível solução é apresentada a seguir.

A variável que contabiliza a quantidade de textos é *cont*. A variável *tex* é utilizada para leitura e impressão dos textos. Note que, como foi utilizada a estrutura *Enquanto*, é necessário atribuir um valor inicial para a variável *tex*, utilizada na condição da estrutura de repetição. A estrutura de desvio *Se* é utilizada para filtrar os textos válidos. Toda vez que um texto válido é digitado, é impresso pela instrução *Imprimir tex* e contabilizado pela instrução *cont = cont+1*. Ao final, o algoritmo imprime quantos textos válidos foram inseridos pelo usuário.

```
Algoritmo repetirTexto()
Início
   Inteiro cont=0
   Texto tex= “vazio”
   Enquanto (tex != “!@#$”)Faça
      Imprimir “Digite um texto”
      Ler tex
      Se(tex != “!@#$”) Então
         Imprimir tex
         cont=cont+1
      Fim Se
   Fim Enquanto
   Imprimir “A quantidade de textos digitados foi ”+ cont
Fim
```

2. Para esta questão, devemos fazer várias adaptações. Uma possível solução é mostrada a seguir.

A primeira adaptação feita é a inicialização da variável *x* para que a repetição possa ser inicializada corretamente. Na solução apresentada, foi escolhido o valor *1*, mas qualquer valor maior que zero poderia ser utilizado. Deve-se ficar atento, contudo, ao fato de que o valor inicial da variável *x* não foi digitado pelo usuário, logo, ele não deve ser impresso, a não ser que o usuário digite *1*. Para corrigir isso, a primeira coisa a fazer dentro da repetição é uma mudança no valor da variável *x*. Para isso, é impressa uma mensagem passando instruções para o usuário e é feita uma leitura da variável. Em seguida, é feita uma verificação sobre o valor de *x*. Caso o usuário tenha digitado um valor maior que zero, este valor é impresso e o algoritmo retorna ao início da repetição. Caso o valor digitado seja menor ou igual a zero, o comando de impressão é ignorado e o algoritmo volta ao início da repetição. Ao terminar a primeira iteração do algoritmo, uma nova verificação é feita sobre o valor de *x*. Caso o usuário tenha digitado um valor maior que zero na iteração anterior, o algoritmo começa uma nova iteração. Caso contrário, o fluxo é desviado para o final da estrutura de repetição e o algoritmo é encerrado.

```
Algoritmo enquantoMaisCondicional()
Início
   Real x=1
   Enquanto (x>0) Faça
      Imprimir "Digite um número positivo para imprimi-lo na tela"
      Ler x
      Se (x>0) Então
         Imprimir x
      Fim Se
   Fim Enquanto
Fim
```

3. Para resolver esta questão, será usada uma estratégia diferente. Uma possível solução é apresentada a seguir.

Nesta solução, são criadas uma variável *cont* para contabilizar quantos alunos tem a turma, uma variável *soma* para calcular a soma das notas e uma variável *nota* para lidar com as leituras de cada nota, individualmente. Inicialmente, o valor zero é atribuído às variáveis *soma* e *cont*. Em seguida, é feita uma repetição, que é responsável por ler cada uma das notas e atualizar a soma e o número de alunos. Note que, mesmo quando for digitada uma nota negativa, ela é contabilizada em *soma* e é acrescida uma unidade à variável *cont*. Isso é intencional, pois a estra-

tégia utilizada é corrigir o valor dessas variáveis, posteriormente. Logo após a repetição, a variável *cont* é decrescida de uma unidade. Isso é feito para corrigir o valor desta variável.

Note que, ao digitar uma nota negativa, *cont* também é aumentada em uma unidade pela instrução da linha 9. Contudo, a instrução da linha 10 não deixa que a repetição prossiga. Isso significa que, ao sair da repetição, *cont* estará sempre com uma unidade a mais do que deveria. Sendo assim, basta subtrair uma unidade para corrigir seu valor. Um acerto similar é feito para a variável *soma*. Note que a nota negativa digitada é incorporada em *soma* pela instrução da linha 8. Mas a instrução da linha 10 não deixa que a repetição prossiga. Logo, o que deve ser feito é subtrair aquele valor negativo incorporado para corrigir o valor de soma. Por último, a média é calculada e exibida.

```
Algoritmo media()
Início
   Inteiro cont=0
   Real soma=0,nota
   Faça
      Imprimir "Digite uma nota"
      Ler nota
      soma=soma+nota
      cont=cont+1
   Enquanto (nota>=0)
   cont=cont-1
   soma=soma - nota
   Imprimir "A média foi "+ soma/cont
Fim
```

Para maior compreensão, vamos dar um exemplo. Suponha uma turma de três alunos com notas 5.2, 6.1 e 7.3. Suponha, também, que o valor digitado para parar o algoritmo seja −8. Na primeira iteração, *nota* recebe o valor 5.2 informado pelo usuário. *Soma* recebe seu próprio valor (0) acrescido de *nota* (5.2), que resulta em 5.2. A variável *cont* tem seu valor (0) acrescido de uma unidade, portanto, recebe 1. Na segunda iteração, *nota* recebe o valor 6.1 informado pelo usuário. *Soma* recebe seu próprio valor (5.2) acrescido de *nota* (6.1), que resulta em 11.3. A variável *cont* tem seu valor (1) acrescido de uma unidade, portanto, recebe 2. Na terceira iteração, *nota* recebe o valor 7.3 informado pelo usuário. *Soma* recebe seu próprio valor (11.3) acrescido de *nota* (7.3), que resulta em 18.6. A variável *cont* tem seu valor (2) acrescido de uma unidade, portanto, recebe 3. Na quarta iteração, *nota* recebe o valor -8 informado pelo usuário. *Soma* recebe seu próprio valor (18.6) acrescido de *nota*

(−8), que resulta em 10.6. A variável *cont* tem seu valor (3) acrescido de uma unidade, portanto, recebe 4. Como *nota* tem um valor negativo, a repetição se encerra. Em seguida, *cont* é reduzida em uma unidade, portanto, *cont* tem seu valor configurado para 3. *Soma* tem subtraído de seu valor (10.6) o valor de *nota* (−8), ou seja, *soma* = 10.6−(−8), que resulta em 18.6. Na última linha, é calculada a média 18.6/3 que resulta em 6.2, que é exatamente a média das notas da turma mencionada.

4. A primeira coisa a ser feita é perceber que, apesar de a meia vida de materiais radioativos poder ser descrita como uma progressão geométrica, aplicar as fórmulas de progressão pode não dar bons resultados. Note que, apesar de *y* sempre ser o termo inicial da p.g. de razão meio, não necessariamente *z* é parte da progressão. Exemplo: suponha que um material perde metade de sua massa a cada 45 segundos e queremos saber quanto tempo é necessário para que 10 Kg deste material atinjam um limiar de 4 Kg. Note que a progressão seria 10, 5, 2.5 ... Quando o limiar é um termo da progressão, fica trivial encontrar a quantidade de termos. Por exemplo: se o limiar fosse 2.5, bastaria utilizar a equação geral do termo da progressão dada por $a_n=a_1q^{n-1}$, em que $a_n=2.5$, $a_1=10$ e $q=1/2$ para encontrar o valor de n.

O problema é quando o limiar desejado está entre os termos da progressão, como é o caso do exemplo, em que o limiar é 4. Isso inviabiliza o uso das expressões da progressão para resolver o problema, visto que o limiar desejado não é um termo da progressão. Desse modo, a solução envolve calcular os termos da progressão até encontrar o primeiro termo menor ou igual ao limiar, calculando, assim, o número de meias vidas necessárias para chegar até lá.

Uma vez que temos o número de meias vidas, basta multiplicá-lo pelo tempo de cada meia vida para obtermos o tempo total do processo, em segundos. No exemplo mencionado, note que são necessárias duas meias vidas para atingir o limiar. Como cada meia vida leva 45 segundos, tem-se um total de 90 segundos. Agora, basta transformar isso em horas, minutos e segundos, ou seja, 0h 1m e 30s. Para fazer isso, basta encontrar os quocientes e os restos de divisões apropriados. Por exemplo, uma hora tem 3.600 segundos, assim, para saber quantas horas o processo levou, basta encontrar o quociente do tempo por 3.600. Note que o resto desta divisão (tempo % 3.600) apresenta os segundos, já excluídas as horas. Basta, então, transformar este resto em minutos, utilizando como divisor o valor 60. Um processo análogo deve ser feito para encontrar os segundos. Uma possível solução é apresentada a seguir.

Nesta solução, a variável *cont* é utilizada para contar o número de meias vidas. A variável *x* armazena o tempo da meia vida e as variáveis *y* e *z* são a massa inicial e o limiar de massa, respectivamente.

```
Algoritmo meiaVida()
Início
   Real y, z
   Inteiro x, cont
   Imprimir "digite a massa inicial, final e o tempo da meia vida respectivamente"
   Ler y, z, x
   Enquanto (y>z) Faça
      y=y/2
      cont = cont+1
   Fim Enquanto
   Inteiro segundosTotais = x*cont
   Inteiro horas = segundosTotais/3600
   Inteiro minutos = segundosTotais%3600/60
   Inteiro segundos = segundosTotais%3600%60
   Imprimir  "Tempo= "+horas+ " h "+ minutos+ " m "+ segundos + " s"
Fim
```

Note que, uma vez calculado o tempo total do processo em segundos, há várias formas de fazer a transformação do tempo. Uma delas é ir subtraindo o valor total e calcular as horas e minutos através de estruturas de repetição.

5. Para resolver esta questão, a primeira tarefa é analisar a descrição dos passos do algoritmo. Um exemplo pode ajudar neste caso.

Suponha que queiramos identificar o m.d.c. entre 25 e 15. Por experiência, sabe-se que este m.d.c. é 5. Seja *a*=25 e *b*=15. O primeiro passo é calcular o valor do resto da divisão de *a* por *b* e armazenar em *c* (ou seja, *c*=25%15, que resulta em 10). Ao analisar o valor do resto, percebe-se que ele é diferente de zero. Então, deve-se substituir *a* por *b* (ou seja, o valor de *a* passa de 25 para 15), substituir *b* por *c* (ou seja, *b* passa de 15 para 10) e calcular um novo valor de *c*. Note que agora *c*=15%10, que resulta em 5.

Novamente, este valor é diferente de zero, portanto, o valor de *a* é alterado para 10 e o valor de *b* alterado para 5. Na terceira iteração, *c*=10%5, que resulta em zero. Ao analisar este valor, percebe-se que *b* é o m.d.c. . Como *b* vale 5, então, tem-se a resposta correta do problema. Uma possível solução para este problema é mostrada a seguir.

```
Algoritmo mdc()
Início
   Inteiro a,b,c
   Imprimir “Digite 2 números para calcular o m.d.c. ”
   Ler a,b
   Faça
      c = a%b
      Se (c == 0) Então
         Imprimir “o m.d.c. é ”+b
      Senão
         a=b
         b=c
      Fim Se
   Enquanto (c!=0) Faça
Fim
```

Resumo

Nesta aula, você aprendeu a lidar com estruturas de repetição sem contador e a compor algoritmos com aninhamento de estruturas de desvio e repetição. Foi visto que a quantidade de repetições nem sempre pode ser determinada e que, nesses casos, utilizar estruturas de repetição sem contador é mais adequado.

Informações sobre a próxima aula

Na próxima aula, você verá como lidar com armazenamento de dados em vetores e matrizes.

Aula 7

Vetores

Tiago Araújo Neves

Meta

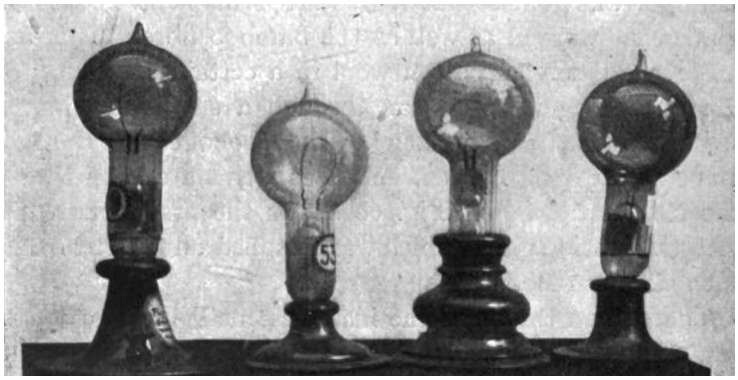
Expor os conceitos de vetores, mostrando sua sintaxe e funcionamento.

Objetivos

Esperamos que, ao final desta aula, você seja capaz de:

1. definir e aplicar o conceito de vetores;
2. fazer algoritmos que combinem estruturas de desvio e repetição juntamente com vetores para resolver problemas.

Introdução



William J. Hammer

Figura 7.1: Lâmpadas elétricas: invenção de Thomas Edison.
Fonte: http://commons.wikimedia.org/wiki/File:Edison_incandescent_lights.jpg

A história diz que Thomas Edison, grande inventor norte-americano, gastou algumas centenas de tentativas para aperfeiçoar o seu modelo de lâmpada elétrica. A cada tentativa errada, ele tomava notas detalhadas de cada experimento e se certificava de que as falhas não se repetiriam. Os historiadores se utilizaram dessas notas para escrever vários livros sobre a criação da lâmpada e também sobre temas relacionados. Uma pergunta que se pode fazer é a seguinte: será que o trabalho de Edison teria sido mais fácil se ele não tivesse guardado tantos registros? A resposta é negativa. Edison usava as anotações para não cometer os mesmos erros. Existe uma história, não confirmada, de que, ao ser questionado por um repórter sobre suas várias falhas, ele respondeu: "Eu não falhei, apenas aprendi muitos modos de como não se deve fazer uma lâmpada". Mesmo que a história não seja confirmada, ela serve de ilustração sobre como as informações das notas ajudaram Edison a não repetir erros.

Outra analogia que podemos aplicar para entender a necessidade do armazenamento de dados é quando temos tantos compromissos para fazer em um determinado dia, que fica impossível lembrar de todos sem o auxílio de algum dispositivo, como agenda, celular etc. Ou seja, em muitos casos, não poder armazenar informações de alguma forma torna extremamente difícil, ou mesmo impossível, a realização de algumas tarefas. De modo semelhante, de nada adianta ter as informações armazenadas sem que seja possível acessá-las. Imagine qual seria a utilidade de uma agenda em que se pudesse inserir informações, mas não se pudesse consultá-las.

Variância

Medida de dispersão utilizada em estatística. Ela indica o quão longe, em geral, os valores de uma determinada medida se encontram do valor esperado dela.

Um exemplo real com que muitos engenheiros lidam no dia a dia é o calculo de **variância**. A variância pode ser dada por: $1/n * \sum_{i=0}^{n-1} (x[i]-\mu)^2$, em que μ é a média do conjunto de medidas que queremos analisar e *n* é a quantidade de elementos desse conjunto. Note que você já sabe fazer um algoritmo para calcular parte dos elementos da equação. Algoritmos para cálculo de média já foram apresentados. Mas esse é apenas o começo do cálculo de variância.

Um exemplo pode ajudar a compreender a complexidade da operação:

1. a partir de um conjunto numérico:

   $x=\{1,2,3,4,5,6\}$,

2. calcular a média do conjunto (μ):

   $(1+2+3+4+5+6)/6=3.5$,

3. subtrair essa média de cada um dos elementos utilizados no cálculo dela (por conveniência, cria-se um segundo conjunto):

   $x'=\{(1-3.5),(2-3.5),(3-3.5),(4-3.5),(5-3.5),(6-3.5)\}$

   $x'=\{-2.5,-1.5,-0.5,0.5,1.5,2.5\}$,

4. elevar o resultado dessas subtrações ao quadrado, criando um terceiro conjunto:

   $x''=\{(-2.5)^2,(-1.5)^2,(-0.5)^2,(0.5)^2,(1.5)^2,(2.5)^2\}$

   $x''=\{6.25,2.25,0.25,0.25,2.25,6.25\}$,

5. somar os quadrados:

   $6.25+2.25+0.25+0.25+2.25+6.25=17.5$

6. e dividir esse total pelo número de elementos do conjunto (*n*). $17.5/6=2.9166666...$

   Ou seja, a variância do conjunto {1, 2, 3, 4, 5, 6} é algo próximo de 2.91666.

Nesta aula, você aprenderá a lidar com vetores, que são um dos modos de armazenar informações em algoritmos.

Vetores: declaração e acesso

Vetores, *arrays* ou matrizes unidimensionais são sinônimos dentro do mundo da informática. Fisicamente, são um conjunto identificável de posições de memória, contíguas, nas quais se podem armazenar informações de um tipo específico. Podem ser vistos como um conjunto de elementos do mesmo tipo em que se pode identificar, acessar e alterar cada um desses elementos.

Dois exemplos de declaração de *arrays* são dados a seguir:

```
Real [ ] notas
Inteiro [ ] idades, faltas
```

A primeira linha é utilizada para declarar um único vetor de elementos de um determinado tipo. A sintaxe geral de declaração para um único vetor é a seguinte:

```
tipo [ ] nome
```

Nela, os caracteres *[e]* são a indicação de que o que está sendo declarado são *arrays*. O tipo indica a natureza da informação que estará contida nos *arrays*. Qualquer tipo válido pode ser usado. No exemplo acima, notas é o nome de um único vetor de números reais.

A segunda linha é utilizada para declarar vários *arrays* distintos de elementos de um mesmo tipo:

```
tipo [ ] nome1, nome2,..., nomen
```

Nessa sintaxe, *nome1*, *nome2* etc. são os nomes dos *arrays* que serão criados. Pode haver quantos nomes forem necessários, desde que eles sejam separados por vírgulas. No exemplo mostrado, são criados dois vetores de números inteiros, um chamado *idade* e outro chamado *faltas*.

Uma operação fundamental ao lidar com *arrays* é a **alocação**. No momento em que um *array* é declarado, são informados o nome e o tipo dos seus elementos, mas nada é informado a respeito do seu tamanho. A tarefa de definir o tamanho de um *array* é feita separadamente durante a alocação. Isso permite que esses tamanhos sejam definidos após a sua criação.

Alocação

Tarefa de solicitar/utilizar memória ao/do sistema operacional feita durante a execução de programas de computador.

O tamanho de um *array* é sempre um número inteiro, independentemente do tipo dos elementos dos *arrays*. Esse tamanho pode ser definido por constantes, variáveis e expressões matemáticas, desde que o resultado seja um número inteiro. Afinal, não faz sentido um conjunto ter três elementos e meio, por exemplo. Ou ele tem três ou ele tem quatro elementos.

A alocação é feita utilizando-se um operador de atribuição seguido de um tipo, que, por sua vez, é seguido de um tamanho entre colchetes (=tipo[tamanho]). A **Figura 7.2** mostra exemplos de declaração e alocação de vetores. Nessa figura são declarados dois *arrays*, um chamado *notas* e outro chamado *faltas*. Para notas é atribuído um tamanho determinado pela variável *n*, ou seja, *notas* será um vetor com três elementos, no qual cada elemento é um número real. Para *faltas* é atribuído um tamanho determinado por uma constante (10). Logo, *faltas* será um *array* de 10 elementos, em que cada elemento é um número inteiro.

```
Algoritmo declaracaoVetores()
Início
   Inteiro n=3
   Real [ ] notas = Real[n]
   Inteiro [ ] faltas
   faltas = Inteiro [10]
Fim
```

Figura 7.2: Exemplo de declaração e alocação.

Note que o fato de poder utilizar uma variável como tamanho de um *array* auxilia muito na composição de algoritmos. Isso permite fazer cálculos e/ou interações com o usuário para determinar corretamente os tamanhos dos vetores. Um fato importante a ser mencionado é que os *arrays* não podem ter seu tamanho alterado. Uma vez que se determina que o *array faltas* terá 10 posições, não é possível alterar seu tamanho para 11 ou 8. O tamanho de *notas* será sempre 10, embora não seja obrigatório usar todas as posições do *array*. Isso significa que podemos alocar um vetor de 15 posições e utilizar somente 8, por exemplo.

É importante ressaltar que, apesar de a suposição de memória infinita ser válida para algoritmos, a mesma não pode ser aplicada para a programação de computadores reais. Ao alocar um vetor de 15 posições e utilizar somente 8, um programa desperdiça 7 posições que poderiam ser utilizadas por outros programas. Portanto, deve-se ter cautela ao utilizar esse tipo de recurso.

Na quarta linha do algoritmo da **Figura 7.2**, tem-se o *array notas* sendo criado e alocado. Agora note que o mesmo não ocorre com o vetor *faltas*. O *array faltas* é criado na linha 5, mas só é alocado na linha 6. Isso torna possível, por exemplo, fazer um algoritmo que crie um vetor, faça a leitura de uma variável e utilize essa variável como tamanho de um vetor. A **Figura 7.3** mostra um exemplo dessa possibilidade. Nessa figura, o vetor *notas* é criado, mas não alocado. Em seguida, é criada a variável *n* e é feita uma interação com o usuário para que este informe o valor de *n*. Logo depois, o resultado da expressão *n+3* é utilizado como tamanho para o vetor *notas*. Logo, se o usuário insere 4 como valor para *n*, então *notas* terá 7 posições.

```
Algoritmo declaracaoVetores2()
Início
   Real [ ] notas
   Inteiro n
   Ler n
   notas = Real [n+3]
Fim
```

Figura 7.3: Exemplo de alocação com expressão aritmética.

Uma vez que foi definido como declarar vetores, deve-se definir o mecanismo de acesso aos elementos. Para isso, nesta disciplina utilizaremos a sintaxe

```
nome[posição].
```

Nela, *nome* é o identificador do *array* a ser acessado e *posição* indica o índice do elemento desejado. É importante mencionar que as posições começam de zero (0) e vão até o tamanho do vetor, subtraído de um (1). Dessa forma, se temos um *array* de 10 posições, a primeira posição é acessada através do índice 0, a segunda pelo índice 1 e assim sucessivamente até a décima posição, que é acessada através do índice 9. Um exemplo é um *array* de 5 elementos inteiros com as primeiras potências de 2. Nesse caso, como pode ser visto na tabela a seguir, sabendo que o nome do vetor é *potencias2*, pode-se, por exemplo, imprimir o valor da quarta posição através de um comando *Imprimir potencias2[3]*. O resultado será a impressão do número 8, que é o valor contido na quarta posição do vetor *potencias2.*

Valores das potências de 2	1	2	4	8	16
Posição	0	1	2	3	4

É importante mencionar que acessar um *array* fora da sua área de indexação é um erro de lógica. Assim, se a indexação de um *array* é de 0 a 4, não podemos acessar as posições anteriores ao 0, nem posteriores ao 4.

Atividade 1

Atende aos Objetivos 1 e 2

1. Imagine que você foi contratado para fazer parte da automação de um hotel com 25 quartos, cada um deles com preços diferentes. Quais são os vetores, seus tamanhos e seus respectivos tipos de dados que você criaria para armazenar os nomes dos hóspedes e as tarifas de cada quarto?

2. Desenvolva um algoritmo para responder às seguintes perguntas: O quarto x está ocupado? Se sim, qual é o nome do hóspede? Quanto o hotel está faturando com a ocupação atual?

Resposta Comentada

1. São 25 quartos, com preços diferentes, logo, é necessário um vetor de 25 números reais para armazenar os preços *Real[]precos=Real[25]*. Também podem ser 25 hóspedes diferentes, logo, deve-se ter um vetor de 25 variáveis de textos para guardar os nomes *Texto[]hospedes=Texto[25]*.

2. Para calcular o faturamento, esses dois vetores mencionados são insuficientes. O faturamento pode ser dado pela equação:

$$\text{faturamento} = \sum_{i=1}^{25} (preçoQuarto_i \; x \; ocupado_i),$$

na qual $ocupado_i$ é uma variável que assume 1 se o quarto i está ocupado e 0 (zero) caso contrário. Em outras palavras, para calcular o faturamento é necessário saber quais quartos estão ocupados. Para isso, pode-se criar um vetor de 25 números inteiros, que pode ser usado diretamente na equação acima para calcular o faturamento ou, ainda, pode-se usar

um vetor de 25 variáveis lógicas, que assumem como verdadeiro caso o quarto esteja ocupado e como falso caso contrário. Devemos lembrar que, se você optar por usar um vetor de variáveis lógicas, deve fazer adaptações necessárias para calcular o faturamento, visto que variáveis lógicas não podem ser utilizadas em expressões aritméticas.

A **Figura 7.4** mostra um exemplo de uso da sintaxe de acesso. Nesse algoritmo, é criado um vetor chamado notas, de 10 posições. Em seguida, a cada posição, é atribuído um número múltiplo de 15, de modo que a posição de índice 0 contenha o valor 15, a posição de índice 1 contenha o valor 30 e assim sucessivamente, até a posição de índice 9, que contém o valor 150. Note que, apesar de os valores dos índices dos vetores serem numerados de 0 a 9, qualquer valor pode ser colocado como valor de um componente do vetor. Neste exemplo, foram usados números múltiplos de 15 , mas poderiam ser de 10 a 100 com passo 10, poderiam ter sido colocados todos os números ímpares de 1 a 20, ou qualquer outro valor válido. Por último, o algoritmo imprime cada um dos elementos dos vetores a partir de seus índices. Neste ponto, o algoritmo imprime o elemento do vetor notas no índice 0, em seguida imprime o elemento do índice 1 e assim sucessivamente até o índice 9.

```
Algoritmo acessoVetores()
Início
   Inteiro [ ] notas = Inteiro [10]
   Inteiro i
   Imprimir "preenchendo o vetor com múltiplos de 15"
   Para (i=0;i<10;i=i+1) Faça
      notas[i] = 15*(i+1)
   Fim Para
   Imprimir "Os números no vetor são:"
   Para (i=0;i<10;i=i+1) Faça
      Imprimir notas[i]
   Fim Para
Fim
```

Figura 7.4: Exemplo de acesso aos elementos de um vetor.

Como os elementos contidos no vetor são os números de 1 a 10, o algoritmo da **Figura 7.4** imprime a seguinte saída:

```
Os números no vetor são:
15
30
45
60
75
90
105
120
135
150
```

Exemplos de aplicação de vetores:

Exemplo 1: Uma empresa produz cinco produtos e quer saber qual deles representa o maior percentual de vendas. Faça um algoritmo que receba a quantidade vendida de cada produto, calcule o total de vendas e o percentual de cada produto nesse total. Esse percentual deve ser exibido.

Neste exemplo, são utilizadas estruturas de repetição juntamente com vetores para atingir os objetivos. Para tanto, a primeira coisa a ser feita é armazenar a quantidade vendida de cada produto. Para isso, pode ser utilizado um vetor de números inteiros. Uma vez feito isso, o total de vendas pode ser calculado a partir da soma dos elementos do vetor. Os percentuais podem ser obtidos através de uma regra de três. Uma possível solução para o problema utilizando as estratégias mencionadas é mostrada a seguir. Nela, a primeira estrutura de repetição trata da leitura dos valores vendidos e do cálculo do total de vendas. Os percentuais são calculados e exibidos na segunda estrutura de repetição.

```
Algoritmo percentuais()
Início
   Inteiro [] qtdVendas = Inteiro[5]
   Inteiro i, soma=0
   Para(i=0;i<5;i=i+1)Faça
      Imprimir "Digite a quantidade vendida do produto "+ (i+1)
      Ler qtdVendas[i]
      soma = soma+ qtdVendas[i]
   Fim Para
   Imprimir "O total de vendas é de "+ soma + " produtos"
   Imprimir "Os percentuais de cada produto são:"
   Para (i=0;i<5;i=i+1) Faça
      Real percent = (100.0* qtdVendas[i])/soma
      Imprimir "Produto "+(i+1)+ "->"+ percent
   Fim Para
Fim
```

Exemplo 2: Faça um algoritmo que leia e armazene em um vetor um conjunto de números. O algoritmo deve imprimir a soma desses números, quantos foram pares e quantos foram ímpares. O tamanho do conjunto deve ser informado pelo usuário.

Essa atividade ilustra bem como combinar estruturas de desvio, repetição e vetores. O primeiro passo deve ser criar um vetor com um tamanho de elementos informado pelo usuário. Em seguida, devemos interagir com o usuário para que ele insira cada um dos elementos do vetor. Por último, devemos calcular a soma e a quantidade de pares e ímpares. A seguir, encontra-se uma das possíveis soluções para esse problema. Nessa solução, o vetor *numeros* é utilizado para armazenar os números informados pelo usuários. A variável *tamanho* é utilizada como quantidade de elementos do conjunto. As linhas de 8 a 11 mostram a parte de leitura do conjunto numérico. Note que uma posição de um vetor pode ser lida, acessada e utilizada como uma variável comum, desde que a sintaxe seja utilizada corretamente. As linhas de 12 a 19 são as que calculam a soma e a quantidade de pares e ímpares. A linha 20 cuida da impressão dos resultados obtidos.

```
Algoritmo mediaParesImpares()
Início
   Inteiro [ ] numeros
   Inteiro tamanho,i,soma=0,pares=0,impares=0
   Imprimir "Digite o tamanho do conjunto a ser lido"
   Ler tamanho
   numeros = Inteiro[tamanho]
   Para (i=0;i<tamanho;i=i+1) Faça
      Imprimir "digite um número"
      Ler numeros[i]
   Fim Para
   Para (i=0;i<tamanho;i=i+1) Faça
      soma=soma+numeros[i]
      Se (numeros[i]%2==0)Então
         pares = pares+1
      Senão
         impares = impares+1
      Fim Se
   Fim Para
   Imprimir "a soma é: "+ soma + " pares: "+pares + " impares "+impares
Fim
```

Como pôde ser observado ao longo da aula, o armazenamento de dados em *arrays* possibilita compor algoritmos para problemas mais complexos do que os resolvidos até então. Além disso, os *arrays* têm um papel importante na simplificação de certos algoritmos e na criação de

estruturas de dados mais complexas, como *heaps,* tabelas *Hash* e árvores binárias, que você estudará em outras disciplinas. Portanto, fica claro que *arrays* são estruturas fundamentais para a composição de algoritmos e para a solução de diversos tipos de problemas.

Atividade Final

Atende aos Objetivos 1 e 2

1. Faça um algoritmo que calcule a série de Fibonacci com o auxílio de um vetor.

2. Faça um algoritmo que receba dois 2 vetores de 5 posições e calcule a intercalação entre eles. Suponha que não há elementos repetidos. Ex: Sejam os vetores a={1,3,5,4,7} e b={−1,0,2,6,8}. As duas intercalações possíveis são: ab={1,−1,3,0,5,2,4,6,7,8} e ba={−1,1,0,3,2,5,6,4,8,7}.

3. Faça um algoritmo que receba os dados geométricos de 2 vetores em R^3 e calcule a soma vetorial desses 2 vetores.

A equação da soma vetorial em R^3 é $s_i = u_i + v_i \forall i \in \{1,2,3\}$, na qual u e v são os vetores de entrada e s o vetor resultante.

4. Faça um algoritmo que receba os dados geométricos de 2 vetores em R^3 e calcule o produto escalar desses 2 vetores. A equação do produto escalar, denotado *pe*, é dada por: $pe = \sum_{i=1}^{3} u_i \times v_i = u_1 \times v_1 + u_2 \times v_2 \times u_3 \times v_3$.

5. Faça um algoritmo que receba, uma a uma, as letras de uma palavra e determine se essa palavra é um palíndromo. Obs: um palíndromo é uma palavra que, quando escrita de trás para frente, permanece igual. Exemplos: ovo, radar.

6. Faça um algoritmo que leia vários números digitados pelo usuário. O algoritmo deve armazenar esses números em um vetor, de modo que, durante a digitação, o vetor sempre esteja em ordem crescente. A quantidade de números a serem inseridos também deve ser informada pelo usuário.

7. Faça um algoritmo que receba um conjunto de números e calcule a variância desse conjunto. A variância é dada por: $1/n * \sum_{i=0}^{n-1} = ([i]-\mu)^2$, em que μ é a média do conjunto e n é o número de elementos. Teste o seu algoritmo para a sequência {1,2,3,4,5,6}. O resultado deve se aproximar de 2,9.

8. Faça um algoritmo que leia os elementos de um vetor e coloque esses elementos em uma ordem crescente ou decrescente (fica a seu critério escolher qual das duas ordens é mais conveniente).

Resposta Comentada

1. Esta atividade é apenas ilustrativa, uma vez que já foi visto que a série de Fibonacci pode ser calculada sem o uso de vetores. Contudo, o uso de vetores facilita muito o raciocínio sobre o problema. A seguir está uma possível solução para a questão. Primeiramente são criados um vetor para conter os termos da série e duas variáveis inteiras. A variável *tamanho* representa a quantidade de termos da série que devemos calcular, logo, o vetor *numeros* deve ter um número de posições adequado a *tamanho*, o que é feito na linha 7 do algoritmo. Os dois primeiros termos da série são conhecidos da definição matemática (fibonacci(n) resulta em 1 se n = 1 ou n = 2; fibonacci(n) resulta em fibonacci(n−1) + fibonacci(n−2) caso contrário). Assim, as posições de índice *0* e *1* (primeira e segunda posições) são inicializadas com 1. Desse modo, é preciso calcular os valores da série do terceiro termo em diante, por isso a estrutura de repetição *Para* faz a inicialização da variável *i* em 2 (terceira posição). Nesta primeira estrutura de repetição, o termo atual da série (*i*) é calculado em função dos dois termos anteriores (*i−1* e *i−2*). A segunda estrutura de repetição é utilizada para imprimir os termos calculados.

```
Algoritmo fibonacciVetor()
Início
   Inteiro [ ] numeros
   Inteiro tamanho,i
   Imprimir "Digite o termo de a ser encontrado na série"
   Ler tamanho
   numeros = Inteiro[tamanho]
   numeros[0]=1
   numeros[1]=1
   Para (i=2;i<tamanho;i=i+1) Faça
      numeros[i]=numeros[i-1]+numeros[i-2]
   Fim Para
   Para (i=0;i<tamanho;i=i+1) Faça
      Imprimir "Termo "+(i+1)+ " = "+numeros[i]
   Fim Para
Fim
```

2. A intercalação consiste em colocar as posições de índice iguais como vizinhas no vetor resultante. Uma possível solução é dada a seguir. Nela, são criados 3 vetores: *vet1* e *vet2* são os vetores de entrada e *vet3* é o vetor em que será armazenada a intercalação de *vet1* e *vet2*. Nas linhas de 5 a 12 é feita a leitura dos dados. Nas linhas de 13 a 16 é feita a operação de intercalação. Observe que, apesar de *vet3* ter 10 posições, podemos endereçar todas as 10 através de expressões matemáticas em função das posições em *vet1* e *vet2*. As demais linhas mostram o resultado obtido.

```
Algoritmo intercalacao()
Início
   Inteiro [ ] vet1 = Inteiro [5], vet2 = Inteiro[5], vet3 = Inteiro[10]
   Inteiro i
   Imprimir "Digite os termos do primeiro vetor"
   Para (i=0;i<5;i=i+1) Faça
      Ler vet1[i]
   Fim Para
   Imprimir "Digite os termos do segundo vetor"
   Para (i=0;i<5;i=i+1) Faça
      Ler vet2[i]
   Fim Para
   Para (i=0;i<5;i=i+1) Faça
      vet3[2*i]=vet1[i]
      vet3[2*i+1]=vet2[i]
   Fim Para
   Para (i=0;i<10;i=i+1) Faça
      Imprimir "posição "+i+ " = "+vet3[i]
   Fim Para
Fim
```

3. A primeira coisa a se observar é a diferença entre os conceitos de vetor geométrico (às vezes chamado de vetor algébrico ou simplesmente vetor) e vetor computacional (*arrays*). A complicação vem do fato de que, na língua-mãe da computação – a língua inglesa – essas duas entidades, têm nomes completamente distintos. Um vetor geométrico é chamado de *vector* e um vetor computacional, de *array*. Em português elas têm o mesmo nome (vetor). Devido a isso, para esta questão e para a questão 4, faremos uma distinção de termos. Quando tratarmos dos vetores geométricos, utilizaremos o termo *vetor*. Quando tratarmos dos vetores computacionais, utilizaremos o termo *array*. Um vetor é uma entidade que possui associados um valor, uma direção e um sentido. Um *array* é só um conjunto de valores (mais informações sobre vetores em STEINBRUCH; WINTERLE, 1987). Entretanto, uma das formas de se representar um vetor é através de suas coordenadas cartesianas. Para vetores em R^3, são necessários 3 valores, um para x, um para y e um para z. Também é

sabido que a soma de dois vetores é outro vetor, e que as notações *u1, u2, u3* expressam as coordenadas *x,y,z* do vetor *u*. Assim, são necessários 3 vetores de três posições: dois para serem entrada e um para ser o resultado. Uma possível solução é dada a seguir. Nela, são criados 3 *arrays* de 3 posições: *vet1, vet2* (entrada) e *vet3* (resultado). Ou seja, usaremos um *array* para representar um vetor através do armazenamento de suas coordenadas cartesianas. As linhas de 5 a 12 tratam da leitura dos dados. As linhas 13, 14 e 15 implementam a equação da soma vetorial na qual o vetor resultante na posição *i* é igual à soma das posições *i* dos vetores que se quer somar. As demais linhas tratam da impressão do resultado obtido.

```
Algoritmo somaVetorial()
Início
   Real [ ] vet1 = Real [3], vet2 = Real[3], vet3 = Real[3]
   Inteiro i
   Imprimir "Digite as coordenadas do primeiro vetor"
   Para (i=0;i<3;i=i+1) Faça
      Ler vet1[i]
   Fim Para
   Imprimir "Digite as coordenadas do segundo vetor"
   Para (i=0;i<3;i=i+1) Faça
      Ler vet2[i]
   Fim Para
  Para (i=0;i<3;i=i+1) Faça
      vet3[i]=vet1[i]+vet2[i]
   Fim Para
   Para (i=0;i<3;i=i+1) Faça
      Imprimir "posição "+i+ " = "+vet3[i]
   Fim Para
Fim
```

4. Para esta questão, também são válidas as observações feitas na questão 3 a respeito de vetor geométrico e *array*. Uma possível solução é dada a seguir. A primeira observação a ser feita é que, diferentemente da questão anterior, o produto escalar entre dois vetores não é um vetor, e sim um número, logo, apenas dois vetores de entrada são necessários. As linhas de 1 a 13 cuidam da criação das variáveis e da leitura das entradas. As linhas 14, 15 e 16 cuidam da implementação da equação, que é um somatório dos produtos das coordenadas de mesma posição dos vetores. A linha 17 mostra o resultado obtido.

```
Algoritmo produtoEscalar()
Início
   Real [ ] vet1 = Real [3], vet2 = Real[3]
   Real resultado=0
   Inteiro i
   Imprimir “Digite as coordenadas do primeiro vetor”
   Para (i=0;i<3;i=i+1) Faça
      Ler vet1[i]
   Fim Para
   Imprimir “Digite as coordenadas do segundo vetor”
   Para (i=0;i<3;i=i+1) Faça
      Ler vet2[i]
   Fim Para
  Para (i=0;i<3;i=i+1) Faça
      resultado=resultado + vet1[i] * vet2[i]
   Fim Para
   Imprimir “O Produto Escalar é:”+resultado
Fim
```

5. Existem vários modos de se resolver esta questão. Um deles é o uso explícito de vetores para armazenar cada uma das letras. Essa solução é mostrada a seguir. As linhas de 1 a 10 cuidam da inicialização das variáveis e da leitura dos dados. Na linha 11 é criada uma variável *dif* que é inicializada com 0. Esta é uma variável que será usada como marcação. A estratégia é simples: percorrer o vetor de letras comparando a primeira com a última, a segunda com a penúltima e assim sucessivamente até a metade do vetor. Caso alguma dessas comparações indique letras diferentes, a variável de marcação terá seu valor alterado, indicando que existe uma diferença. Ao final da comparação, se *dif* mantiver o valor zero, significa que nenhuma diferença foi encontrada, logo, a palavra é um palíndromo.

```
Algoritmo palindromoVetorImplicito()
Início
   Inteiro i=0,tam, dif=0
   Texto palavra
   Imprimir “Digite uma palavra e a quantidade de letras que esta palavra tem”
   Ler palavra,tam
   Enquanto(i<tam/2 & dif==0) Faça
      Se (palavra[i] != palavra[tam - i -1]) Então
        dif=1
      Fim Se
      i = i+1
   Fim Enquanto
  Se (dif==0) Então
    Imprimir “É palíndromo”
  Senão
    Imprimir “Não é palíndromo”
  Fim Se
Fim
```

O modo mais racional de resolver esta questão é aplicar a ela a mesma lógica apresentada no parágrafo anterior, mas, em vez de utilizar explicitamente um vetor, utilizar os operadores de texto apresentados na Aula 2. Outra alteração a ser feita é o número de comparações. Ao nos depararmos com uma diferença, não é necessário fazer as demais comparações, visto que já foi detectado que a palavra não é um palíndromo. Uma possível solução que utiliza essas observações é mostrada a seguir. Tais observações simplificam o uso de variáveis do tipo texto, uma vez que fica claro que elas são, de fato, *arrays*.

6. Para manter um vetor ordenado durante o processo de leitura, deve-se ler o novo elemento e executar duas tarefas: a) encontrar a posição correta para armazená-lo, ou seja, devemos encontrar a posição do primeiro número maior que ele; b) deslizar todos os números, a partir da posição encontrada, uma casa para a direita. Isso abrirá espaço para a inserção do novo número. Uma solução que implementa essa estratégia é mostrada a seguir. As linhas de 1 a 6 tratam da criação e da inicialização das variáveis. A estrutura *Para* da linha 7 do algoritmo serve para contar quantos elementos foram lidos até o presente momento. As linhas 8 e 9 tratam da leitura do novo número a ser inserido. As linhas de 10 a 15 servem para encontrar a posição do primeiro número maior que o valor de *x*. Note que, se nenhum número foi digitado ainda, essa posição é zero. As linhas 16, 17 e 18 tratam do deslizamento dos números uma casa para a direita. A linha 19 insere *x* na posição correta. As demais linhas exibem o resultado.

```
Algoritmo armazenaOrdenado()
Início
   Inteiro tamanho,i,j,x,pos
   Imprimir "Digite a quantidade de números"
   Ler tamanho
   Inteiro[] num = Inteiro[tamanho]
   Para (i=0;i<tamanho;i=i+1) Faça
      Imprimir "Digite os números"
      Ler x
      j=0
      Enquanto (j< i & num[j] <= x) Faça
         j = j+1
      Fim Enquanto
      pos=j
      Para (j=i-1; j>=pos; j=j-1) Faça
          num[j+1]=num[j]
      Fim Para
      num[pos]=x
   Fim Para
   Para (i=0;i<tamanho;i=i+1) Faça
      Imprimir "posição "+i+ " = "+num[i]
   Fim Para
Fim
```

7. O problema da variância já foi detalhado na Introdução desta aula. Uma possível solução para ele é mostrada a seguir. Nela, as linhas de 1 a 10 cuidam da criação das variáveis e da leitura dos dados. As linhas de 11 a 14 tratam do cálculo da média. As linhas de 15 a 18 calculam a variância.

```
Algoritmo variancia()
Início
   Inteiro tam,i
   Real med=0, var=0
   Imprimir "Digite a quantidade de números"
   Ler tam
   Real [] num = Real[tam]
   Para (i=0;i<tam;i=i+1) Faça
      Ler num[i]
   Fim Para
   Para (i=0;i<tam;i=i+1) Faça
      med = med+num[i]
   Fim Para
   med = med/tam
   Para (i=0;i<tam;i=i+1) Faça
      var = var + (num[i]-med)*(num[i]-med)
   Fim Para
   var = var/tam
   Imprimir "A variância é: "+var
Fim
```

8. A diferença entre esta questão e a questão 6 é a ordem em que as coisas devem ser feitas. Na questão 6, a leitura e a ordenação deviam ser feitas simultaneamente. Nesta questão, os dados são lidos primeiro e só depois deve-se ordenar o vetor. Desse modo, uma possível solução é apresentada a seguir. Ela percorre o *array*, analisando as posições vizinhas. Sempre que encontra dois números vizinhos em posição trocada, o algoritmo corrige essas posições (linhas 11 a 17). Para garantir a ordenação, é necessário realizar esse percurso *n* vezes, em que *n* é o tamanho do vetor. Por isso, as linhas de 11 a 17 precisam ser repetidas. Isso é feito pela estrutura *Para*, que começa em 10 e termina em 18. Essa estratégia é chamada método bolha de ordenação, pois se baseia no princípio físico de líquidos não solúveis com densidades diferentes. Estes, quando misturados, separam-se em camadas, sendo que as camadas mais pesadas ficam no fundo e as mais leves ficam mais próximas à superfície. A ideia é fazer com que elementos menores migrem para a esquerda e elementos maiores, para a direita, à medida que o algoritmo executa.

```
Algoritmo ordenar()
Início
   Inteiro tam,i,j
   Imprimir "Digite a quantidade de números"
   Ler tam
   Real [] num = Real[tam]
   Para (i=0;i<tam;i=i+1) Faça
      Ler num[i]
   Fim Para
   Para (i=0;i<tam;i=i+1) Faça
      Para (j=0;j<tam-1;j=j+1) Faça
        Se(num[j]>num[j+1])Então
           Real aux = num[j]
           num[j] = num[j+1]
           num[j+1] = aux
        Fim Se
     Fim Para
   Fim Para
    Para (i=0;i<tamanho;i=i+1) Faça
      Imprimir "posição "+i+ " = "+num[i]
   Fim Para
Fim
```

Resumo

Nesta aula você aprendeu a declarar e acessar vetores. Viu que os vetores são conjuntos de variáveis de um mesmo tipo e que é possível acessá-los por meio do nome do vetor e de um índice que indica sua posição no conjunto. Também foi mostrado como compor algoritmos combinando vetores, estruturas de repetição e de desvio, a fim de resolver problemas mais complexos.

Informações sobre a próxima aula

Na próxima aula, você verá como lidar com armazenamento e manipulação de dados em matrizes.

Aula 8

Matrizes

Tiago Araújo Neves

Meta

Expor os conceitos de matrizes, mostrando sua sintaxe e funcionamento.

Objetivos

Esperamos que, ao final desta aula, você seja capaz de:

1. definir e aplicar o conceito de matrizes;

2. fazer algoritmos que combinam estruturas de desvio e repetição juntamente com matrizes para resolver problemas;

3. representar matrizes graficamente quando possível e logicamente, independente do número de dimensões.

Introdução

Em muitos casos, tratar armazenamento com uma indexação única não é conveniente. Um exemplo típico são os calendários utilizados no dia a dia. Em suas linhas, o calendário de um dado mês armazena os dias do mês que estão na mesma semana. Nas colunas, estão os dias do mês que caem no mesmo dia da semana. A **Figura 8.1** mostra um exemplo de calendário. Note que, ao inspecionar a última linha da tabela, verificamos todos os dias que pertencem à ultima semana do mês em questão. Ao inspecionarmos a sexta coluna, encontramos todos os dias que caem em um sábado.

Segunda-feira	Terça-feira	Quarta-feira	Quinta-feira	Sexta-feira	Sábado	Domingo
						1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31					

Figura 8.1: Calendário, exemplo de matriz.

Nesse caso, a representação por tabela ou matriz é mais conveniente do que uma representação por vetor. Matrizes também são muito naturais em diversos problemas da Matemática e da Engenharia e, por isso, seu estudo é objeto de diversas disciplinas/ áreas. Para exemplificar, matrizes são utilizadas na resolução de sistemas de equações, no processamento de imagens, em simulações, geoprocessamentos e em muitas outras situações. Devido a isso, é de fundamental importância saber como construir algoritmos capazes de manipular matrizes, uma vez que muitas das tarefas mencionadas podem ser feitas manualmente em alguns casos mas, na maioria deles, inclusive nos mais complexos, é necessário o uso de computadores.

Matrizes: declaração e acesso

Matrizes ou *arrays* multidimensionais são um conjunto identificável de posições de memória, em que se podem armazenar informações de um tipo específico. Elas podem ser vistas como um conjunto de elementos do mesmo tipo, no qual é possível identificar, acessar e alterar cada elemento. A diferença entre uma matriz e um *array* (ou vetor) é o número de indexações necessárias para se fazer o acesso. Como o *array* só possui uma dimensão, apenas um índice é necessário. Já com as matrizes, é preciso um índice para cada dimensão, ou seja, para acessar as informações de uma matriz de duas dimensões, são necessários dois índices; para uma matriz de três dimensões, três índices, e assim sucessivamente.

A declaração de matrizes segue a seguinte sintaxe:

tipo [ ] [ ] nome

tipo [ ] [ ] [ ] nome1, nome2,..., nomen

A primeira linha é utilizada para declarar uma única matriz de duas dimensões. A segunda é utilizada para declarar várias matrizes de três dimensões. Nessa sintaxe, nome1, nome2 etc. são os nomes das matrizes que serão criados e pode haver quantos nomes forem necessários, desde que eles sejam separados por vírgulas. Cada par de colchetes ("[" e "]") indica uma dimensão. Logo, para matrizes de três dimensões são necessários três pares de colchetes e, assim, não há limite para o número de dimensões. O tipo indica a natureza da informação que estará contida nas matrizes; qualquer tipo válido pode ser usado.

Para exemplificar o uso de matrizes, considere a seguinte situação: uma empresa tem quatro vendedores, que vendem cinco produtos. No final do dia, cada vendedor entrega uma nota de cada tipo de produto diferente vendido. Cada nota contém: 1) número do vendedor; 2) número do produto e 3) valor total vendido desse produto em reais. Faça um algoritmo que leia as notas de cada produto para cada vendedor e calcule a comissão de cada um deles. A comissão é calculada pela seguinte equação:

Comissão=10%*(totalProd1)+20%*(totalProd2)+10%*(totalProd3)+20%*(totalProd4)+10%*(totalProd5).

Essa situação é típica para o uso de matrizes. Note que a situação descreve que cada vendedor entrega uma nota para cada produto, mas nada

é mencionado a respeito da ordem de processamento das notas. Se fosse garantido que todas as notas de um mesmo vendedor fossem processadas em conjunto, seria possível conseguir calcular a comissão de cada um facilmente, mas isso não necessariamente acontece. O funcionário do setor financeiro pode processar as notas em ordem aleatória ou agrupando-as por produto. Nesse sentido, devem-se armazenar os dados em uma matriz que contém os dados de todos os vendedores para cada produto. Só depois de ter armazenado todos os dados é possível calcular a comissão de cada funcionário. A solução desse problema é descrita detalhadamente mais adiante. Por enquanto, vamos aprender a declarar corretamente as matrizes e utilizá-las corretamente em algoritmos.

A **Figura 8.2** mostra um exemplo de declaração, preenchimento e impressão de matriz. Uma matriz de duas dimensões com três linhas e três colunas é criada na linha três do algoritmo. As linhas de 5 a 9 tratam do preenchimento da matriz. Note que, assim como nos vetores, os índices das matrizes começam em zero. Desse modo, os índices das linhas e das colunas, neste exemplo, vão de 0 a 2. As linhas de 10 a 14 tratam da impressão dos elementos da matriz. Já a **Figura 8.3** mostra o mesmo algoritmo com cada uma de suas partes destacadas. Observe:

```
Algoritmo declararMatrizes()
Início
   Real [] [] num = Real [3][3]
   Inteiro i,j
   Para (i=0;i<3;i=i+1) Faça
     Para(j=0;j<3;j=j+1)Faça
       num[i][j]=i*3+j+1
     Fim Para
   Fim Para
   Para (i=0;i<3;i=i+1) Faça
     Para(j=0;j<3;j=j+1)Faça
       Imprimir num[i][j]
     Fim Para
   Fim Para
Fim
```

Figura 8.2: Exemplo de declaração, preenchimento e impressão de uma matriz de números reais com duas dimensões.

```
Algoritmo declararMatrizes()
Início
  Real[][]num=Real [3][3]
  Inteiro i,j
  Para(i=0;i<3;i=i+1)Faça
    Para(j=0;j<3;j=j+1)Faça
          num[i][j]=i*3+j+1
        Fim Para
  Fim Para
  Para(i=0;i<3;i=i+1)Faça
    Para(j=0;j<3;j=j+1)Faça
      Imprimir num[i][j]
    Fim Para
  Fim Para
Fim
```

Uma matriz de duas dimensões com três linhas e três colunas é criada.

A matriz é preenchida. Note que, assim como nos vetores, os índices das matrizes começam em zero. Deste modo, os índices das linhas e das colunas, neste exemplo, vão de 0 a 2.

Impressão dos elementos da matriz.

Figura 8.3: Exemplo de declaração, preenchimento e impressão de uma matriz de números reais com duas dimensões com partes destacadas.

Note que, apesar de imprimir todos os elementos da matriz, os algoritmos das **Figuras 8.2** e **8.3** imprimem os elementos como se fossem uma única coluna. Isso ocorre porque cada comando de impressão sempre utiliza uma linha. Na repetição das linhas 11 a 13, são impressos, para um dado *i*, todos os *j* possíveis, ou seja, três valores são impressos para cada *i*, tomando, então, três linhas. Como os múltiplos valores de *i* são tratados pela repetição que começa na linha 10 e são tratados três possíveis valores para *i*, temos que a repetição das linhas de 11 a 13 é executada três vezes. Logo, são impressos 3x3=9 elementos no total. A impressão do algoritmo da **Figura 8.2** é exibida do seguinte modo:

```
1
2
3
4
5
6
7
8
9
```

Observe que esse não é o modo usual utilizado para exibir uma matriz de duas dimensões. Fica difícil diferenciar uma matriz impressa dessa maneira de um vetor com nove posições. Para podermos visualizar as nuances de uma matriz, o ideal é representá-la com linhas e colunas, como é feito tradicionalmente na Matemática. As **Figuras 8.4** e **8.5** mostram como se faz isso. Note que a parte inicial do algoritmo é igual ao algoritmo das **Figuras 8.2** e **8.3**: a diferença está nas repetições que tratam da impressão. O comando *imprimir* nas **Figuras 8.4** e **8.5** não está dentro da repetição interna, como nas **Figuras 8.2** e **8.3**, mas dentro da repetição externa. Isso faz com que somente três impressões sejam feitas (uma para cada valor de *i*). O ponto crítico é montar corretamente o que será impresso a cada chamada. Observe que na linha 11 é criada uma variável de texto chamada *t*, que é inicializada com um texto vazio (“”). Em seguida, para cada *j* é concatenado a *t* o valor contido na matriz na posição de índices *i* e *j* e, em seguida, é concatenado um espaço em branco. Isso faz com que, na primeira execução, seja inserido o elemento *num[0][0]*, seguido de um espaço, ou seja, *t* valerá “1”. Depois, o valor contido em *num[0][1]* é concatenado ao valor de *t*, seguido de outro espaço, ou seja, *t* valerá “1 2”. O mesmo ocorre para *num[0][2]*, onde *t* se torna “1 2 3”. Note que, nesse ponto, a repetição em *j* termina e a variável *t* é impressa. Observe também que isso faz com que toda a primeira linha da matriz seja impressa de uma só vez. Quando *i* muda seu valor para 1, a variável *t* é recriada com um texto vazio. Essa variável será preenchida usando os elementos da matriz seguindo o valor atual de *i* (1) e todos os valores de *j*. Como *i* agora é 1, ao terminar a repetição em *j*, a variável *t* conterá os elementos da segunda linha.

```
Algoritmo declararMatrizes2()
Início
   Real [] [] num = Real [3][3]
   Inteiro i,j
   Para (i=0;i<3;i=i+1) Faça
     Para(j=0;j<3;j=j+1)Faça
       num[i][j]=i*3+j+1
     Fim Para
   Fim Para
   Para (i=0;i<3;i=i+1) Faça
     Texto t= “”
     Para(j=0;j<3;j=j+1)Faça
        t = t + num[i][j] + “ ”
     Fim Para
     Imprimir t
   Fim Para
Fim
```

Figura 8.4: Algoritmo com impressão de matriz no formato linha × coluna.

```
Algoritmo declararMatrizes2()
Início
  Real[][]num=Real [3][3]
  Inteiro i,j
  Para(i=0;i<3;i=i+1)Faça
    Para(j=0;j<3;j=j+1)Faça
          num[i][j]=i*3+j+1
        Fim Para
  Fim Para
  Para(i=0;i<3;i=i+1)Faça
        Texto t=""
    Para(j=0;j<3;j=j+1)Faça
      t=t+num[i][j]+" "
        Fim Para
        Imprimir t
  Fim Para
Fim
```

O comando imprimir não está dentro da repetição interna como na Figura 8.2, e sim dentro da repetição externa. Isso faz com que somente três impressões sejam feitas (uma para cada valor de *i*).

A repetição em *j* define o que será impresso em cada linha. A variável *t* é usada para concatenar os diferentes elementos contidos na matriz nas diferentes posições de *j* para um mesmo *i*.

Figura 8.5: Algoritmo de impressão de matriz no formato linha × coluna com partes destacadas.

A saída do algoritmo da **Figura 8.3** é mostrada a seguir. Note que este é um modo bem mais inteligível para se visualizar uma matriz.

```
1 2 3
4 5 6
7 8 9
```

Assim como em *arrays* de uma dimensão, acessar uma posição fora da área de indexação em matrizes também é um erro de lógica. Isso é válido para qualquer dimensão, ou seja, não adianta acessar corretamente as linhas de uma matriz de duas dimensões se, ao acessar as colunas, utiliza-se um índice *–1*, por exemplo. Mesmo que apenas uma das várias dimensões de uma matriz seja acessada erroneamente, o algoritmo está errado.

Atividade 1

Atende aos Objetivos 1 e 2

Faça um algoritmo que calcule a soma dos elementos da diagonal principal de uma matriz 5x5 de números reais.

Resposta Comentada

Os primeiros passos para resolver essa questão é declarar as variáveis necessárias e fazer a leitura dos elementos da matriz. Como estes são números reais, a soma dos elementos da diagonal principal também deve ser declarada como real. Uma vez feitas as declarações e as leituras, é necessário fazer a soma. Note que, na diagonal principal de uma matriz, o índice da linha é sempre igual ao índice da coluna. Assim, para controlar a mudança entre os elementos, apenas uma estrutura de repetição é necessária. A seguir, é mostrada uma possível solução para o problema. Nela, a soma é feita pela repetição das linhas 12, 13 e 14, em que à variável soma é adicionado o valor da matriz na posição {i, i}. Ou seja, na primeira iteração, será adicionada à soma a posição {0,0}, depois a posição {1,1} e assim sucessivamente. No final, o resultado da soma dos elementos da diagonal principal é exibido.

```
Algoritmo diagonalPrincipal()
Início
   Inteiro i,j
   Real [] [] num = Real [5][5]
   Real soma =0
   Imprimir “Digite os elementos da matriz 5x5”
   Para (i=0;i<5;i=i+1) Faça
     Para(j=0;j<5;j=j+1)Faça
       Ler num[i][j]
     Fim Para
   Fim Para
   Para (i=0;i<5;i=i+1) Faça
      soma = soma+ num[i][i]
   Fim Para
   Imprimir “A soma é ”+ soma
Fim
```

Atividade 2

Atende aos Objetivos 1 e 2

Faça um programa que calcule e imprima a soma dos elementos das colunas de uma matriz de números inteiros.

Resposta Comentada

Existem vários mecanismos para resolver essa questão. Um dos mais sofisticados é utilizar, além da matriz mencionada, um vetor para armazenar a soma de cada uma das colunas. A seguir é apresentada uma possível solução que usa esta estratégia para esse problema. As linhas de 1 a 13 tratam da declaração e da inicialização das variáveis. Note que, como deseja-se utilizar o vetor para armazenar a soma de cada coluna, esse vetor é inicializado com um número de posições equivalente ao número de colunas da matriz. As linhas de 14 a 19 tratam do preenchimento do vetor. Note que, como o que se deseja é somar as colunas, a repetição que controla as linhas é posicionada internamente àquela que controla as colunas. Veja também que, para cada coluna a ser somada, a posição correspondente do vetor é zerada (linha 15) e, em seguida, alterada através de sucessivas operações de soma. As demais linhas tratam da impressão do vetor calculado. Observe que, nas primeiras linhas, a manipulação do vetor era feita com a variável *j* e que agora, durante a impressão, essa manipulação é feita com a variável *i*. Isso pode ser feito sem nenhum problema, desde que se tome o devido cuidado com a inicialização, condição e passo da repetição.

```
Algoritmo somaColuna()
Início
   Inteiro i,j,l,c
   Imprimir "Digite o número de linhas e colunas das matrizes"
   Ler l,c
   Inteiro [] [] mat = Inteiro [l][c]
   Inteiro [ ]  vet = Inteiro[c]
   Imprimir "Digite os elementos da matriz"
   Para (i=0;i<l;i=i+1) Faça
     Para(j=0;j<c;j=j+1)Faça
       Ler mat[i][j]
     Fim Para
   Fim Para
   Para (j=0;j<c;j=j+1) Faça
     vet[j]=0
     Para(i=0;i<l;i=i+1)Faça
       vet[j]=vet[j]+ mat[i][j]
     Fim Para
   Fim Para
   Imprimir "A soma das colunas são: "
   Para (i=0;i<c;i=i+1) Faça
      imprimir vet[i]
   Fim Para
Fim
```

Atividade 3

Atende aos Objetivos 1 e 2

Uma empresa tem quatro vendedores que vendem cinco produtos. No final do dia, cada vendedor entrega uma nota de cada tipo de produto diferente. Cada nota contém: 1) número do vendedor; 2) número do produto e 3) valor total vendido desse produto em reais. Considere que, mesmo que não tenha vendido nada, o vendedor deve preencher uma nota com o valor total vendido igual a zero.

Faça um algoritmo que leia as notas de cada produto para cada vendedor e calcule a comissão de cada um deles. A comissão é calculada pela seguinte equação:

Comissão=10%*(totalProd1)+20%*(totalProd2)+10%*(totalProd3)+20%*(totalProd4)+10%*(totalProd5).

Resposta Comentada

Para resolver essa tarefa, deve-se utilizar uma matriz bidimensional e estabelecer o que as linhas e o que as colunas representarão. As linhas podem ser usadas para representar vendedores, e as colunas, produtos, ou vice-versa. Uma possível solução utilizando as linhas como vendedores é apresentada a seguir. Nela, são criadas duas variáveis auxiliares e uma matriz de quatro linhas por cinco colunas (uma linha para cada vendedor e uma coluna para cada produto). Desse modo, para calcular a comissão de um vendedor, deve-se percorrer a linha correspondente a ele. Cada coluna dessa linha representa o montante vendido, em reais, para um determinado produto. Note que a equação da comissão apresenta 10% para o total dos produtos ímpares e 20% para os produtos pares. Contudo, a equação apresenta produtos que começam em 1 e vão até 5. A matriz que está sendo utilizada possui indexação de colunas começando em 0 e indo até 4. Nesse sentido, é preciso fazer uma pequena adaptação, visto que os dados referentes ao produto 1 estarão armazenados na coluna 0, os dados do produto 2 estarão armazenados na coluna 1, e assim sucessivamente, até os dados do produto 5, que estarão armazenados na coluna 4. Essa adaptação pode ser vista nas linhas de 13 a 17 do algoritmo,

onde os produtos armazenados em colunas pares são multiplicados por 10% e os armazenados em colunas ímpares são multiplicados por 20%. Ajustes similares também foram feitos para que os dados do vendedor 1 estejam armazenados na linha 0 da matriz, e os dados do vendedor 4 sejam armazenados na linha 3. Note que a solução permite que os dados sejam inseridos fora da ordem e, só depois de inseridos os dados de todas as 20 notas (5 produtos x 4 vendedores), o algoritmo começa a calcular as comissões.

```
Algoritmo comissao()
Início
   Inteiro i,j,k
   Real [] [] m = Real [4][5]
   Para (k=0;k<20;k=k+1) Faça
     Imprimir "Digite o número do vendedor, do produto e o valor vendido"
     Ler i,j
     Ler m[i-1][j-1]
   Fim Para
   Para (i=0;i<4;i=i+1) Faça
     Real com =0
     Para(j=0;j<5;j=j+1)Faça
        Se(j%2==0)Então
           com=com+m[i][j]*10.0/100
        Senão
           com=com+m[i][j]*20.0/100
        Fim Se
     Fim Para
     Imprimir "comissão do vendedor "+(i+1)+"="+com
   Fim Para
Fim
```

Matrizes de mais de três dimensões

Como mencionado, não há limites para o número de dimensões que podem ser utilizadas em uma matriz. Entretanto, frequentemente as pessoas têm dificuldade para visualizar matrizes com mais de três dimensões. Durante o Ensino Médio, nos habituamos a pensar em matrizes com duas e três dimensões. Para visualizá-las, criamos mentalmente associações com tabelas (duas dimensões) e cubos (três dimensões). Porém, visualizar uma estrutura com mais de três dimensões, apesar de ser uma tarefa complicada, às vezes é algo fundamental para a solução de problemas relacionados à indexação. Uma analogia que é comumente feita para

ajudar na visualização desse tipo de matrizes é a associação com *arrays* e tabelas de acordo com a quantidade de dimensões. Caso o número de dimensões seja um, tem-se um *array*. Caso o número de dimensões seja dois, tem-se uma tabela. Até aqui, nenhuma novidade. Para matrizes com três ou mais dimensões, aplicamos a seguinte regra: se o número de dimensões da matriz for ímpar, visualizamos essa matriz como um vetor em que cada posição é uma tabela. Se o número de dimensões do vetor for par, cada posição da tabela é uma outra tabela. As **Figuras 8.6** e **8.7** representam graficamente essa analogia. Note que, na **Figura 8.6**, uma matriz de três dimensões é representada como se fosse um vetor, em que cada posição é uma tabela. A **Figura 8.7** mostra uma matriz de quatro dimensões. Nessa figura, a matriz é mostrada como uma tabela, em que cada posição é uma outra tabela (também de duas dimensões). Para o caso de uma matriz de cinco dimensões, uma lógica envolvendo a junção das duas ideias pode ser utilizada. Uma matriz de cinco dimensões pode ser vista como um vetor, em que cada entrada é uma tabela. Nessa tabela, cada entrada é uma outra tabela. Um raciocínio similar pode ser aplicado para matrizes de seis, sete, *n* dimensões, devendo-se sempre tomar cuidado para diferenciar se o número de dimensões é par ou ímpar.

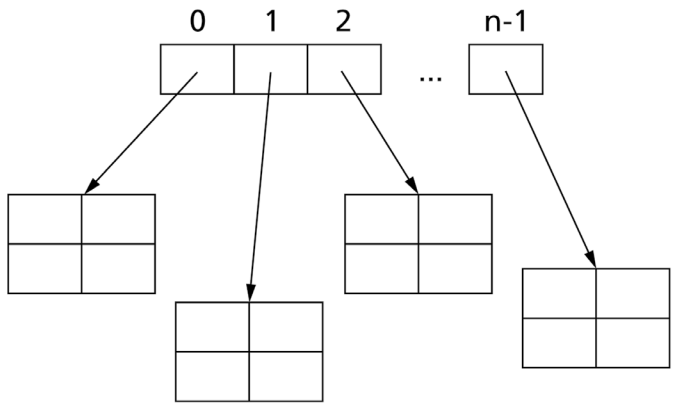


Figura 8.6: Representação gráfica de matrizes de três dimensões.

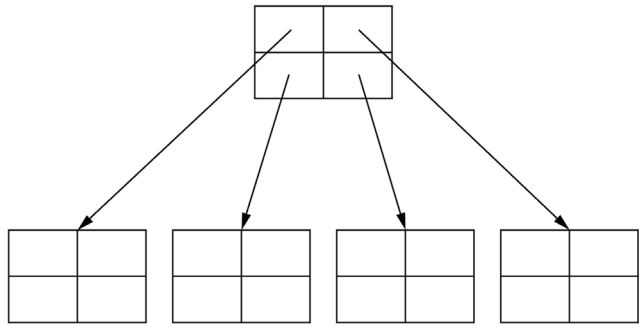


Figura 8.7: Representação gráfica de matrizes de quatro dimensões.

É importante mencionar que não é obrigatório o uso das analogias anteriores para representar matrizes com três ou mais dimensões. Muitas vezes, uma representação mais ligada ao problema é mais conveniente. Como exemplo, podemos citar um problema em que se tem um conjunto de objetos que se locomovem no espaço tridimensional ao longo do tempo. Todos eles começam em uma posição qualquer do espaço e, após um tempo, mudam de posição. O tempo de deslocamento desses objetos é desprezível. Desse modo, pode-se considerar que o tempo de viagem é zero e que eles mudam instantaneamente de posição. A posição dos objetos no espaço pode ser modelada como um *array* de três posições, que contém os valores em cada uma das coordenadas (x, y, z) do espaço. Logo, para um instante de tempo, existem vários *arrays* de três posições (um para cada objeto). Como os objetos mudam de posição após certo período, então precisamos ter múltiplos conjuntos de *arrays*. Assim, uma possível representação desse problema seria uma matriz de vetores que teria, em uma dimensão, os objetos e, na outra dimensão, os períodos. Cada célula dessa matriz teria um *array* de três posições indicando a localização cartesiana do objeto no espaço. Imagine que há uma linha para cada objeto e uma coluna para cada período de tempo. Essa matriz conteria na linha *i* e na coluna *j* as coordenadas *x*, *y*, *z* do objeto *i* no período *j*. A **Figura 8.8** mostra graficamente essa representação. Note que, ao contrário do convencional, essa matriz de três dimensões não é representada como um cubo, nem como um vetor de tabelas, mas como uma tabela de vetores.

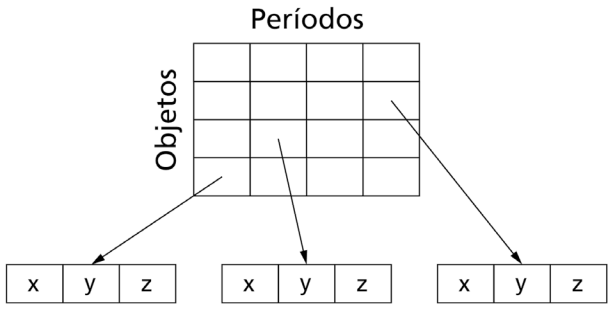


Figura 8.8: Representação gráfica de uma matriz de três dimensões, na qual cada elemento de uma tabela é um vetor.

Atividade 4

Atende aos Objetivos 1, 2 e 3

Aquela mesma empresa da Atividade 3 está revendo seus procedimentos internos: eles são quatro vendedores que vendem cinco produtos. Agora, no final do dia, cada vendedor entrega uma nota de cada tipo de produto diferente vendido. Cada nota contém: 1) número do vendedor; 2) número do produto; 3) dia do mês em que a venda ocorreu e 4) valor total vendido desse produto em reais nesse dia. O vendedor só é obrigado a preencher a nota caso tenha vendido o produto. No final do mês, o gerente da loja processa as notas de todos os funcionários para calcular a remuneração detalhada de cada um deles.

a) Represente graficamente uma matriz de três dimensões que contenha esses dados.

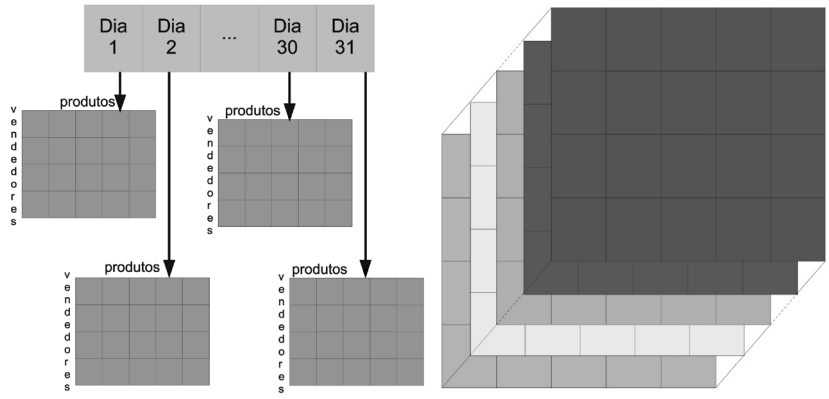
b) Faça um algoritmo que leia as informações de cada uma das notas, calcule e mostre a comissão diária de cada vendedor. Suponha que o número total de notas deve ser fornecido pelo gerente. Para a comissão

diária, devem ser considerados os valores de um mesmo dia. A comissão diária é calculada pela seguinte equação:

Comissão=10%*(totalProd1)+20%*(totalProd2)+10%*(totalProd3)+20%*(totalProd4)+10%*(totalProd5).

Resposta Comentada

A figura a seguir mostra duas possíveis representações para a matriz que organiza os dados. A primeira mostra um cubo e a segunda, uma lista de tabelas. Na primeira, cada cor representa um dia. Assim, deve existir uma tabela de quatro vendedores por cinco produtos nas duas representações. As linhas tracejadas e as reticências indicam a presença de outros itens não representados.



b) Esse problema requer o uso de uma matriz de três dimensões: uma para representar os vendedores, outra para representar os produtos e uma terceira para representar os dias. Uma das informações que são passadas pelo enunciado é a de que somente as vendas são reportadas; portanto, o vendedor não é obrigado a preencher notas de um produto quando ele não fez vendas daquele produto. Isso indica que pode não haver notas para todos os funcionários/dias/produtos, o que, por sua vez, indica que será necessário fazer uma inicialização dos valores da matriz utilizada. Outra informação dada é a de que o número de notas deve ser informado pelo gerente e essa informação facilitará o processo de leitura dos dados. Uma vez tendo os dados colocados na matriz de maneira correta, deve-se calcular para cada vendedor a comissão de cada dia. Para isso, é necessário fixar um vendedor e um dia, e percorrer todos os produtos vendidos naquele dia para calcular sua comissão. Ao término do cálculo, deve-se imprimir a comissão do dia e repetir essa tarefa para todos os dias e para todos os vendedores. A seguir está uma possível solução que utiliza esta estratégia. A primeira parte do algoritmo (linhas de 3 a 11) mostra a criação e a inicialização das variáveis. Note que a matriz *m* tem três dimensões. As linhas representam vendedores, as colunas, produtos, e as cotas representam os dias do mês. Como não serão lidas notas para todos os dias/produtos/vendedores, os valores da matriz devem ser inicializados com zero, para que não interfiram no cálculo da comissão. A segunda parte do algoritmo (linhas de 12 a 18) trata da leitura dos dados. Nela, é tratado o fato de que os dados do funcionário 1 estarão armazenados na linha zero e tratamento similar é feito para os produtos e dias. A terceira parte do algoritmo (linhas 19 a 31) trata do cálculo da comissão de cada dia para cada funcionário e da impressão de todos os valores calculados. Note que, para fazer o

cálculo, é fixado um vendedor (i) e um dia (d), e percorrem-se todos os produtos (j) para este vendedor/dia. Ao terminar de percorrer todos os produtos, o algoritmo exibe a comissão calculada.

```
Algoritmo comissao3D()
Início
   Inteiro i,j,d,notas
   Real [] [] [] m = Real [4][5][31]
   Para (i=0;i<4;i=i+1) Faça
      Para (j=0;j<5;j=j+1) Faça
         Para(d=0;d<31;d=d+1) Faça
            m[i][j][d]=0
         Fim Para
      Fim Para
   Fim Para
   Imprimir "Digite a quantidade de notas a serem processadas"
   Ler notas
   Para (k=0;k<notas;k=k+1) Faça
     Imprimir "Digite o número do vendedor, do produto, o dia e o valor vendido"
     Ler i,j,d
     Ler m[i-1][j-1][d-1]
   Fim Para
   Para (i=0;i<4;i=i+1) Faça
     Para (d=0;d<31;d=d+1) Faça
        Real com =0
        Para(j=0;j<5;j=j+1)Faça
           Se(j%2==0)Então
              com=com+soma*10.0/100
           Senão
              com=com+soma*20.0/100
           Fim Se
        Fim Para
        Imprimir "Comissão do vendedor "+(i+1)+" no dia "+d+"="+com
     Fim Para
   Fim Para
Fim
```

Como pôde ser observado, o uso de matrizes e vetores é fundamental para lidar com alguns problemas. Elas podem ser utilizadas tanto para guardar informações simples, como os dias em um calendário, como resultados de cálculos de modelos matemáticos sofisticados. Em outras disciplinas, como Álgebra Linear e Métodos Numéricos, você verá a aplicação de matrizes em soluções de problemas matemáticos e de engenharia, portanto o estudo dos algoritmos relacionados à criação e manipulação de matrizes lhe será muito útil ao longo do seu curso.

Atividade Final

Atende ao Objetivo 2

1. Lembrando que a soma de matrizes é feita pela soma dos termos correspondentes, faça um algoritmo que calcule a soma de duas matrizes de números inteiros de mesma dimensão.

2. Lembrando que a multiplicação de matrizes só é possível quando o número de colunas da primeira matriz é igual ao número de linhas da segunda matriz, e que a operação é feita multiplicando os termos da linha da primeira matriz pelos membros da coluna da segunda, faça um algoritmo que calcule a multiplicação de duas matrizes.

Resposta Comentada

1. Para fazer essa atividade, primeiro é preciso relembrar a operação da soma de duas matrizes. No Ensino Médio é mostrado que a soma de duas matrizes é uma terceira matriz e que a equação que descreve a matriz resultante é a seguinte: $c_{ij} = a_{ij} + b_{ij} \forall i \in \{1,\ldots,m\}, \forall j \in \{1,\ldots,n\}$, em que a é a primeira matriz, b a segunda, c a matriz resultante, m o número de linhas e n o número de colunas. Ou seja, para calcular um elemento da matriz resultante, basta somar os elementos de mesma posição nas

matrizes *a* e *b*. Uma possível solução é apresentada a seguir. Note que ela é mais extensa do que complicada. Em geral, algoritmos que envolvem múltiplas matrizes são extensos porque os programadores costumam tratar cada operação de maneira isolada. É possível, por exemplo, fazer as três operações (leitura do elemento de *a*, leitura do elemento de *b* e cálculo do elemento de *c*) simultaneamente. Porém, em vários casos, misturar tarefas torna o problema mais complicado e o código mais confuso de ser entendido.

```
Algoritmo somaMatrizes()
Início
   Inteiro i,j,l,c
   Imprimir "Digite o número de linhas e colunas das matrizes"
   Ler l,c
   Inteiro [] [] m1 = Inteiro [l][c], m2=Inteiro[l][c], m3 = Inteiro[l][c]
   Imprimir "Digite os elementos das matrizes"
   Para (i=0;i<l;i=i+1) Faça
     Para(j=0;j<c;j=j+1)Faça
       Ler m1[i][j]
     Fim Para
   Fim Para
   Para (i=0;i<l;i=i+1) Faça
     Para(j=0;j<c;j=j+1)Faça
       Ler m2[i][j]
     Fim Para
   Fim Para
   Para (i=0;i<l;i=i+1) Faça
     Para(j=0;j<c;j=j+1)Faça
       m3[i][j] = m1[i][j] +m2[i][j]
     Fim Para
   Fim Para
   Imprimir "A matriz resultante é:"
   Para (i=0;i<l;i=i+1) Faça
     Texto t= ""
     Para(j=0;j<c;j=j+1)Faça
       t = t + m3[i][j] + " "
     Fim Para
     Imprimir t
   Fim Para
Fim
```

2. Este também é um algoritmo bem extenso, caso todas as tarefas sejam feitas separadamente. A seguir, está uma possível solução do problema. A primeira parte do algoritmo (linhas 1 a 17) trata da inicialização e da leitura dos dados. A segunda trata da multiplicação das matrizes (linhas 18 a 25) e a terceira (linhas 26 a 34), da impressão da matriz resultante. Para relembrar, a equação para calcular a multiplicação de matrizes é $c_{ij} = \sum_{k=1}^{m} a_{ik} \times b_{kj} \forall i \in \{1,\ldots,l\}, \forall j \in \{1,\ldots,n\}$, sendo *c* a matriz resultante, *a*

e *b* as duas matrizes a serem multiplicadas, *l* o número de linhas da matriz *a*, *m* o número de colunas da matriz *a* (e consequentemente o número de linhas de *b*) e *n* o número de colunas da matriz *b*. Note que, para encontrar um elemento da matriz *c*, é preciso fazer uma série de multiplicações dos elementos da matriz *a* por elementos da matriz *b* e, depois, somar os resultados dessas multiplicações. Desse modo, antes de começar a somar, é preciso zerar a variável em que a soma será armazenada (linha 20). Em seguida, ao valor da variável, é somado o resultado da multiplicação dos respectivos termos de *a* e *b* (linha 22). As repetições que cuidam das multiplicações são controladas de acordo com as definições do problema. As linhas da matriz resultante são tratadas em função das linhas de *a*, as colunas da matriz resultante são tratadas em função das colunas de *b* e a repetição mais interna é controlada de acordo com o número de colunas de *a* (que é igual ao número de linhas de *b*).

```
Algoritmo multiplicaMatrizes()
Início
   Inteiro i,j,k,l,c,c2
   Imprimir "Digite o número de linhas  m1 e de linhas e colunas de m2"
   Ler l,c,c2
   Real [] [] m1 = Real [l][c], m2=Real[c][c2], m3 = Real[l][c2]
   Imprimir "Digite os elementos das matrizes"
   Para (i=0;i<l;i=i+1) Faça
     Para(j=0;j<c;j=j+1)Faça
       Ler m1[i][j]
     Fim Para
   Fim Para
   Para (i=0;i<c;i=i+1) Faça
     Para(j=0;j<c2;j=j+1)Faça
       Ler m2[i][j]
     Fim Para
   Fim Para
   Para (i=0;i<l;i=i+1) Faça
     Para(j=0;j<c2;j=j+1)Faça
       m3[i][j] = 0
       Para(k=0;k<c;k=k+1)
         m3[i][j] = m3[i][j] + m1[i][k] *m2[k][j]
       Fim Para
     Fim Para
   Fim Para
   Imprimir "A matriz resultante é:"
   Para (i=0;i<l;i=i+1) Faça
     Texto t= ""
     Para(j=0;j<c2;j=j+1)Faça
       t = t + m3[i][j] + " "
     Fim Para
     Imprimir t
   Fim Para
Fim
```

Resumo

Nesta aula você aprendeu a declarar e acessar matrizes. Viu que elas são conjuntos de variáveis de um mesmo tipo e que são acessadas através do nome e de múltiplos índices, de acordo com o número de dimensões que possuem. Também foi mostrado o modo de compor algoritmos combinando matrizes, estruturas de repetição e de desvio, a fim de resolver problemas.

Informações sobre a próxima aula

Na próxima aula, você verá como lidar com funções e procedimentos.

Aula 9

Funções e procedimentos: parte I

Tiago Araújo Neves

Meta

Expor os conceitos básicos de funções e procedimentos, mostrando sua sintaxe e funcionamento.

Objetivos

Esperamos que, ao final desta aula, você seja capaz de:

1. definir e aplicar o conceito de funções e procedimentos;
2. fazer algoritmos que utilizem funções para resolver problemas.

Introdução

Em muitos algoritmos feitos até o momento, mesmo fazendo uso de estrutura de repetição, é necessário fazer replicação do código. Um exemplo claro são os algoritmos que envolvem matrizes. Quando é preciso fazer a leitura de dados de uma matriz de duas dimensões, fazem-se duas estruturas de repetição, uma para as linhas e outra para as colunas. Dentro dessas duas estruturas, fica a instrução de leitura da matriz nas posições controladas pelas repetições. Para ler uma segunda matriz, deve-se criar um segundo conjunto com duas estruturas de repetição. Para ler uma terceira matriz, um terceiro conjunto, e assim sucessivamente. Esse padrão de repetições sucessivas de conjuntos muito semelhantes de estruturas acaba por aumentar em demasia o tamanho dos algoritmos. Além disso, fazer manutenção (corrigir erros) em algoritmos que seguem esse modelo é uma tarefa árdua, uma vez que, se um dos conjuntos de repetição deve ser alterado, provavelmente muitos outros também devem ser.

Para lidar com esse tipo de problema (repetições de comportamento para entidades diferentes) dentro da computação, são utilizadas funções e **procedimentos**. Tanto as funções quanto os procedimentos podem ser definidos como trechos de códigos parametrizados, invocáveis a partir de um nome, que executam uma ou mais tarefas computacionais. A ideia é similar às funções da matemática. Tome como exemplo a **função** $f(x)=2x+1$. Ela pode ser aplicada a qualquer valor de x pertencente a $\mathbb{R}$, e o comportamento será sempre o mesmo: o valor de x será dobrado e depois acrescido de uma unidade. Desse modo, o que as funções fazem é estabelecer comportamentos que serão executados a um conjunto de dados de entrada. Note que, para diferentes valores de x, há diferentes resultados. Por exemplo: para $x=-2$, tem-se $f(x)=-3$; para $x=2$, tem-se $f(x)=5$. Observe que, dependendo do valor do parâmetro de entrada, a função apresenta um resultado diferente, mas o comportamento é sempre o mesmo. O parâmetro de entrada é dobrado e acrescido de uma unidade.

Procedimento

Subprograma sem retorno obrigatório. Também chamado de subrotina.

Função

Subprograma com retorno obrigatório. Os resultados de funções podem ser usados para compor expressões de maneira direta.

Nesta aula, você verá que o uso de funções e procedimentos pode simplificar muito os algoritmos, bem como facilitar sua manutenção. Além disso, o uso de funções e procedimentos promove, de maneira geral, a redução do tamanho dos algoritmos, o que facilita a construção e o entendimento de algoritmos para problemas complexos.

Funções e procedimentos: declaração e uso

A declaração de subprogramas segue a sintaxe da **Figura 9.1**: em (a), a sintaxe de declaração de funções e em (b), a sintaxe de declaração de procedimentos. Ambas têm a mesma finalidade: repetir um padrão de comportamento sobre o conjunto de parâmetros dado. Note que as duas estruturas possuem um "nome", que nada mais é do que o nome da função/procedimento. Os nomes dados a funções/procedimentos podem ser quaisquer identificadores válidos; contudo, normalmente se utilizam identificadores que expressam o que a função/procedimento irá fazer. A matemática tem por hábito nomear as funções com letras, por exemplo, função *f*, função *g* etc. Embora isso seja possível, também em algoritmos está prática não é bem vista por cientistas e programadores. Um bom nome para uma função/procedimento deve dar uma ideia clara do que a função/procedimento faz, sem que seja necessário olhar toda a estrutura da função/procedimento. Suponha que existam duas funções, uma chamada *a1* e outra chamada *raizQuadrada*. Note que só pelo nome é possível deduzir o que a função raizQuadrada irá fazer, o que já não ocorre com a função *a1*. Neste caso, para saber o que a função *a1* faz, é necessário olhar para toda a sua estrutura.

(a)

```
tipo nome(<parâmetros>)
Início
   /*instruções*/
Fim
```

(b)

```
Procedimento nome(<parâmetros>)
Início
   /*instruções*/
Fim
```

Figura 9.1: Sintaxe de declaração de funções (a) e procedimentos (b).

Toda função tem um tipo, que é o tipo de dado que será a resposta da função. Exemplo, se uma função é do tipo inteiro, a resposta que ela retornará será um número inteiro; se ela for do tipo texto, ela retornará uma palavra, etc. Procedimentos, também chamados de subrotinas ou funções sem tipo, não são obrigados a retornar nada. Os procedimentos costumam ser as ferramentas mais adequadas nos casos em que o que se deseja fazer é simplesmente executar as instruções, sem necessariamente calcular ou construir uma resposta. Como exemplo desses casos, podemos citar a impressão de dados sem formato específico.

Entre o início e o fim de um subprograma, estão as instruções que compõem o corpo da função/procedimento.

Funções e procedimentos podem ser divididos em duas partes:

1. assinatura (que contém a classificação do subprograma, nome e lista de parâmetros);
2. corpo (instruções que se deseja executar sobre os parâmetros).

Quaisquer instruções podem ser utilizadas: criação de variáveis e vetores, desvios condicionais, estruturas de repetição etc. A única exceção é a declaração de funções/procedimentos, ou seja, dentro de uma função, podem existir quaisquer instruções de código, exceto a definição de outras funções/procedimentos. Para se criar mais de uma função/procedimento, deve-se fazê-las separadamente. Assim, para fazer 3 funções, por exemplo, faz-se a primeira do início ao fim. Depois, constrói-se a segunda do início ao fim e, por último, a terceira.

Os parâmetros são passados com tipo e nome separados por vírgulas. Um possível exemplo de lista de parâmetros é o seguinte: *Inteiro idade, Real salário, Texto nome*. Em alguns casos, podem existir parâmetros do mesmo tipo: *Inteiro Idade, Inteiro numFilhos, Inteiro numFilhas*. Nestes casos, deve-se especificar o tipo de cada parâmetro, mesmo que eles tenham o mesmo tipo. É importante ressaltar que o tipo dos parâmetros nada tem a ver com o tipo de resposta de uma função. Por exemplo, pode haver funções que recebem números reais como parâmetros e retornam números reais. Entretanto, também pode haver funções que recebem textos como parâmetros e retornam números como resposta. Um exemplo seria uma função para calcular quantas letras uma palavra tem. Esta função receberia uma variável do tipo texto e retornaria um número inteiro.

O uso de funções em algoritmos

Para ilustrar o uso de funções, vejamos a solução de um problema muito comum no dia a dia de um engenheiro: o problema de conversão de unidades. Sabe-se que um centímetro corresponde a 0.3937 polegadas. Devem-se imprimir as polegadas que correspondem a 0, 0.5, 1.0,..., 10 centímetros. A **Figura 9.2** mostra a função utilizada para esse problema. Ela converte um valor real passado como parâmetro (cm) para seu correspondente em polegadas através de uma regra de três simples. Note que, dentro da função, existe a instrução *Retorne*. Essa instrução indica que o que está escrito à frente dela deve ser retornado como resultado da função. Assim, primeiramente será feita a multiplicação da variável *cm* com a constante 0.3937. Em seguida, o resultado da multiplicação é retornado como resultado da função.

```
Real converterCmParaPol(Real cm)
Início
   Retorne cm*0.3937
Fim
```

Figura 9.2: Função para converter centímetros em polegadas.

A **Figura 9.3** mostra um algoritmo que faz uso da função *converterCmParaPol* descrita na **Figura 9.2** para resolver o problema citado. Note que a função é chamada diversas vezes dentro da estrutura de repetição e que seu resultado é impresso diretamente.

```
Algoritmo testeFuncao()
Início
   Real i
   Para(i=0;i<=10;i=i+0.5)Faça
      Imprimir converterCmParaPol(i)
   Fim Para
Fim
```

Figura 9.3: Algoritmo que usa a função definida na Figura 9.2.

O uso de procedimentos em algoritmos

A **Figura 9.4** mostra um exemplo de procedimento cujo objetivo é imprimir os dados de um vetor. Note que os parâmetros são o vetor e o número de posições. Como o objetivo é somente imprimir os dados, não há necessidade de respostas e, portanto, um procedimento é mais adequado.

```
Procedimento imprimirVetorFormatoLinha(Real[] vetor, Inteiro tam )
Início
   Inteiro j
   Texto t= “”
   Para(j=0;j<tam;j=j+1)Faça
      t=t+vetor[j]+“ ”
   Fim Para
   Imprimir t
Fim
```

Figura 9.4: Um exemplo de procedimento para imprimir um vetor.

Para utilizar este procedimento, deve-se criar um algoritmo para fazer chamadas ao procedimento. A **Figura 9.5** mostra um exemplo de uso. O algoritmo declara dois vetores, preenche-os com valores digitados pelo usuário e depois os imprime.

```
Algoritmo testeProcedimeto()
Início
   Inteiro t1,t2
   Imprimir "Dite o tamanho de dois vetores"
   Ler t1,t2
   Real[] v1=Real[t1],v2=Real[t2]
   Imprimir "Digite os valores dos vetores"
   Inteiro i
   Para(i=0;i<tam;i=i+1)Faça
      Ler v1[i]
      Ler v2[i]
   Fim Para
   Imprimir "os vetores Digitados foram"
   imprimirVetorFormatoLinha(v1,t1)
   imprimirVetorFormatoLinha(v2,t2)
Fim
```

Figura 9.5: Algoritmo que faz uso do procedimento definido na Figura 9.4.

Cuidados necessários na elaboração de subprogramas

Neste ponto, é preciso levantar algumas questões. A primeira é: onde posicionar o procedimento? Antes ou depois do algoritmo? Esta é uma questão que varia muito de acordo com cada linguagem de programação. Por questões didáticas, usaremos a mesma regra utilizada para as variáveis, ou seja, funções só podem ser utilizadas depois de definidas, e o mesmo vale para os procedimentos. Logo, para resolver a questão no nosso exemplo, primeiro define-se o procedimento (**Figura 9.4**) para depois utilizá-lo (**Figura 9.5**).

Mesma lógica, diferentes formas

Em C/C++ é usual colocar as funções antes do programa que faz as chamadas (embora existam mecanismos para burlar isso). Em Fortran, as funções/subrotinas normalmente são colocadas depois do programa que faz a chamada. Em Java, não existe restrição. As funções podem ser colocadas antes ou depois do programa.

Uma segunda questão é: como funciona a chamada de uma função ou de um procedimento? O fluxo do programa segue normalmente até a chamada da função/procedimento. Neste momento, ocorre um desvio de fluxo para dentro da função/procedimento, fazendo com que as instruções do corpo da função/procedimento sejam executadas. Quando o fluxo chega ao final do corpo da função/procedimento (Fim), ele é novamente desviado para o algoritmo que fez a chamada. Ao retornar para o algoritmo, o fluxo continua do ponto após a chamada. É importante mencionar que as variáveis declaradas dentro de funções/procedimentos são criadas no momento da chamada e destruídas quando o fluxo de execução chega ao fim do corpo da função/procedimento. A **Figura 9.6** mostra a trajetória do fluxo graficamente. Ele inicia a execução do início do algoritmo e vai até a chamada do procedimento. Neste ponto, é desviado para o procedimento e executa as instruções contidas nele. Ao terminar a execução do procedimento, o fluxo é novamente desviado para o algoritmo, retornando uma instrução após a chamada. Note que neste caso o ponto de retorno é uma segunda chamada de procedimento. O fluxo é desviado uma segunda vez para o procedimento, que tem suas instruções executadas pela segunda vez. Ao término da segunda execução, o fluxo retorna ao algoritmo uma instrução após a segunda chamada. Como o ponto de retorno é justamente o fim do algoritmo, isso encerra sua execução, porém, se houvesse instruções após a segunda chamada, elas seriam executadas normalmente até o fim do algoritmo.

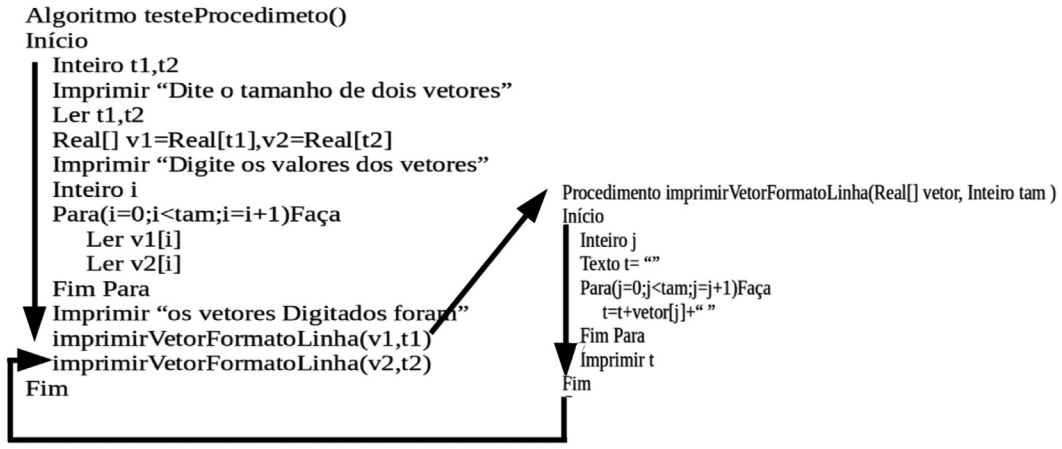


Figura 9.6: Trajetória do fluxo de programa com chamada de procedimento.

A terceira questão a ser levantada é como um mesmo subprograma pode ser usado para um vetor ou variáveis que tem nome diferente do que foi definido? Este é exatamente o objetivo de se utilizarem funções e procedimentos: repetir padrões de comportamento para dados diferentes. Se só fosse possível utilizar um procedimento para variáveis com mesmo nome que os definidos por ele, haveria necessidade de criar vários procedimentos idênticos, e o reuso seria muito baixo. O uso de nomes diferentes é possível porque, no momento da chamada da função/procedimento, é criada uma cópia dos dados, e as cópias são passadas para o procedimento. No nosso exemplo, na primeira chamada do procedimento, é criada uma cópia de *v1* e uma cópia de *t1*. Elas são denominadas *vetor* e *tam*, respectivamente. Logo, o procedimento trabalha com a cópia dos dados. Ao terminar a execução, essas cópias, bem como todas as variáveis criadas pelo procedimento, são descartadas. Devido a isso, são chamadas variáveis automáticas. Elas são criadas e destruídas automaticamente.

A função só pode ser chamada com exatamente a quantidade e o tipo de parâmetros estipulados em sua definição. O mesmo vale para os procedimentos.

É importante mencionar que, no momento de definir uma função/procedimento, o número de parâmetros, o tipo de cada um deles e a ordem em que são dispostos são de extrema importância. Ao definir uma função com três parâmetros, ela não pode ser chamada com apenas dois ou quatro parâmetros. No nosso exemplo, o procedimento *imprimirVetorFormatoLinha* só pode ser chamado com dois parâmetros. O tipo de cada um deles também é muito importante. Se na definição de uma função/procedimento for passado como parâmetro um vetor de números reais, não adianta passar um vetor de números inteiros no momento da chamada. No nosso exemplo, o procedimento criado só serve para vetores de números reais. Para vetores de números inteiros, é necessária a criação de um segundo procedimento. A ordem dos parâmetros estabelecida no momento da definição da função/procedimento também deve ser respeitada. Logo, não é possível chamar o procedimento do exemplo passando primeiro o tamanho e depois o vetor. Isso ocorre porque, no momento da definição, estabeleceu-se que o primeiro parâmetro é sempre o vetor, e o segundo é sempre o tamanho.

Uma quarta questão que surge é: se foi feito um procedimento para imprimir os vetores, é possível criar um procedimento para a leitura dos vetores? A resposta é sim. Porém o momento não é oportuno para mostrar esse tipo de procedimento. Com as ferramentas mostradas até aqui, ao chamar um procedimento para leitura de um vetor, será criada uma cópia do vetor, e os dados da cópia serão lidos. Como a cópia é destruída automaticamente, os dados lidos serão perdidos. Para criar uma função/procedimento que altere os dados originais de entrada, é preciso utilizar um mecanismo chamado passagem de parâmetro por referência. Ele será detalhado na Aula 10. Por enquanto, vamos apenas trabalhar com passagem de parâmetros por valor, ou seja, por meio de cópias.

Atividade 1

Atende ao Objetivo 1

Mostre o que será impresso ao executar o seguinte algoritmo.

```
Procedimento procedimentoAuxiliar( Inteiro t )
Início
   Imprimir (t+1)
Fim

Algoritmo questao1()
Início
   Inteiro t1,t2
   t1=3
   t2=5
   procedimentoAuxiliar(t1)
   Imprimir t1
   procedimentoAuxiliar(t2)
   Imprimir t2
Fim
```

Resposta Comentada

```
4
3
6
5
```

O algoritmo começa criando duas variáveis, *t1* e *t2*, e atribuindo a *t1* o valor 3 e a *t2* o valor 5. Em seguida, é feita uma chamada ao procedimento, passando *t1* como parâmetro. O fluxo de execução é transferido para o procedimento que executa a instrução de imprimir o parâmetro adicionado a 1; como *t1* tem valor 3, então será impresso 4. O fluxo retorna ao algoritmo, e a próxima instrução a ser executada imprime o valor de *t1*, logo, a segunda linha a ser impressa conterá o valor 3. Na sequência, o procedimento é chamado, mas desta vez com a variável *t2* como parâmetro. O procedimento então se encarrega de

imprimir a terceira linha que será *t2*+1, ou seja, 6. O fluxo retorna ao programa a instrução para imprimir *t2*. Ela é executada, imprimindo o valor 5. Como não há mais instruções após esta última impressão, o algoritmo é encerrado.

Atividade 2

Atende aos Objetivos 1 e 2

Faça um procedimento que receba como parâmetro um vetor (matriz unidimensional) de números inteiros e também o tamanho deste vetor. Esse procedimento deve imprimir o vetor na ordem normal e na ordem inversa.

Resposta Comentada

O enunciado da questão diz que tanto o vetor quanto o tamanho do vetor devem ser passados como parâmetros, portanto, devem estar na assinatura do procedimento. Ao analisar a situação, você perceberá que nenhum outro parâmetro é preciso. Para imprimir um vetor, é preciso saber qual vetor será impresso, seu tamanho, uma variável de contagem e uma estrutura de repetição. Tendo o vetor e o tamanho como parâmetros, a variável de contagem e a estrutura de repetição podem fazer parte do corpo do procedimento. Uma possível solução é mostrada a seguir. Nela, foi usada uma variável *i* como contador e uma estrutura de repetição *Para*, com o objetivo de imprimir o vetor na ordem normal. A mesma variável de contagem foi utilizada em uma segunda estrutura de repetição para imprimir o vetor na ordem inversa.

```
Procedimento imprimirNormalInversa(Inteiro[] vet, Inteiro tam)
Início
   Inteiro i
   Imprimir "vetor na ordem normal"
   Para(i=0;i<tam;i=i+1)Faça
      Imprimir vet[i]
   Fim Para
   Imprimir "vetor na ordem inversa"
   Para(i=tam-1;i>=0;i=i-1)Faça
      Imprimir vet[i]
   Fim Para
Fim
```

Particularidades de funções

O fluxo do algoritmo que utiliza funções segue os mesmos princípios daquele que usa procedimentos. O fluxo vai até a chamada de função, executa o corpo da função e retorna ao algoritmo logo depois da chamada. Contudo, o fluxo dentro de funções não tem o mesmo comportamento que o fluxo dentro de procedimentos. Nestes, o fluxo vai do início ao fim. Em funções, ele vai do início até encontrar a primeira instrução *Retorne*. A **Figura 9.7** mostra um exemplo de função com duas instruções *Retorne*. Essa função calcula o valor absoluto de um número dado pela equação a seguir:

$$|x| = \begin{cases} x, se\ \ x \geq 0 \\ -x,\ c.c. \end{cases}$$

A equação diz que o valor absoluto de um número x, também chamado módulo de x, é o próprio x se x for maior ou igual a zero ou x multiplicado por −1.

```
Real abs(Real x)
Início
   Se (x>=0)Então
      Retorne x
   Senão
      Retorne -x
   Fim Se
Fim
```

Figura 9.7: Função para cálculo do valor absoluto de um número.

Note que não é possível saber, *a priori*, até onde o fluxo irá na função *abs*. Caso *x* seja positivo, o fluxo encontrará o primeiro *Retorne,* e o restante da função será ignorado. Caso *x* seja negativo, o comando *Se* desvia o fluxo para o segundo *Retorne*. Como uma função termina quando encontra uma instrução *Retorne*, quando essa instrução aparece múltiplas vezes, fica difícil – ou até impossível – determinar até onde o fluxo de uma função irá passar.

Utilizando funções para compor novas funções

O resultado de funções também pode ser armazenado em variáveis ou utilizado para compor expressões mais complexas. Também é possível fazer chamadas de funções/procedimentos dentro de outras funções/procedimentos. As **Figuras 9.8** e **9.9** mostram um exemplo da chamada de uma função dentro de outra função. O problema tratado pelas funções é encontrar o número de combinações possíveis, compostas de *p* elementos escolhidos dentro de um conjunto de *n* elementos. Por exemplo: dado um conjunto *A*, de dez elementos, quantos subconjuntos de três elementos é possível construir utilizando os elementos de *A*? A solução deste problema é dada pela equação da combinação de *n*(10) elementos tomados *p*(3) a *p*(3). A equação do número de possíveis combinações é a seguinte:

$$c(n,p)=\frac{n!}{p!(n-p)!}$$

Note que, para calcular o número de possíveis combinaçõe, é preciso calcular vários fatoriais. Este é um indicativo de que, se uma função para calcular o fatorial de um número for definida, a solução do problema de encontrar o número de combinações ficará mais fácil. A definição matemática do fatorial de um número é mostrada a seguir:

$$n!=\begin{cases}1, se\ n\in\{0,1\}\\ n(n-1)!, c.c.\end{cases}$$

A **Figura 9.8** mostra uma função para calcular o fatorial de um número qualquer *x*. Note que, se *x* for igual a zero ou a um, a estrutura de repetição não altera o valor de *res*. Caso *x* seja maior que um, *res* é alterada sucessivamente até conter o resultado desejado.

```
Inteiro fat(Inteiro x)
Início
   Inteiro i,res
   res=1
   Para(i=2;i<=x;i=i+1)Faça
      res=res*i
   Fim Para
   Retorne res
Fim
```

Figura 9.8: Função para o cálculo do fatorial.

Uma vez que a função para o cálculo do fatorial já está definida, ela pode ser usada para compor a função do cálculo do número de combinações. A **Figura 9.9** mostra o cálculo do número de combinações de *n* elementos tomados *p* a *p* utilizando a função definida na **Figura 9.8**.

```
Inteiro combinacoes(Inteiro n, Inteiro p)
Início
   Retorne fat(n)/(fat(p)*fat(n - p))
Fim
```

Figura 9.9: Função para o cálculo do número de combinações utilizando a função definida na Figura 9.8.

Observe que, uma vez definida a função *fat* para o cálculo do fatorial, basta utilizá-la para compor a função para calcular o número de combinações. Também é possível fazer a função do número de combinações sem o uso de uma função auxiliar, contudo, esta função envolveria três estruturas de repetição: 1) para calcular o fatorial de *n*; 2) para calcular o fatorial de *p* e 3) para calcular o fatorial de *n−p.* Note que isso aumentaria consideravelmente o tamanho (em linhas) da função para o cálculo de combinações. Também é importante mencionar que expressões aritméticas/lógicas podem ser usadas como parâmetros de função durante sua chamada. Note que a expressão da função definida na **Figura 9.9** apresenta um *fat(n−p)*. Isso é possível devido ao fato de que as expressões aritméticas/lógicas são resolvidas antes das chamadas das funções. Desse modo, primeiro será computado o resultado da expressão *n−p* e depois serão feitas as chamadas da função *fat*. Vamos acompanhar passo a passo a execução desta função para *n=5* e *p=3.*

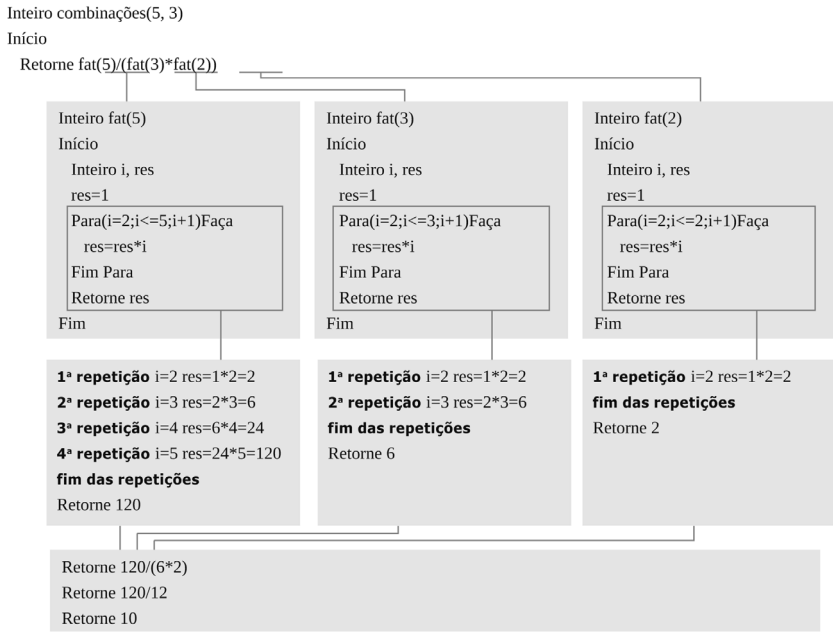


Figura 9.10: Passo a passo da execução desta função para descobrir quantos subconjuntos de três elementos (p=3) existem dentro de um conjunto de 5 elementos (n=5).

1) A primeira coisa a se fazer é substituir os valores na expressão: *retorne fat(n)/(fat(p)*fat(n−p))* é convertido para *retorne fat(5)/(fat(3)*fat(5−3))*.

2) São executadas as expressões aritméticas e lógicas, logo, a expressão 5−3= 2 é resolvida e se torna *retorne fat(5)/(fat(3)*fat(2))*.

3) Para terminar a solução, é necessário executar as chamadas de função, que são avaliadas da esquerda para a direita, logo, a expressão é convertida para *retorne 120/(fat(3)*fat(2)),* depois para *retorne 120/(6*fat(2))* e, por último, *Retorne 120/(6*2)*.

4) Neste ponto, a expressão aritmética é resolvida começando pelos parênteses, sendo convertida para *retorne 120/12* que, por sua vez, é convertida para *retorne 10*.

Atividade 3

Atende aos Objetivos 1 e 2

Escreva um algoritmo que utilize uma função com a seguinte assinatura: Lógico e Triângulo (Real a, Real b, Real c). Essa função é capaz de dizer se três números representam os lados de um triângulo, e retorna *verdadeiro* quando os três números correspondem aos lados de um triângulo, e *falso*, em caso contrário. Suponha também que a função já é fornecida, ou seja, não é necessário defini-la. O algoritmo deve ler diversos conjuntos de números e só parar quando o usuário digitar um número que seja menor ou igual a zero.

Resposta Comentada

O enunciado informa que uma função já está definida e que ela recebe três números reais, e indica se esses três números correspondem ou não aos lados de um triângulo. Como esta função já está definida, o que se deve fazer então é um algoritmo que a utilize. Em uma possível solução, três variáveis reais são utilizadas para armazenar os números a serem checados. Como vários conjuntos numéricos devem ser lidos, foi utilizada uma estrutura de repetição para fazer essas múltiplas leituras, tomando cuidado para que a estrutura seja interrompida quando um número menor ou igual a zero for digitado. Uma vez lidos os números, eles são submetidos à função dada pelo enunciado, e o resultado é utilizado em uma estrutura de desvio condicional. Caso a resposta da função seja *verdadeiro*, a mensagem "os números correspondem a um triângulo" será impressa. Caso contrário, uma mensagem informando que os números não correspondem a um triângulo será exibida. O algoritmo prossegue com esse comportamento até que um número menor ou igual a zero

seja digitado. Neste ponto, o algoritmo imprimirá a mensagem de que os números não correspondem a um triângulo (não pode haver triângulos onde um dos lados tenha comprimento negativo ou igual a zero) e terminará sua execução.

```
Algoritmo utilizaETriangulo()
Início
   Real l1=1,l2=1,l3=1
   Enquanto(l1>0 & l2 >0 & l3 >0) Faça
      Imprimir "Digite os três números"
      Ler l1, l2, l3
      Se (eTriangulo(l1,l2,l3)) Então
         Imprimir "Os números correspondem a um Triângulo"
      Senão
         Imprimir "Os números Não correspondem a um Triângulo"
      Fim Se
   Fim Enquanto
Fim
```

Atividade Final

Atende aos Objetivos 1 e 2

1. Faça uma função para resolver a série: $sen(x)=x-x^3/3!+x^5/5!-x^7/7!\ldots$, em que x é o valor de um ângulo em radianos.

2. Faça uma função que receba um conjunto de números e o tamanho dele, e calcule a variância desse conjunto que é dada por: $\frac{1}{n}\sum_{i=1}^{n}(x_i-\mu)^2$, em que μ é a média do conjunto. Teste a sua função para a sequência {1,2,3,4,5,6}. O resultado deve se aproximar de 2,91.

Resposta Comentada

1. Apesar de esta questão já ter sido feita na Aula 5, é bom revisitá-la para que você possa ver a utilidade de funções nesse tipo de situação. A série do seno é composta por uma soma de vários termos e para cada termo deve-se calcular um fatorial e uma potenciação. Como uma função para o cálculo do fatorial já foi definida nesta aula, pode-se utilizá-la para criar uma função para o cálculo do seno. Para se chegar a uma solução, é preciso determinar primeiro como será a assinatura da função. O seno de um ângulo é um valor real entre zero e um, logo, é natural que o tipo de retorno da função seja real. Como o objetivo é calcular o seno de um ângulo, seno é um nome bastante apropriado para a função. Como parâmetros, é necessário saber o ângulo e o número de termos da série que será utilizado. Por razões óbvias, o ângulo deve ser um número real e a quantidade de termos um número inteiro. Também são necessárias algumas variáveis auxiliares: *soma*, para armazenar a soma dos termos; *sinal*, para representar a mudança de sinal em cada termo; *i*, para contar a quantidade de termos e *exp*, para controlar expoente em cada termo. Note que a expressão na linha 6 da solução apresentada usa a função *fat* definida anteriormente para compor o termo atual. Como a repetição controla o número de termos, o expoente e o sinal são controlados separadamente. Ao final, o valor de soma é retornado, pois ele contém a soma de todos os termos da série que foram calculados.

```
Real seno(Real ang, Inteiro termos)
Início
   Real soma=0,sinal=1
   Inteiro i, exp=1
   Para(i=1;i<=termos;i=i+1) Faça
      soma=soma+ sinal*(ang^exp)/fat(exp)
      exp = exp +2
      sinal = -sinal
   Fim Para
   Retorne soma
Fim
```

2. Este problema também já foi resolvido na Aula 7, porém, o uso de funções pode dar uma nova perspectiva sobre sua solução. Note que, para calcular a variância de um conjunto, primeiro devemos calcular a média desse conjunto. Portanto, para facilitar a solução do problema, uma função que calcula a média de um conjunto será definida.

```
Real mediaConjunto(Real [] vet, Inteiro tam)
Início
   Real soma=0
   Inteiro i
   Para(i=0;i<tam;i=i+1) Faça
      soma=soma+ vet[i]
   Fim Para
   Retorne soma/tam
Fim
```

Essa função calcula a soma dos elementos do conjunto utilizando uma estrutura de repetição e retorna a soma dividida pelo número de elementos. Uma vez definida esta função da média dos elementos de um conjunto, deve-se analisar a equação da variância, que pode ser reescrita da seguinte maneira: $\frac{1}{n}\sum_{i=1}^{n} y_i$, em que $y_i=(x_i-\mu)^2$ e μ a média dos elementos. Note que a variância também é a média dos elementos de um segundo conjunto, em que cada elemento é dado como o quadrado de um elemento do primeiro subtraído da média. Com isso, uma possível estratégia é criar um segundo conjunto, no qual cada termo seja definido com a equação de y_1 e utilizar a função que calcula a média de um conjunto para calcular a variância. Uma solução que utiliza essa estratégia é mostrada a seguir. Nela, a primeira tarefa é calcular a média do conjunto *vet*. Em seguida, é criado um conjunto *aux* que tem cada elemento definido de acordo com a equação de y_1. Por último, a variância de *vet* é dada pela média dos elementos de *aux*.

```
Real varianciaConjunto(Real [] vet, Inteiro tam)
Início
   Real mi=mediaConjunto(vet,tam)
   Real[] aux = Real[tam]
   Para(i=0;i<tam;i=i+1) Faça
      aux[i]=(vet[i]-mi)^2
   Fim Para
   Retorne mediaConjunto(aux,tam)
Fim
```

Resumo

Nesta aula, você aprendeu a sintaxe de declaração de funções e procedimentos, bem como os mecanismos para utilizá-los. Viu que funções retornam resultados que podem ser usados para composição de expressões. Também aprendeu que funções e procedimentos podem ser utilizados dentro de outras funções e procedimentos, buscando criar soluções para problemas mais complexos.

Definir boas funções e utilizar funções definidas para compor funções mais complexas são práticas comuns entre programadores e são também consideradas boas práticas de programação, uma vez que o uso de funções e procedimentos promovem a simplificação e a reutilização de algoritmos, além de facilitar a manutenção nos mesmos. Desse modo, você já está apto a entender, e até mesmo criticar, a organização algorítmica de soluções computacionais para problemas complexos, como os que são apresentados em artigos científicos e monografias de final de curso. Muitas soluções dessa natureza são construídas como um único algoritmo, o que sem dúvida dificulta a legibilidade e a manutenção desses códigos. Além disso, vimos que o uso de funções ou procedimentos simplifica o raciocínio, o que pode facilitar as soluções e economizar tempo em sua elaboração.

Informações sobre a próxima aula

Na próxima aula, você verá como lidar com recursão e com passagem de parâmetros por referência.

Aula 10

Funções e procedimentos: parte II

Tiago Araújo Neves

Meta

Expor os conceitos avançados de funções e procedimentos, permitindo a construção de algoritmos mais sofisticados.

Objetivos

Esperamos que, ao final desta aula, você seja capaz de:

1. diferenciar passagem de parâmetros por valor e por referência e utilizar a passagem por referência para projetar procedimentos mais versáteis;

2. reconhecer e escrever algoritmos que se utilizem de recursão;

3. avaliar se recursão é o método mais adequado para executar uma tarefa qualquer.

Introdução

Mandar uma encomenda para uma pessoa é uma tarefa relativamente simples nos dias de hoje. Você vai até uma agência dos correios, informa o endereço onde o pacote deve ser entregue, paga as taxas do serviço e pronto: em alguns dias, a encomenda chega a seu destino. Agora, imagine mandar uma encomenda para uma pessoa sem utilizar o sistema de endereçamento.

Toda a logística de entrega depende de um sistema de referências que tem como base um ponto de partida e um destino. Sem essas referências, fazer movimentação de cargas e translado de pessoas se torna uma tarefa difícil. Dentro da computação, as referências têm um papel um pouco diferente, mas também fundamental. Elas são feitas para áreas de memória, e seu uso permite que procedimentos e funções possam alterar valores de parâmetros, de modo que essas alterações sejam mantidas após a execução. Também é possível evitar cópias desnecessárias de dados através do uso de referências.

Nesta aula, você verá, além do conceito de referência, o de recursão, ambos sofisticados no uso de funções e procedimentos. Os recursos mostrados serão de grande utilidade quando você precisar de procedimentos que devem retornar um ou mais valores e também para construir algoritmos que partem de definições matemáticas. O uso desses conceitos permite fazer alterações em variáveis passadas para procedimentos e funções, fornece maior agilidade aos algoritmos, promove drástica simplificação, bem como maior elegância e legibilidade. Portanto, o estudo deles é de fundamental importância para o aprimoramento de todos os programadores.

Passagem de parâmetros por valor e por referência

Passagem de parâmetro

Ato de utilizar valores, variáveis ou expressões como argumentos de um procedimento ou função quando ele é chamado.

Até o momento, só foram expostos exemplos com **passagem de parâmetro** por valor, ou seja, por meio de cópias criadas a partir dos parâmetros. Isso impossibilita a alteração de valores dos parâmetros originais e, portanto, impede que se possa fazer um procedimento para ler os elementos de um vetor, por exemplo. Além disso, deve-se levar em conta que, apesar de ser feita de modo transparente para os programadores, a criação de cópias demanda tanto memória quanto tempo, fazendo com que programas que fazem o uso desnecessário de parâmetros passados por valor consumam mais memória e demorem mais para concluir sua execução.

Esses problemas podem ser resolvidos através da passagem de parâmetros por referência, pois ela não cria cópias. Em vez disso, cria uma referência distinta para um elemento. A ideia é similar ao fornecimento de um endereço para entrega de um produto. Quando você faz uma compra pela internet, você simplesmente preenche um endereço de entrega (referência) para que a sua compra chegue até sua casa. Note que, apesar de o endereço não ser sua casa (o endereço é só uma informação sobre a localização da casa), ele é suficiente para que o vendedor consiga fazer a entrega do produto.

Para passar um parâmetro por referência, basta uma pequena alteração na sintaxe da lista de parâmetros dos subprogramas, colocando um *&* antes do nome do parâmetro. Desse modo, se um procedimento para ler vetores tem a seguinte assinatura *Procedimento lerVetor(int tamanho, int [] v)* e queremos que os valores de vetor sejam alterados após a execução do procedimento, devemos alterar a assinatura para *Procedimento lerVetor(int tamanho, int []&v).* A **Figura 10.1** mostra um exemplo de procedimento que utiliza passagem de parâmetros por referência para ler os elementos de um vetor.

```
Procedimento lerVetor(Inteiro tam, Inteiro [] &v)
Início
   Inteiro i
   Para(i=0;i<tam;i=i+1)Faça
      Ler v[i]
   Fim Para
Fim
```

Figura 10.1: Procedimento para ler vetor com uso de passagem por referência.

Note que nada muda no acesso ao vetor, que é lido normalmente. A **Figura 10.2** mostra um exemplo de algoritmo que utiliza o procedimento da **Figura 10.1** para ler um vetor. Observe que nenhuma sintaxe especial é necessária no momento de chamar o procedimento. Isso significa que, para usar passagem de parâmetros por referência, só é necessário modificar a assinatura da função/procedimento.

```
Algoritmo testeLerVetor( )
Início
   Inteiro[ ]c = Inteiro [6]
   lerVetor(6,c)
   Inteiro i
   Para(i=0;i<6;i=i+1)Faça
      Imprimir c[i]
   Fim Para
Fim
```

Figura 10.2: Algoritmo que utiliza um procedimento que faz uso de passagem de parâmetro por referência.

Para passar um parâmetro por referência, nenhuma sintaxe especial é requerida no corpo do subprograma em questão e nem no momento de invocar o subprograma. Apenas a assinatura é alterada, adicionando-se um & na frente do nome do parâmetro.

Nesse exemplo, o algoritmo invoca o procedimento *lerVetor*, que usa um parâmetro passado por valor (a variável *tam*) e um parâmetro passado por referência (*v*). Como *tam* é uma variável recebida por valor, pode assumir qualquer valor, mas será uma cópia do valor passado. Ao passar um parâmetro como valor (caso de *tam*), ela será uma variável que será cópia da variável passada como parâmetro ou, neste caso, *tam* é uma variável que armazenará o resultado da expressão passada como parâmetro no momento da chamada. A chamada foi a seguinte: *lerVetor(6,c)*. O resultado da expressão "6" (apesar de ter só um termo, é uma expressão, pois poderia igualmente ser escrita como 5+1, 6+0, 3*2 etc.) é copiado para uma variável chamada *tam*, que é criada no momento da chamada

do procedimento. Já o vetor *c* é um parâmetro passado por referência. Isso significa que ele não é copiado, mas que existem diferentes mecanismos para acessá-lo. Como *v* é uma referência para *c*, ao ler os elementos de *v*, o procedimento, na verdade, está lendo os valores do vetor *c*. Sempre que se cria uma variável, separa-se uma região de memória para a informação e atribui-se a essa região um nome (*c* no exemplo). Ao criar uma referência para uma variável, o que se cria é um segundo modo de identificar uma mesma região de memória (*v* no exemplo). Uma possível analogia pode ser feita com casa, endereços e coordenadas geográficas. Nesse contexto, *tam* e a expressão 6 são duas coisas iguais, mas em locais diferentes. É como ter duas casas que são exatamente iguais, mas em locais diferentes da cidade. Uma pode estar no centro e outra em um bairro afastado, mas não muda o fato de que elas são exatamente iguais e nem de que não são a mesma casa. Com referência, é um pouco diferente. São modos diferentes (endereço e coordenadas) para apontar para a mesma casa. Por exemplo, o endereço Avenida Rio Branco, 144 – Centro, Rio de Janeiro, RJ pode também ser expresso em termos de sua latitude/longitude como −22,9062 −43, 1774. Podemos usar qualquer um dos modos de identificação (endereço ou coordenadas), e estaremos nos referindo ao mesmo lugar.

Como pode ser observado nas **Figuras 10.1** e **10.2**, procedimentos/funções podem ter ao mesmo tempo parâmetros passados por referência (com o &) e parâmetros passados por valor (sem o &). A passagem por referência só é recomendada em dois contextos:

1. quando se quer alterar valores dos parâmetros para uso futuro;
2. para evitar cópias de grandes estruturas (especialmente grandes matrizes).

Nas demais situações, por segurança, usa-se passagem por valor. A segurança é uma questão de prevenir erros. Imagine a seguinte situação: existem dois vetores com nomes diferentes, mas com o mesmo tamanho, e uma variável é utilizada para armazenar esse tamanho. Se for possível alterá-la em algum procedimento, ela pode ser alterada para um valor que não reflita mais o tamanho dos vetores (um número negativo, por exemplo). Note que isso não é desejado; por isso, usamos passagem por referência somente nos casos necessários.

Como a passagem de parâmetros por referência possibilita fazer alterações nos parâmetros de entrada, utilizar esse tipo de parâmetro permite que procedimentos também possam ter retornos. Para exemplificar, suponha

que se deseja calcular as raízes reais de uma equação do segundo grau do tipo ax^2+bx+c. Para encontrá-las, podem-se utilizar as fórmulas de Bhaskara, que possibilitam que as raízes de uma equação do segundo grau sejam encontradas por meio das equações $x_1=\frac{-b+\sqrt{b^2-4ac}}{2a}$ e $x_2=\frac{-b-\sqrt{b^2-4ac}}{2a}$.

Exemplo Comentado

Faça um procedimento que utilize passagem de parâmetros por referência e que calcule corretamente as raízes reais de uma equação do segundo grau, caso existam. O procedimento deve ter seis parâmetros, sendo 3 de entrada e 3 de saída. As entradas são os coeficientes da equação do segundo grau. As saídas são as raízes e um código de retorno. O código deve conter 0, caso não existam raízes reais; 1, caso as raízes sejam iguais; 2, caso as raízes sejam diferentes.

Resposta

A primeira coisa a se fazer é analisar corretamente a lista de parâmetros descritos pelo enunciado. Os coeficientes *a*, *b* e *c* são naturalmente números reais, assim como as raízes *x1*, *x2* . O código de retorno, no entanto, é um número inteiro. Como as raízes e o código devem ter seus valores alterados, esses três devem ser passados como referência. A seguir, encontra-se uma possível solução para esse problema.

```
Procedimento raizes(Real a, Real b, Real c, Real &x1, Real &x2, Inteiro &cod)
Início
   Real delta = b*b-4*a*c
   Se(delta < 0) Então
      cod=0
   Senão
      x1=(-b+ delta^(1.0/2.0))/(2*a)
      Se (delta==0)Então
         x2=x1
         cod = 1
      Senão
          x2=(-b- delta^(1.0/2.0))/(2*a)
         cod=2
      Fim Se
   Fim Se
Fim
```

Note que é utilizada uma variável auxiliar *delta* para calcular parte das equações. Um possível algoritmo que utiliza esse procedimento é mostrado a seguir. Note que o algoritmo utiliza o procedimento com passagem por referência sem nenhuma alteração na sintaxe de chamada.

```
Algoritmo testeReferencia()
Início
   Real a,b,c,x1,x2
   Inteiro codigo
   Imprimir "Digite os coeficientes de uma equação do segundo grau"
   Ler a, b, c
   raizes (a,b,c,x1,x2,codigo)
   Se(codigo==0)Então
      Imprimir "Esta equação não possui raizes reais"
   Senão
      Imprimir "As raizes são: "+x1+ " "+x2
   Fim Se
Fim
```

Na Aula 9, vimos a primeira diferença entre os funcionamentos de funções e procedimentos: enquanto em procedimentos o fluxo vai do início ao fim, em funções, vai do início até encontrar a primeira instrução *Retorne*. A segunda diferença se refere ao momento de chamada: somente o retorno de funções pode ser usado para compor expressões. Para uma função com a assinatura *Inteiro fatorial(Inteiro n)* que calcula o fatorial de *n* e coloca o resultado do cálculo, é possível criar um algoritmo que inclua a composição *fatorial(n)/fatorial(n−1)* para encontrar o resultado da expressão *n!/(n−1)!*. No entanto, se for criado um procedimento *Procedimento fatorialP(Inteiro n, Inteiro &res)*, que calcula o fatorial de *n* e coloca o resultado em *res*, seriam necessárias duas chamadas separadas do procedimento e uma divisão dos resultados. A **Figura 10.3** mostra um algoritmo que soluciona esse caso. Como dito pelo enunciado, supõe-se que já exista um *procedimento fatorialp* para o calculo do fatorial.

```
Algoritmo testeReferencia2( )
Início
   Inteiro n, res1,res2, expressao
   Imprimir "Digite o valor de n"
   Ler n
   fatorialP(n,res1)
   fatorialP(n-1,res2)
   expressao = res1/res2
   Imprimir "O resultado da expressão é: "+expressao
Fim
```

Figura 10.3: Algoritmo que usa as respostas de procedimentos para calcular uma expressão.

Note que as chamadas do procedimento são feitas separadamente, e seus respectivos resultados usados de maneira conveniente para compor o resultado final. Para esse exemplo, o uso de funções simplificaria o algoritmo final.

Pequenas variações entre linguagens de programação

O mecanismo de passagem por referência varia de linguagem para linguagem. Em C++, este mecanismo deve ser utilizado de modo explícito, como mostrado nesta aula. Ou seja, deve-se informar o que é passado como referência e o que é passado como valor. Em Java, isto não ocorre. Variáveis e referências são passadas exatamente do mesmo modo, sem distinção de sintaxe. Ao iniciar um projeto em uma linguagem, verifique como esta linguagem trabalha com estes conceitos, para não ter problemas.

Atividade 1

Atende ao Objetivo 1

Faça um procedimento que troque os valores de duas variáveis reais, ou seja, o procedimento deve receber duas variáveis, *a* e *b*, por exemplo, e trocar o valor das duas. Suponha que, antes da chamada do procedimento, *a* vale 2.0 e *b* vale 3.14. Depois da chamada do procedimento, *a* deve valer 3.14 e *b* deve valer 2.0.

Resposta Comentada

Para resolver esta questão, é necessário entender que este procedimento tem apenas dois parâmetros (as duas variáveis a serem trocadas), ambos passados por referência. Para trocar os valores das variáveis, deve-se usar a mesma estratégia utilizada para trocar posições de vetores. Uma possível solução é mostrada a seguir:

```
Procedimento inverterValor(Real &a, Real &b)
Início
   Real aux =a
   a = b
   b=aux
Fim
```

Atividade 2

Atende ao Objetivo 1

Faça um procedimento que receba um vetor de números inteiros e seu tamanho como entrada. O procedimento deve calcular a média dos elementos do vetor e fornecer como saída quantos elementos do vetor estão abaixo da média, quantos são iguais à média e quantos são maiores que a média.

Resposta Comentada

Ao analisar o enunciado, percebe-se que este procedimento deve ter cinco parâmetros: dois de entrada e três de saída, passados através de referência. Como a quantidade de números maiores que a média é um valor inteiro, assim como a quantidade de elementos menores e iguais à média, esses parâmetros de saída são todos valores inteiros. Uma possível solução é mostrada a seguir:

```
Procedimento relacaoMedia(Real [] vet, Inteiro tam, Inteiro &abx, Inteiro &med, Inteiro &acm)
Início
   Real media = 0
   Inteiro i
   abx=0
   med = 0
   acm = 0
   Para(i=0;i<tam;i=i+1)Faça
      media = media +vet[i]
   Fim Para
   media = media/tam
   Para(i=0;i<tam;i=i+1)Faça
      Se(vet[i]==media)Então
         med = med+1
      Senão
         Se(vet[i]>media)Então
            acm = acm+1
         Senão
            abx = abx+1
         Fim Se
      Fim Se
   Fim Para
Fim
```

Recursão

Apesar de o nome talvez ser uma novidade, você certamente já utilizou alguma recursão ao longo dos seus estudos. Ela é uma técnica geral de definir algo (funções, estruturas, atividades) em função de instâncias menores. Na Matemática, essa técnica é muito utilizada na definição de funções e estruturas matemáticas. Na computação, é frequentemente utilizada na definição de subprogramas (funções e procedimentos).

Uma definição matemática recursiva muito conhecida é a definição do fatorial, que é dada a seguir:

$$n! = \begin{cases} 1, & se \ \ n \in \{0,1\} \\ n(n-1)!, & c.c. \end{cases}$$

Note que, pela definição, o problema é facilmente resolvido quando n vale 0 ou 1. Para qualquer outro valor de n, multiplica-se n pelo resultado da mesma função fatorial aplicada a $n-1$, ou seja, quando usamos a função fatorial recursivamente para resolver o fatorial de $n-1$.

Toda recursão tem, pelo menos, duas partes. Uma é chamada *caso base* e a outra *caso recursivo*. Em algumas situações, podem existir mais de um caso base e mais de um caso recursivo, mas há, pelo menos, um caso base e um caso recursivo em qualquer definição recursiva. Os casos bases são aqueles em que o problema é facilmente resolvido. Com fatorial, este caso ocorre quando n é igual a zero ou igual a um. O caso recursivo é aquele em que a própria função é usada recursivamente, pelo menos, uma vez. Normalmente, o caso recursivo utiliza a função para resolver uma instância menor do problema e utiliza o resultado obtido nessa instância menor para resolver o problema original. No fatorial, o caso recursivo ocorre quando n é maior que um, quando o resultado do fatorial de $n-1$ é calculado recursivamente e utilizado para compor uma multiplicação que resulta na solução do problema original.

A recursão provê mecanismos simples e elegantes para a solução de diversos problemas e por isso é considerada uma técnica fundamental dentro da teoria da computação. Para desenvolver uma solução recursiva, o primeiro passo é identificar os casos base e recursivo. Uma vez identificados, deve-se construir os algoritmos recursivos que implementem a solução.

A recursão em diversas linguagens

Em algumas linguagens, como C++ e Java, funções recursivas são facilmente implementadas. Entretanto, o uso de recursão em algumas linguagens, como Fortran, não é trivial. Existem inclusive algumas linguagens que são naturalmente recursivas, como Haskell. Portanto, dependendo da linguagem utilizada no projeto no qual você estiver trabalhando, utilizar algoritmos recursivos pode ser uma boa opção.

Uma possível implementação recursiva para o fatorial é dada pela **Figura 10.4.** Note que a função se aproxima bem mais da definição matemática do que as versões feitas até o momento.

```
Inteiro fatorial(Inteiro n)
Início
   Se (n==0 | n==1)Então
      Retorne 1
   Senão
      Retorne n * fatorial(n-1)
   Fim Se
Fim
```

Figura 10.4: Função recursiva para o cálculo do fatorial.

As chamadas de uma função recursiva são resolvidas em uma estratégia de empilhamento, ou seja, a última função a ser chamada é a primeira a ser resolvida. Para exemplificar o mecanismo de chamadas para uma função recursiva, será utilizada a função fatorial definida na **Figura 10.4** aplicada a *5*. Note que 5 não se encaixa no caso base, o que faz com que *fatorial(5)* seja analisado como *5*fatorial(4)*. Perceba que, para calcular o fatorial de 5, deve-se calcular o fatorial de 4, o que faz com que o fatorial de 4 seja resolvido primeiro. Esse *fatorial(4)* resulta em uma nova chamada da função fatorial, em que o parâmetro também não se encaixa no caso base, o que resultará em *4*fatorial(3)*. O *fatorial(3)* resulta em *3*fatorial(2)* que, por sua vez, resulta em *2*fatorial(1)*. Nesse ponto, a recursão encontra um caso base e retorna um resultado que é retornado para o ponto onde a

chamada recursiva foi feita. Nesse caso, *2*fatorial(1)* é avaliado como 2*1 que resulta em 2. Esse resultado é retornado para o ponto onde o fatorial de 2 foi chamada. Neste caso, *3*fatorial(2)* é avaliado como 3*2 que resulta em 6. Esse processo de retornos sucessivos continua até que todas as chamadas recursivas tenham sido resolvidas e o resultado final do fatorial seja calculado. A **Figura 10.5(a)** mostra o processo de chamadas recursivas para a função fatorial. Já a **Figura 10.5(b)** mostra o processo de retorno na pilha de recursão.

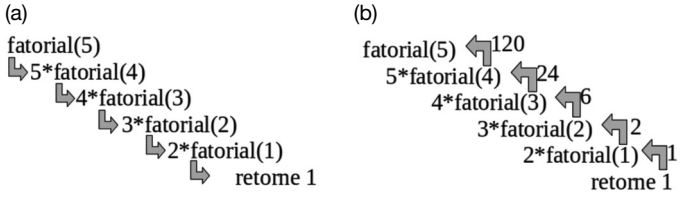


Figura 10.5: Pilha de execução: chamadas(a) e retornos(b).

Um outro exemplo clássico de uso de recursão é o cálculo da série de Fibonacci. A definição matemática dos termos da série é dada a seguir:

$$fib(n) = \begin{cases} 1, sen \in \{1,2\} \\ fib(n-1) + fib(n-2), c.c. \end{cases}$$

Note que, assim como o fatorial, a série de Fibonacci tem um caso base e um caso recursivo, porém, no caso recursivo, existem duas chamadas à função *fib*, uma usando *n–1* como parâmetro e outra usando *n–2*.

Uma possível implementação de uma função para o cálculo da série de Fibonacci é mostrada na **Figura 10.6(a)**, bem como uma versão iterativa (que usa estrutura de repetição) vista anteriormente **(b)**. Note que, além de ser menor, a versão recursiva se aproxima mais da definição matemática e também é mais simples que a versão iterativa.

(a)

```
Inteiro fib(Inteiro n)
Início
   Se (n==1 | n==2)Então
      Retome 1
   Senão
      Retome fatorial(n-1) + fib(n-2)
   Fim Se
Fim
```

(b)

```
Inteiro fibIterativo(Inteiro n)
Início
   Inteiro res=1, fib1=1,fib2=1,i
   Para(i=3;i<=n;i=i+1)Faça
      res = fib1+fib2
      fib2=fib1
      fib2=res
   Fim Para
   Retorne res
Fim
```

Figura 10.6: Funções para o cálculo da série de Fibonacci: recursiva a) e iterativa(b).

Apesar de a recursão oferecer uma maneira simples e eficiente para solucionar problemas, costuma ser menos eficiente, principalmente pela questão do retrabalho. A versão recursiva para o cálculo da série de Fibonacci é um bom exemplo. A **Figura 10.7** mostra todas as chamadas feitas para o cálculo do quinto termo da série. Note que, para concluir o cálculo de *fib(5)*, é necessário resolver duas vezes *fib(3)* e três vezes *fib(2)*.

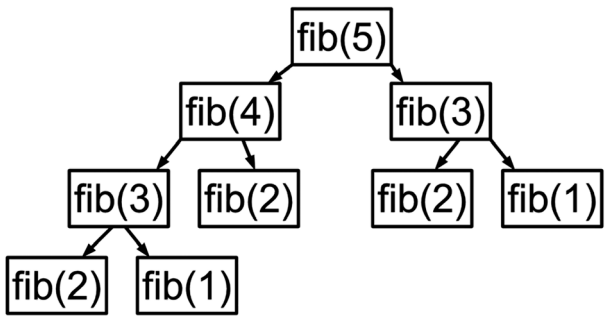


Figura 10.7: Análise das chamadas recursivas para o cálculo da série de Fibonacci.

É importante mencionar que, mesmo em casos em que não há retrabalho, versões recursivas de funções ainda são mais lentas quando implementadas em computadores, pelo fato de chamadas de função serem avaliadas mais lentamente do que operações matemáticas convencionais. Como existem teoremas que afirmam que todo algoritmo recursivo pode ser descrito de forma iterativa e vice-versa, o programador deve avaliar o impacto do uso de recursão nos seus algoritmos. Caso esse impacto seja relativamente pequeno, em vista da simplicidade proporcionada na solução, o uso de recursão pode ser uma boa opção.

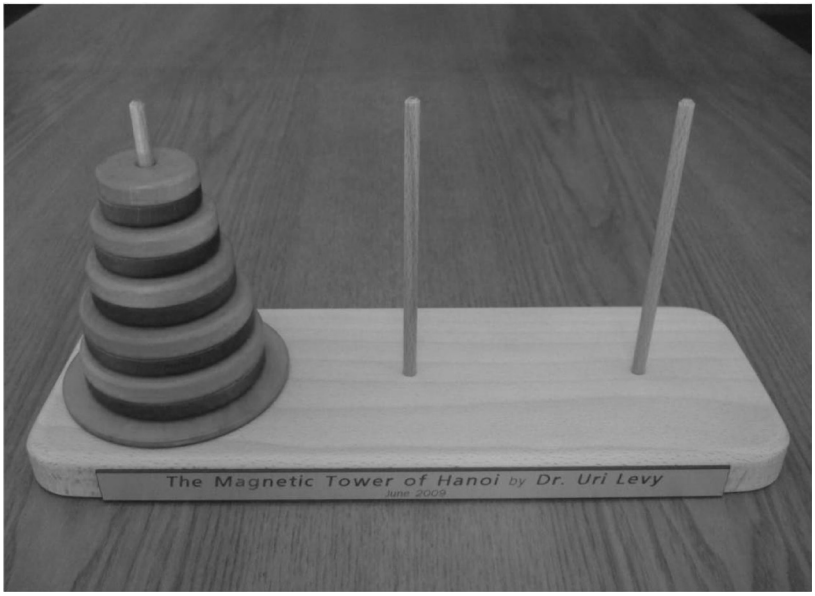
Ao longo desta aula, foi mostrado como o uso de passagem de parâmetros por referência e a recursão podem ajudar na simplificação de algoritmos. A rigor, todos os parâmetros podem ser passados como referência para procedimentos e funções. Isso, de fato, aumenta a eficiência dos algoritmos, pois evita cópias desnecessárias de matrizes, assim como permite alterar valores de variáveis, de modo que essas alterações repercutam fora dos, subprogramas. Porém, permitir que certos valores sejam alterados pode causar grandes problemas durante a execução. Sempre que você perceber que um parâmetro deve ser usado, mas não alterado por um subprograma, certifique-se de que ele esteja sendo passado por valor (sem o &). Isso tornará seus algoritmos e funções mais seguros em relação a falhas.

Quanto à recursão, o único inconveniente para seu uso é o desempenho. Subprogramas recursivos são mais lentos que subprogramas iterativos. Se você não está preocupado com a eficiência dos seus algoritmos, o uso de recursão pode trazer grande simplificação. Agora, se a eficiência é um fator crítico, a recursão não é recomendada. Para esta disciplina, a eficiência de um algoritmo não é um dos critérios de avaliação. O objetivo aqui é fazer com que você desenvolva sua lógica. Portanto, caso você proponha uma solução recursiva para um problema e ela esteja correta, será considerada tão boa quanto uma solução iterativa.

Atividade 3

Atende aos Objetivos 1, 2 e 3

A torre de Hanói é um brinquedo muito conhecido. Existem três pinos e uma série de discos de diâmetros diferentes. O objetivo do jogo é mudar todos os discos de um pino para outro, podendo usar o terceiro pino como auxiliar, sem que um disco de diâmetro maior seja empilhado sobre um disco de diâmetro menor, embora discos de diâmetro menor possam ser empilhados sobre discos de diâmetro maior.



Uri-Levy

Fonte: http://commons.wikimedia.org/wiki/File:Free_MToH_five_disks_JPEG_110924.jpg

Faça uma função em que, dado um número *n* de discos a serem transferidos, deve-se calcular quantos movimentos são necessários para completar o jogo, sabendo-se que um movimento compreende tirar um disco de um pino e colocar em outro. Sabe-se que o número de movimentos necessários é dado por *h(n)*, em que *n* é o número de discos e segue a seguinte relação:

$$h(n) = \begin{cases} 1, se\ n = 1 \\ 1 + 2h(n-1), c.c. \end{cases}$$

Resposta Comentada

Seguindo a definição recursiva e os conteúdos passados, esta questão tem implementação simples. Uma possível solução é:

```
Inteiro numHanoi(Inteiro discos)
Início
   Se(discos == 1) Então
      Retorne 1
   Senão
      Retorne 1+ 2*numHanoi(discos-1)
   Fim Se
Fim
```

Se quiser jogar a torre de Hanói, acesse: http://www.softschools.com/games/logic_games/tower_of_hanoi/.

O site é em inglês, mas o jogo não depende da língua. Como descrito acima, o objetivo é mudar todos os discos de um pino para outro sem que um disco de diâmetro maior seja empilhado sobre um disco de diâmetro menor.

Atividade 4

Atende aos Objetivos 1, 2 e 3

Faça um procedimento que receba um vetor de números reais, o tamanho do vetor e um número inteiro. O procedimento deve ordenar os elementos do vetor. Se o valor do número inteiro passado for maior que zero, a ordem deve ser crescente. Se o número for menor ou igual a zero, a ordem deve ser decrescente.

Resposta Comentada

Nesta questão, o único parâmetro que precisa ser passado como referência é o vetor, pois seus valores serão alterados. Uma possível solução é mostrada abaixo. Nela, foi utilizado o procedimento definido na Atividade 3 para trocar os valores. A solução pode ser dividida em duas partes, marcadas pelos comentários, uma para ordenar em ordem crescente e outra para ordenar em ordem decrescente.

```
Procedimento ordenar(Real[] &vet, Inteiro tam, Inteiro ordem)
Início
   Inteiro i,j
   Se (ordem>0)Então /*ordem crescente*/
      Para(i=0;i<tam-1;i=i+1)Faça
         Para(j=tam-1;j>i;j=j-1)Faça
            Se (vet[j]<vet[j-1])Então
               inverterValor(vet[j],vet[j-1])
            Fim Se
         Fim Para
      Fim Para
   Senão  /*ordem decrescente*/
      Para(i=0;i<tam-1;i=i+1)Faça
         Para(j=tam-1;j>i;j=j-1)Faça
            Se (vet[j]>vet[j-1])Então
               inverterValor(vet[j],vet[j-1])
            Fim Se
         Fim Para
      Fim Para
   Fim Se
Fim
```

Atividade 5

Atende aos Objetivos 1, 2 e 3

O máximo divisor comum (MDC) entre dois números inteiros *a* e *b* pode ser denotado por mdc (a,b) e pode ser obtido pela relação descrita abaixo. Faça uma função que calcule o MDC recursivamente, de acordo com essa definição:

$$mdc(a,b) = \begin{cases} a, \text{ se } b = 0 \\ mdc(b, a\%b), \text{ c.c.} \end{cases}$$

Resposta Comentada

Uma vez estabelecido o padrão da recursão, a solução deste problema é bem mais simples que a solução do mesmo problema apresentada na Aula 6.

```
Inteiro mdcRec(Inteiro a, Inteiro b)
Início
   Se(b==0)Então
      Retorne a
   Senão
      Retorne mdcRec(b,a%b)
   Fim Se
Fim
```

Resumo

Nesta aula, você aprendeu a lidar com passagem de parâmetros por referência, técnica que permite alterar o valor de parâmetros, fazendo com que procedimentos possam ter saídas. Também aprendeu a lidar com a técnica de recursão, que permite utilizar uma função dentro dela mesma, tornando a definição de algumas funções mais simples.

Ao longo do curso, você viu várias técnicas de construção de algoritmos. Todas as técnicas mostradas são válidas na maioria das linguagens de programação imperativas, de modo que agora é hora de começar a utilizar o seu conhecimento na construção de programas de computador, ou seja, já é hora de começar a transformar seus algoritmos em programas. A disciplina Computação 2, continuação natural desta, irá ajudar nessa tarefa.

Boa sorte!

Referências

Aula 1

ASCENCIO, A.F.G.; CAMPOS, E.A.V. *Fundamentos da programação de computadores*. São Paulo: Pearson, 2012.

CORMEN, T.H. et al. *Introduction to Algorithms*. 3. ed. Cambridge: MIT Press, 2009.

FARRER, H. et al. *Programação estruturada de computadores*. 2. ed. Rio de Janeiro: Guanabara, 1989.

MONTEIRO, M.A. *Introdução à organização de computadores*. 4. ed. Rio de Janeiro: LTC, 2002.

Aula 2

ASCENCIO, A.F.G.; CAMPOS, E.A.V. *Fundamentos da programação de computadores*. São Paulo: Pearson, 2012.

CORMEN, T.H. et al. *Introduction to Algorithms*. 3. ed. Cambridge: MIT Press, 2009.

DEITEL, P.; DEITEL, H.C. *Como programar*. 6. ed. São Paulo: Pearson, 2011.

FARRER, H. et al. *Programação estruturada de computadores*. 2. ed. Rio de Janeiro: Guanabara, 1989.

Aula 3

ASCENCIO, A.F.G.; CAMPOS, E.A.V. *Fundamentos da programação de computadores*. São Paulo: Pearson, 2012.

CORMEN, T.H. et al. *Introduction to Algorithms*. 3. ed. Cambridge: MIT Press, 2009.

DEITEL, P.; DEITEL, H.C. *Como programar*. 6. ed. São Paulo: Pearson, 2011.

FARRER, H. et al. *Programação estruturada de computadores*. 2. ed. Rio de Janeiro: Guanabara, 1989.

Aula 4

ASCENCIO, A.F.G.; CAMPOS, E.A.V. *Fundamentos da programação de computadores*. São Paulo: Pearson, 2012.

CORMEN, T.H. et al. *Introduction to Algorithms*. 3. ed. Cambridge: MIT Press, 2009.

DEITEL, P.; DEITEL, H.C. *Como programar*. 6. ed. São Paulo: Pearson, 2011.

FARRER, H. et al. *Programação estruturada de computadores*. 2. ed. Rio de Janeiro: Guanabara, 1989.

Aula 5

ASCENCIO, A.F.G.; CAMPOS, E.A.V. *Fundamentos da programação de computadores*. São Paulo: Pearson, 2012.

CORMEN, T.H. et al. *Introduction to Algorithms*. 3. ed. Cambridge: MIT Press, 2009.

DEITEL, P.; DEITEL, H.C. *Como programar*. 6. ed. São Paulo: Pearson, 2011.

FARRER, H. et al. *Programação estruturada de computadores*. 2. ed. Rio de Janeiro: Guanabara, 1989.

Aula 6

ASCENCIO, A.F.G.; CAMPOS, E.A.V. *Fundamentos da programação de computadores*. São Paulo: Pearson, 2012.

CORMEN, T.H. et al. *Introduction to Algorithms*. 3. ed. Cambridge: MIT Press, 2009.

DEITEL, P.; DEITEL, H.C. *Como programar*. 6. ed. São Paulo: Pearson, 2011.

FARRER, H. et al. *Programação estruturada de computadores*. 2. ed. Rio de Janeiro: Guanabara, 1989.

Aula 7

ASCENCIO, A.F.G.; CAMPOS, E.A.V. *Fundamentos da programação de computadores*. São Paulo: Pearson, 2012.

CORMEN, T.H. et al. *Introduction to Algorithms*. 3. ed. Cambridge: MIT Press, 2009.

DEITEL, P.; DEITEL, H. *Como programar*. 6. ed. São Paulo: Pearson, 2011.

FARRER, H. et al. *Programação estruturada de computadores*. 2. ed. Rio de Janeiro: Guanabara, 1989.STEINBRUCH, A.; WINTERLE, P. *Geometria analítica*. 2. ed. São Paulo: McGraw-Hill, 1987.

Aula 8

ASCENCIO, A.F.G.; CAMPOS, E.A.V. *Fundamentos da programação de computadores*. São Paulo: Pearson, 2012.

CORMEN, T.H. et al. *Introduction to Algorithms*. 3. ed. Cambridge: MIT Press, 2009.

DEITEL, P.; DEITEL, H. *Como programar*. 6. ed. São Paulo: Pearson, 2011.

FARRER, H. et al. *Programação estruturada de computadores*. 2. ed. Rio de Janeiro: Guanabara, 1989.

STEINBRUCH, A.; WINTERLE, P. *Geometria analítica*. 2. ed. São Paulo: McGraw-Hill, 1987.

Aula 9

ASCENCIO, A.F.G.; Campos, E.A.V. *Fundamentos da programação de computadores*. São Paulo: Pearson, 2012.

CORMEN, T.H. et al. *Introduction to Algorithms*. 3. ed. Cambridge: MIT Press, 2009.

DEITEL, P.; DEITEL, H.C. *Como programar*. 6. ed. São Paulo: Pearson, 2011.

FARRER, H. et al. *Programação estruturada de computadores*. 2. ed. Guanabara, 1989.STEINBRUCH, A.; WINTERLE, P. *Geometria analítica*. 2. ed. McGraw-Hill, 1987.

Aula 10

ASCENCIO, A.F.G.; CAMPOS, E.A.V. *Fundamentos da programação de computadores*. São Paulo: Pearson, 2012.

CORMEN, T.H. et al. *Introduction to Algorithms*. 3. ed. Cambridge: MIT Press, 2009.

DEITEL, P.; DEITEL, H. *Como programar*. 6. ed. São Paulo: Pearson, 2011.

FARRER, H. et al. *Programação estruturada de computadores*. 2. ed. Rio de Janeiro: Guanabara, 1989.

STEINBRUCH, A.; WINTERLE, P. *Geometria analítica*. 2. ed. São Paulo: McGraw-Hill, 1987.