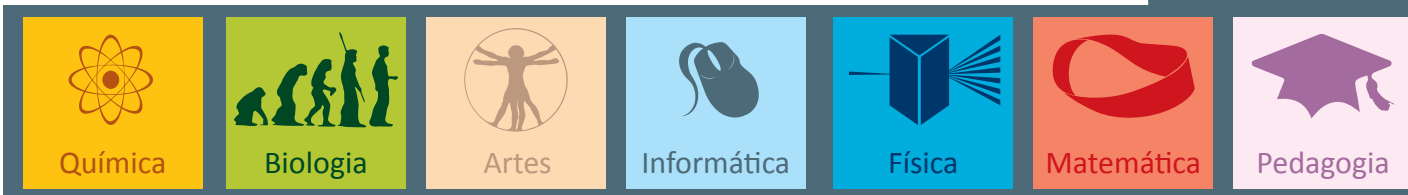
Informática

Algoritmos e Programação

Ricardo Reis Pereira
Jerffeson Teixeira de Souza
Jeandro de Mesquita Bezerra

3ª edição
Fortaleza
2013



Presidenta da República
Dilma Vana Rousseff

Ministro da Educação
Aloizio Mercadante

Presidente da CAPES
Jorge Almeida Guimarães

Diretor de Educação a Distância da CAPES
João Carlos Teatini de Souza Climaco

Governador do Estado do Ceará
Cid Ferreira Gomes

Reitor da Universidade Estadual do Ceará
José Jackson Coelho Sampaio

Pró-Reitora de Graduação
Marcília Chagas Barreto

Coordenador da SATE e UAB/UECE
Francisco Fábio Castelo Branco

Coordenadora Adjunta UAB/UECE
Eloísa Maia Vidal

Diretor do CCT/UECE
Luciano Moura Cavalcante

Coordenadora da Licenciatura em Informática
Francisco Assis Amaral Bastos

Coordenadora de Tutoria e Docência em Informática
Maria Wilda Fernandes

Coordenadora Editorial
Rocylânia Isidio

Projeto Gráfico e Capa
Roberto Santos

Ilustrador
Mikael Baima

Diagramador
Francisco José da Silva Saraiva

Secretaria de Apoio às Tecnologias Educacionais
Av. Paranjana, 1700 - Campus do Itaperi - Fortaleza - Ceará
(85) 3101-9962

Sumário

Capítulo 1

Fundamentos

Objetivos

- Apresentar o conceito de algoritmos e suas formas básicas de representação: texto, fluxograma e pseudocódigo. Outros fatores importantes são: entradas e saída, seguidas de outros fundamentos como variáveis.

1. A idéia de algoritmo

Para realizar uma viagem é primordial fazer o seu planejamento. De onde se partirá e qual o destino? Que meios de transporte serão usados, qual o custo e tempo que cada um leva? São necessárias novas roupas? Se mais de um lugar foi visitado no percurso quanto tempo se ficará e o que se visitará em cada um deles? Onde se ficará em cada lugar, em um hotel, uma pousada ou um albergue? Tendo-se essas informações (entre outras) pode-se colocá-las em papel para estimar o custo total e ver a viabilidade da viagem. Mas se imprevistos acontecerem? Uma lista de planos alternativos poderá evitar muitas dores de cabeça.

Partir de uma receita para se preparar alguma coisa é rotina comum em culinária. Quem escreve tais receitas lista o que é necessário, tanto em ingredientes quanto em utensílios além, claro, nas etapas que devem ser seguidas. Seguir a risca, entretanto, pode às vezes ser inviável por que algum ingrediente poderá não estar disponível (um alternativo poderá ser usado). A boa receita deve prever as faltas potenciais para sua realização e documentá-las.

Montar um novo equipamento, que acabou de chegar pelos correios numa grande caixa, pode não ser uma tarefa muito fácil principalmente quando as instruções de montagem não são bem documentadas ou fracamente ilustradas. Boas instruções vêm em passos escritos em blocos separados e ao lado de gravuras legíveis e de fácil compreensão. Em muitos casos o fabricante fornece, para auxiliar seus clientes, uma listagem das ferramentas apropriadas (quando necessárias). Bicicletas, móveis e alguns brinquedos ilustram bem esta situação.

Pessoas que tem pouco tempo a perder precisam de uma agenda (e às vezes também uma secretária) para darem conta de todas as suas obrigações. As tarefas ou compromissos são listados por data e hora. Imprevistos

obviamente podem acontecer e desta forma requerem decisões tais como cancelamento de algumas etapas ou remarcação para outro dia. Executivos e homens de negócio conhecem bem a necessidade do agendamento.

As situações descritas ilustram a idéia de **algoritmo**. Um algoritmo é um conjunto de passos finitos que devem ser seguidos para que certo objetivo seja alcançado. Algoritmos têm natureza teórica e correspondem a planejamentos estruturados (e preferencialmente otimizados) de algo que se deseja realizar, como fazer uma viagem, preparar um prato especial, montar uma bicicleta ou solucionar as pendências de uma empresa. Para cumprir um algoritmo **decisões** precisam ser tomadas, ações precisam ser realizadas e **recursos** restritos precisam ser gerenciados. O processo de realização de um algoritmo é conhecido como execução do algoritmo.

O planejamento de uma viagem impresso em papel é seu algoritmo ao passo que a viagem em si é sua execução. Os recursos são dinheiro, transportes, hotéis, roupas, guias e etc. As decisões ocorrem em diversos níveis do processo, como por exemplo, a seqüência de visitas das obras de arte num certo museu.

A receita de um bolo numa sessão culinária de revista é um algoritmo. Os recursos para sua execução são ingredientes, utensílios em geral, forno, etc. A execução é o trabalho do cozinheiro. Decisões diversas podem ser tomadas neste processo como a simples complementação de um ingrediente por um alternativo ou o aumento do tempo em forno devido o bolo não ter alcançado o aspecto ou textura desejados.

As instruções de montagem de uma bicicleta exemplificam outro algoritmo. Neste caso os recursos são peças, ferramentas e etc. A execução é o trabalho do montador. Um exemplo comum de decisão deste processo é abandonar certa etapa somente quando há certeza da fixação mecânica das partes daquela etapa.

Cada data em uma agenda representa um algoritmo para aquela data. A execução é o cumprimento de todos os compromissos durante o dia. Dependendo do portador da agenda os recursos podem variar (carro, dinheiro, roupas apropriadas, computador portátil, secretária, escritório, celular e etc.). O trabalho de um executivo é praticamente tomar decisões para cumprimento de seu agendamento.

Um **algoritmo computacional** é aquele construído baseado nas capacidades e recursos disponíveis de um computador. Os recursos disponíveis em computadores modernos são representados por seu hardware, ou seja, a CPU, a memória principal, a memória de armazenamento e os periféricos de entrada/saída. Dificilmente estes recursos são acessados diretamente porque

um software especial, o **sistema operacional**, gerencia o uso deles de forma a equilibrar a demanda dos softwares em execução.

Um algoritmo computacional requer a definição de uma **linguagem** para que possa ser expresso. Essa linguagem consiste em um conjunto relativamente pequeno de regras pré-estabelecidas, denominadas **sintaxe** da linguagem, que devem ser mantidas em todo o corpo construtivo de um algoritmo computacional. O texto em questão visa apresentar uma sintaxe simples para a criação de algoritmos computacionais baseada em outras metodologias de criação de algoritmos que investigamos nos últimos anos. Daqui para frente trataremos algoritmos computacionais simplesmente por algoritmos.

Um algoritmo necessita ser transformado em um **programa** para que possa ser executado por um computador. Essa transformação consiste numa **tradução** do algoritmo para uma **linguagem de programação**. Uma linguagem de programação é, como uma metodologia de ensino de algoritmos, um conjunto de regras. Entretanto a sintaxe de tais linguagens é mais complexa e rígida e requer estudo mais aprofundado (Sebesta, 2007).

Denomina-se **compilador** a um software capaz de reconhecer um arquivo de texto contendo um programa escrito numa dada linguagem de programação e traduzi-lo num código em **linguagem de máquina** para que possa ser finalmente executado pelo computador. Se um algoritmo é expresso numa linguagem X hipotética, então precisa de um compilador X para ser reconhecido e traduzido (processo de **compilação**). Um compilador hipotético Y, por exemplo, não reconhece códigos escritos na linguagem X. A compilação não é a única metodologia prática para execução de um programa. O leitor ainda pode investigar a **interpretação pura** e **interpretação híbrida** em (Sebesta, 2007). A metodologia preferencial neste texto é a compilação.

Há inúmeras linguagens de programação projetadas para fins diversos que vão entre criação de sistemas operacionais a desenvolvimento de aplicações Web. Bom domínio sobre construção de algoritmos é pré-requisito necessário para o ingresso no estudo de uma linguagem de programação.

2. Construção de algoritmos

Há muitas formas de se representar um algoritmo. As mais comuns são por **texto livre**, por **fluxogramas** e por **pseudocódigos**. Num algoritmo em texto livre a idéia é expressa utilizando-se todos os recursos existentes numa linguagem natural específica (português, inglês, etc.). Um exemplo comum desta abordagem são as receitas de bolo normalmente apresentadas em dois parágrafos: aquele com a lista de ingredientes e outro descrevendo a prepara-

ção passo a passo. A lista de ingredientes corresponde à fase **declarativa** do algoritmo, ou seja, a parte que antecede o próprio algoritmo e visa buscar os recursos necessários a sua realização.

Algoritmos representados por fluxogramas utilizam obviamente a estratégia gráfica para se expressarem. Cada uma das etapas (ou grupo de etapas) é representada por um elemento geométrico (retângulo, círculo, etc.). Todos os elementos constituintes são então conectados logicamente por arestas orientadas (setas). Efetuar cada uma das ações requeridas nos elementos geométricos na seqüencia das setas corresponde à execução do algoritmo. O fluxograma a seguir ilustra um algoritmo simples para fazer uma viagem a cidade de Fortaleza (os retângulos representam ações e os losangos decisões).

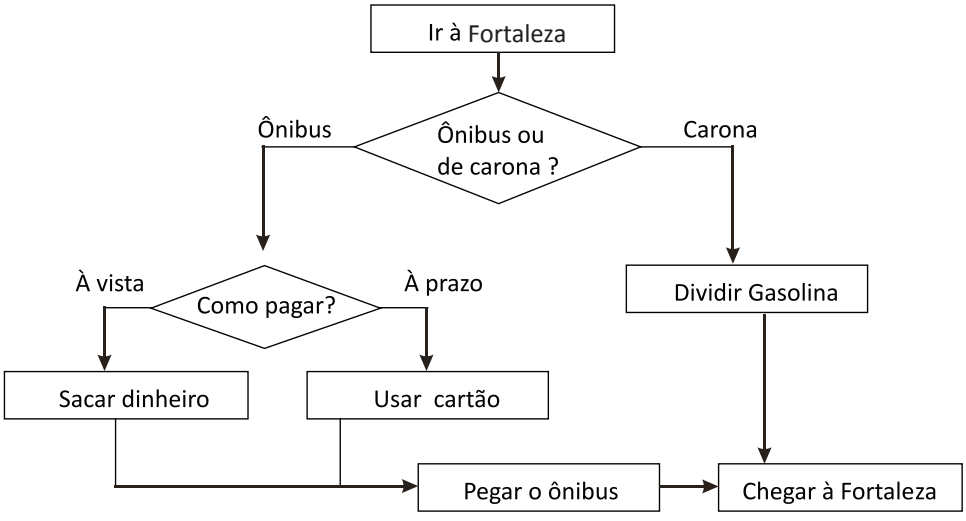


Figura 1 -

A construção de algoritmos via pseudocódigos é a preferencial deste texto. Nesta representação são usados recursos de linguagem natural, mas com quantidade bastante reduzida de vocábulos e estruturas para melhor assimilação. Os vocábulos desta metodologia são divididos em duas categorias: de aplicação pré-determinada, chamadas palavras reservadas[1] e de definição customizada (durante criação do algoritmo), mas com regras pré-estabelecidas. Palavras reservadas podem ser **comandos** ou **elementos de estruturas.** Os comandos são invocações de tarefas específicas nos algoritmos, por exemplo, um comando de impressão deve imprimir uma mensagem de texto. A invocação de um comando é conhecida como **chamada** do comando. Os elementos de estruturas associam-se nas **estruturas de controle** para efetuar decisões ou repetições (dependendo do contexto). Importantes comandos e estruturas de controle serão estudados ao decorrer deste texto.

[1] Palavras reservadas
Em programação de computadores, uma palavra reservada é uma palavra que, em algumas linguagens de programação, não pode ser utilizada como um identificador por ser reservada para uso da gramática da linguagem. Por exemplo, na linguagem de programação PASCAL, um programador não pode utilizar uma variável com o nome read pois esta palavra é reservada para a leitura de variáveis. Ela é uma "palavra chave", e por ser de uso restrito, é também uma "palavra reservada".

As **regras de sintaxe** de um algoritmo tratam-se das restrições pré-estabelecidas para possibilitar a criação de pseudocódigos. Neste texto são definidas várias regras de sintaxe e mantidas até o final. É bom lembrar que tais regras não são universais, mas que em qualquer apresentação de metodologia algorítmica, um conjunto de regras de sintaxe se mostrará necessário para garantir concisão e clareza dos algoritmos.

A palavra **semântica** será esporadicamente utilizada para se referir ao significado lógico dos elementos ou estruturas que constituem um algoritmo. Isso é importante porque muitas construções feitas não são intuitivas e a compreensão só será possível se houver prévio conhecimento do que elas se propõem a fazer.

Algoritmos em pseudocódigo apresentam-se como um texto contendo diversas linhas de código. Uma mesma linha pode conter um ou mais comandos ou ainda estruturas de controle. Várias linhas de código podem estar encapsuladas por um conceito lógico denominado **bloco**. O código de um bloco possui dessa forma início e fim e representa um sub-processo do processo geral de execução do algoritmo. Veja o exemplo esquemático:

```
//bloco de instruções
instrução 1
instrução 2
instrução 3
```

Um mesmo algoritmo pode conter diversos blocos seriados ou blocos dentro de outros blocos. Se um bloco está dentro de outro dizemos que há **aninhamento** de blocos. O aninhamento é construído por **indentação**, ou seja, colocando-se recuos no texto-código para produzir uma hierarquia. Veja o exemplo esquemático:

```
//bloco 1
instrução 1-1
instrução 1-2
   // bloco 2 indentado
   instrução 2-1
   instrução 2-2
   instrução 2-3
instrução 1-3
```

Neste exemplo o bloco 2 é parte integrante do bloco 1, mas tem significado lógico independente dentro deste bloco e desta forma é indentado. Vários níveis de indentação são permitidos num mesmo pseudocódigo. Mais tarde será estudado como a indentação auxilia fortemente na legibilidade de algoritmos.

Blocos também permitem a criação de **módulos**. Módulos são blocos do pseudocódigo com ações a executarem através de uma chamada, ou seja, são comandos personalizados. Um mesmo algoritmo pode possuir diversos módulos onde um deles, exatamente aquele onde a execução principia, é chamado **bloco principal** ou **função principal**. Até o estudo dos módulos, os pseudocódigos apresentados neste texto serão constituídos apenas do bloco principal.

O par de barras //, utilizado nos exemplos esquemáticos anteriores, marca a linha como comentário de forma que a mensagem que ele precede possui apenas valor informativo.

3. Execução de algoritmos

Executar um algoritmo significa executar seqüencialmente cada uma das linhas que formam o bloco principal onde pode haver dezenas ou mesmo centenas de linhas. Não necessariamente todas as linhas serão processadas. Poderão ocorrer as seguintes possibilidades: (i) algumas linhas poderão ser **saltadas**; (ii) um grupo de linhas poderá ser diversas vezes **repetido** (iii) um comando ou módulo poderá ser chamado principiando o processamento de linhas de outro bloco processamento; (iv) a execução é encerrada porque a última linha foi executada ou porque foi **suspenso** explicitamente.

O salto e a repetição em algoritmos serão estudados mais adiantes. Para suspender um algoritmo antes da última linha utilizaremos o comando **pare**, como segue.

```
instrução 1
instrução 2
pare
instrução 3
instrução 4
```

Neste exemplo a execução de **pare** implica suspensão imediata de processamento (sem realização das duas últimas linhas).

Linhas são elementos construtivos de textos e normalmente tem significado impreciso em algoritmos. Ao invés de linhas consideraremos o conceito de **instruções**. Uma instrução corresponde a cada porção do pseudocódigo tomada para execução. Comumente há uma instrução por linha (o que

torna linhas e instruções sinônimas), entretanto há instruções montadas normalmente utilizando mais de uma linha. No restante deste texto abordaremos algoritmos como listagens de instruções e não mais de linhas.

4. Entrada e saída

4.1. Dispositivos de e/s

Dispositivos de entrada e saída, ou simplesmente E/S, são todos e quaisquer hardwares responsáveis pelo fornecimento ou leitura de informações do/para o computador. A forma mais tradicional de entrada de dados ainda é a alfanumérica e é feita através de teclados. Devido sua grande importância, definiremos neste texto os teclados como sendo a **entrada padrão** dos programas. Similarmente a mais tradicional saída de dados é visual e ocorre via monitores (naturalmente associados às placas de vídeo). Definiremos a associação monitor/placa de vídeo como sendo a **saída padrão**.

4.2. Entrada padrão

Como definimos anteriormente, o teclado é o dispositivo padrão de entrada. Todas as informações fornecidas por teclado são alfanuméricas, ou seja, letras, números e alguns símbolos. O processo de entrada de dados através de teclados funciona da seguinte forma: Cada tecla pressionada gera um caractere que é armazenado num *buffer* mantido por algum software em execução (e que obviamente detém temporariamente o controle sobre o teclado).

Quando uma tecla de confirmação é por fim pressionada (normalmente ENTER em teclados convencionais), ocorre a transferência da informação contida no buffer (esvaziamento de buffer) para seu destino final. Muitas vezes essa transferência demanda algum processamento adicional: isso ocorre porque toda informação recebida é de fato textual e assim o computador precisa identificar e transformar parte deste texto em outro tipo de informação.

Por exemplo, se um usuário digita "*produto 25*, *custo 9.78*", de fato 22 caracteres, incluindo vazios, é que serão recebidos; entretanto se algum processamento adicional estiver associado, então o inteiro 25 e o real 9.78 poderão ser extraídos.

A sintaxe que evoca a entrada padrão será construída com a palavra reservada, leia, como no exemplo:

```
leia x
```

Observe que todos os detalhes funcionais do dispositivo físico são ocultados e simplificados num único comando imperativo (neste caso, "leia"). Mais detalhes sobre entrada de dados com **leia** serão vistos mais adiante.

4.3. Saída padrão

Como já definido, a associação placa de vídeo/monitor é a saída padrão. Em computadores mais antigos os monitores eram capazes apenas de mostrar texto: a resolução da tela restringia-se a um plano cartesiano de poucas colunas por poucas linhas onde em cada célula desta grade invisível era possível escrever um caractere apenas.

Quando a tela enchia os caracteres começavam a "rolar" para cima num aparente efeito de paginação. Cada programa posto em execução tinha a tela como alvo de sua saída (ou em outros casos a impressora) de forma que é fácil, neste contexto, associar o monitor como um todo à saída visual de um programa. Nos dias atuais o conceito de saída tornou-se sofisticado: o sistema operacional normalmente consegue manter diversas janelas gráficas cada uma delas associada a um programa em execução.

Cada uma destas janelas gerencia de forma independente entrada e saída do programa a que estão ligadas. Quando uma janela contém o **foco** (como usualmente se diz), significa que esta janela está à frente de todas as outras (em termos de visibilidade) e ainda que qualquer entrada via teclado será recebida por ela.

Devido à natureza gráfica das janelas flutuantes nos sistemas operacionais modernos, existem cada vez mais formas poderosas de interagir com o aplicativo. De fato atualmente investem-se bastante em interfaces cada vez mais inteligentes que facilitem a interação do usuário com o computador.

Neste estudo de algoritmos não serão considerados detalhes sobre as modernas interfaces gráficas de saída. Ao invés disso consideraremos todas as saídas do algoritmo sendo escritas para uma janela hipotética cujo comportamento é idêntico ao dos antigos monitores, ou seja, trata-se de um contêiner alfanumérico para onde são enviadas mensagens de texto: se a janela enche um efeito de paginação se procede empurrando os caracteres para cima de forma que sempre o último conteúdo impresso ficará disponível na parte inferior da janela.

A sintaxe que evoca o envio de dados para a saída padrão é construída aqui com a palavra reservada, **escreva**, como no exemplo:

```
escreva 'olá mundo!!'
```

Este exemplo ilustra o uso mais simples de **escreva**: a impressão de mensagens. Uma mensagem é simplesmente um agregado de caracteres entre aspas simples. Uma mensagem aparecerá na saída padrão tal como foi escrita no pseudocódigo. Mais adiante sintaxe adicional de **escreva** será apresentada para impressão de outros tipos de informação.

5. Variáveis

5.1. A memória principal e as variáveis

A memória principal é a memória volátil do computador, ou seja, só armazenará alguma informação enquanto corrente elétrica cruzar seus circuitos (A memória projetada para manter informações, mesmo com o equipamento desligado, é a memória secundária, como por exemplo, os discos rígidos). Na memória principal serão colocados os programas para que sejam executados. Estes por sua vez precisarão de memória adicional para realizar suas tarefas. Nos computadores modernos muitos programas podem estar em execução ao mesmo tempo e assim concorrerem pela mesma memória livre no momento em que suas atividades requererem mais espaço.

A problemática geral sobre o **gerenciamento de memória** em computadores é a seguinte: como impedir que programas distintos não sejam gravados na mesma área de memória e ainda, como impedir que eles concorram pela mesma região livre de memória no momento em que requerem espaço adicional? De fato esta responsabilidade é do sistema operacional através do **gerenciador de memória**.

Os projetistas de um sistema operacional devem prever que os programas, que rodarão sobre ele, farão tais exigências em relação à memória principal. Tal delegação de controle da memória ao sistema operacional implica no seguinte fato: qualquer demanda por memória será resolvida mediante **requisição formal** ao sistema operacional mantendo-o sempre no controle da memória principal. Uma analogia deste comportamento é o funcionamento de alguns espaços para estacionamento de carros: o cliente faz a solicitação formal por uma vaga à entrada do estabelecimento.

O funcionário na guarita (que trabalha como o gerenciador de memória) registra a entrada do cliente e marca como usada a vaga que ele preencherá. Quando ele volta e recolhe seu carro, espaço é liberado e o funcionário atualiza o status daquela vaga como livre. Se um cliente qualquer requer uma vaga não há risco de ser indicado a um lugar já ocupado.

Na memória secundária os dados são gerenciados através de blocos de bytes chamados **arquivos** que se encontram em **diretórios (pastas)** estrutura-

das numa grande árvore de diretórios. Essa metodologia é transparente aos usuários de computadores os quais contam com as possibilidades de adicionar e remover arquivos e pastas. E na memória principal, como ocorre o gerenciamento? Cada **processo** (programa) consiste num conjunto de instruções gravadas numa parte da memória que foi requisitada ao sistema. O sistema busca por essa memória e a marca como ocupada numa **tabela de controle**. Quando o processo se encerra (fim da execução do programa) seu espaço é liberado removendo-se a marca da tabela: esta área agora está apta ao uso por outros processos.

E quando cada processo requer por memória adicional? Neste caso instruções do próprio programa fazem a requisição de memória ao sistema operacional. Os programas devem dizer quanto precisam e o sistema responde marcando áreas livres de memória que assumirão e manterão o status de ocupadas enquanto estiverem sobre a tutela daquele processo que as requereu. E como os programas gerenciarão internamente cada bloco de memória requisitado? Para tanto existe um mecanismo denominado **vinculação**: ele consiste em ligar uma **entidade abstrata** interna definida no código do programa (pelo programador) a uma parte finita da memória. Tais entidades são denominadas **variáveis** do programa e a região a que são vinculadas de **células de memória**.

Um programa pode requerer quantas células de memória adicional precisar através do uso de variáveis. Cada variável conterá seu próprio **nome** e seu **escopo** (este conceito será abordado mais adiante). Devido ao caráter formal de requisição de memória as variáveis precisam ser referenciadas no programa antes de serem utilizadas. Esse referenciamento é denominado **declaração de variáveis**.

Variáveis podem ser **primitivas** ou **derivadas**. As variáveis primitivas tratam da representação de dados elementares como números e caracteres. Existem categorias distintas de variáveis primitivas as quais se diferenciam basicamente pelo **tipo de dados** que se propõem a armazenar e pela **faixa de representação** destes dados. Variáveis derivadas (como os vetores) são aquelas constituídas por mais de um elemento primitivo e serão estudadas mais adiante.

5.2. Os tipos de variáveis

São quatro os principais tipos primitivos de variáveis: **Inteiros**, **Reais**, **Caracteres** e **Lógicos**. A seguir fazemos a descrição de cada um.

a) Tipos numéricos: inteiros e reais

É muito comum em programação de computadores o uso de valores numéricos. Eles dividem-se basicamente em dois grandes grupos: o dos **inteiros** e

o dos reais. Freqüentemente há confusão entre o conceito de inteiros e reais em computação e em matemática. Nesta última tais definições correspondem a conjuntos com quantidade infinita de números além do fato de o conjunto dos inteiros ser contido no dos reais. Em computação valores inteiros e reais também representam conjuntos de números, mas neste caso de quantidade finita de elementos. Além do mais a idéia de inteiros contido nos reais não se aplica com exatidão neste caso. Analisaremos estes dois aspectos nos parágrafos seguintes.

Por que há finitos inteiros e reais nos tipos numéricos dos computadores? A resposta é simples: cada célula de memória possui uma quantidade finita de bits. Assim cada número em base decimal é representado em circuito por um layout de um grupamento de bits (base binária) previamente vinculado a uma variável.

De fato o que se faz é pré-estabelecer uma quantidade fixa de bytes para variáveis de um dado tipo e verificar que faixa de valores pode ser representada. Por exemplo, com 1-byte podem-se montar até no máximo 256 combinações de zeros e uns que poderão tanto representar números inteiros entre -128 e 127 (denominados inteiros de 1-byte **com sinal**) quanto de 0 a 255 (inteiros de 1-byte **sem sinal**). Para mais detalhes sobre bases numéricas e suas transformações o leitor pode consultar (FERNANDEZ; FEDELI, 2003).

b) Tipo caractere

A maior parte da informação em formato eletrônico contido nos computadores em todo mundo é textual, ou seja, é formada de **caracteres** das diversas linguagens naturais (português, inglês, francês, alemão etc.). É obviamente importante desenvolver métodos computacionais seguros que permitam o armazenamento e recuperação deste tipo de informação.

Como uma máquina intrinsecamente numérica pode armazenar texto? Na realidade qualquer informação registrada e recuperada por computadores será binária, e ainda, após um tratamento de conversão, por exemplo, binário para decimal, estas seqüências de bits se apresentarão como números inteiros. Então onde entram os caracteres? Estes são mantidos por tabelas existentes em várias categorias de softwares como editores de textos ou mesmo o próprio sistema operacional.

Nestas tabelas há duas colunas relacionadas: na primeira encontram-se números normalmente inteiros sem sinal; na segunda um conjunto de grafemas de uma ou mais línguas naturais: são os **caracteres**. Nestas tabelas existe um único inteiro para cada caractere e vice-versa.

As tabelas de caracteres podem ter tamanhos variados de acordo com a quantidade de bits dedicados a representação dos valores inteiros da pri-

meira coluna. Com 1-byte, por exemplo, tem-se 8-bits e logo números entre 0 e 255: assim com 1-byte 256 caracteres distintos podem ser representados.

Uma das mais comuns tabelas de caracteres utiliza 7-bits e denomina-se ASCII (*American Standard Code for Information Interchange*, que em português significa "Código Padrão Americano para o Intercâmbio de Informação"). Este padrão tem uso difundido em todo o mundo e é aceito praticamente por todos os softwares de edição de texto. Neste texto os algoritmos tomarão seus caracteres do padrão ASCII. A representação em pseudocódigo ora aparecerá como número (que neste caso está na faixa 0 a 27-1, ou seja, 0 a 127) ora aparecerá como o caractere propriamente dito (grafema). Neste último caso os grafemas correspondentes serão escritos sempre entre aspas simples (' '). A seguir ilustramos parcialmente a tabela ASCII fazendo pares do valor numérico com o caractere (entre aspas simples) equivalente:

32	' '	48	'0'	64	'@'	80	'P'	96	'`'	112	'p'
33	'!'	49	'1'	65	'A'	81	'Q'	97	'a'	113	'q'
34	'"'	50	'2'	66	'B'	82	'R'	98	'b'	114	'r'
35	'#'	51	'3'	67	'C'	83	'S'	99	'c'	115	's'
36	'$'	52	'4'	68	'D'	84	'T'	100	'd'	116	't'
37	'%'	53	'5'	69	'E'	85	'U'	101	'e'	117	'u'
38	'&'	54	'6'	70	'F'	86	'V'	102	'f'	118	'v'
39	'''	55	'7'	71	'G'	87	'W'	103	'g'	119	'w'
40	'('	56	'8'	72	'H'	88	'X'	104	'h'	120	'x'
41	')'	57	'9'	73	'I'	89	'Y'	105	'i'	121	'y'
42	'*'	58	':'	74	'J'	90	'Z'	106	'j'	122	'z'
43	'+'	59	';'	75	'K'	91	'['	107	'k'	123	'{'
44	','	60	'<'	76	'L'	92	'\'	108	'l'	124	'|'
45	'-'	61	'='	77	'M'	93	']'	109	'm'	125	'}'
46	'.'	62	'>'	78	'N'	94	'^'	110	'n'	126	'~'
47	'/'	63	'?'	79	'O'	95	'_'	111	'o'	127	'□'

Na prática muitos editores de texto simples salvam e recuperam seus arquivos de texto pela leitura byte a byte considerando cada um destes como um caractere ASCII: o oitavo bit obviamente não é representativo. Por englobar apenas 128 grafemas (27), ele é pouco representativo no que diz respeito às inúmeras línguas naturais e seus diversificados caracteres. A representação de letras acentuadas, por exemplo, em língua portuguesa não é suportada convenientemente em tal sistema. Por isso é muito comum encontrar fontes de dados em formato eletrônico (na internet, por exemplo) com erros de acentuação. Um sistema mais universal e mais complexo, o UNICODE, substitui progressivamente o ASCII na representação de caracteres.

E como funcionam os **tipos caracteres** em programação? Cada variável declarada como caractere num programa armazena de fato um número

inteiro. Na maioria das implementações de linguagens de programação, as células de memória vinculadas são de 1-byte (isso tem mudado para 2-bytes em linguagens mais atuais com o UNICODE). Então quando se recorrem às tabelas de caracteres? Isso acontece na **impressão**.

A impressão é o momento onde informação é enviada para um dispositivo (hardware) de saída como uma impressora, um disco ou a tela. Cada hardware reage de forma diferente ao recebimento de dados para impressão: a impressora deposita tinta em papel, no disco arquivos são criados ou têm alterados seus conteúdos; e na tela são impressos momentaneamente imagens para leitura. Em todas essas ações a informação textual recebe arte final pela recorrência às tabelas de caracteres.

Mais adiante serão estudadas as **cadeias de caracteres**: elas correspondem a junções (**concatenações**) de dois ou mais caracteres e são de fato as entidades representativas dos textos.

c) Tipo lógico

Uma variável é lógica quando ela possui apenas dois valores possíveis: genericamente **Verdadeiro** e **Falso**, ou ainda 0 e 1. Tais notações provêm da álgebra booleana e por essa razão este tipo de dados é comumente denominado também tipo booleano.

Em termos de memória 1-bit seria necessário para representar variáveis lógicas. Entretanto o que se vê na prática é o uso de 1-byte. Isso se deve ao modelo de endereçamento das máquinas modernas efetuado byte a byte (o endereçamento bit a bit seria custoso e ineficiente). O uso de um byte completo por booleanos é resolvido da seguinte forma: uma variável lógica será **Falsa** se todos os oito bits forem zeros, do contrário será **Verdadeira**.

Em algumas linguagens de programação há uma relação direta entre variáveis inteiras de 1-byte e variáveis lógicas. Se x é uma variável lógica, por exemplo, ao receber um valor inteiro, caso seja nulo (todos os bits iguais a 0) então a x será atribuído de fato o valor Falso; caso contrário a x será atribuído Verdadeiro. Analogicamente se y é inteiro e a y é atribuído Falso, tal variável receberá 0; do contrário se a y for atribuído Verdadeiro, tal variável receberá 1.

5.3. A declaração de variáveis

A primeira parte de um algoritmo computacional é normalmente a declarativa, ou seja, nela são definidas as variáveis que serão utilizadas no restante do código. Considere o exemplo a seguir:

```
declare
   i: inteiro
   x, y: real
   b: lógico
   c: caractere
//instruções do bloco de código
```

Este pseudocódigo ilustra como funciona a declaração de cinco variáveis: uma inteira, chamada **i**, duas reais, chamadas **x** e **y;** uma lógica, chamada **b**; e uma de tipo caractere chamada **c**. A sessão de declaração de variáveis possui a seguinte sintaxe fixa: a palavra chave **declare** marcando o início da sessão e logo abaixo a lista de variáveis indentada em um nível para representar o **bloco de variáveis**.

Cada linha desta lista é constituída por um subgrupo de uma ou mais variáveis de mesmo tipo; caso haja mais de uma variável por subgrupo, elas serão separadas por vírgulas (como ocorreu a **x** e **y**). Cada subgrupo de variáveis encerrará em dois pontos (:) seguido pelo tipo base do subgrupo. Os tipos primitivos base terão os nomes **inteiro**, **real**, **caractere** e **lógico** (cada qual referindo-se aos tipos já apresentados).

O nome das variáveis normalmente aparece como um único caractere: entretanto isso não é obrigatório. Nomes completos são legais e as regras para criá-los são as seguintes: não são usadas palavras reservadas (como **declare**, **pare**, etc.); não são usados caracteres especiais na grafia (como *, # ou &); não são usados espaços em branco em um mesmo nome; números são permitidos desde que não estejam no começo do nome (exemplo, x88 é válido, mas 5w não é); uso de underline (_) é legal. Exemplos:

```
declare
   alpha, beta: inteiro
   dia _ de _ sol: lógico
   x200, y _ 90, hx: real
   ch _ base _ 5: caractere
```

Após a sessão declarativa de variáveis segue o bloco principal do algoritmo. Variáveis declaradas e não utilizadas neste bloco deverão ser removidas.

5.4. Entrada e saída com variáveis

Depois de declaradas cada variável funcionará como uma "caixa" onde se podem colocar valores e de onde também se podem lê-los. É importante observar que tais caixas suportam um valor de cada vez e que desta forma a idéia de empilhamento de mais de um valor naquele mesmo espaço é inconcebível. Outra idéia inconcebível é a de "caixa vazia": sempre haverá uma informação disponível numa variável, mesmo logo após sua declaração. Normalmente esta informação não tem sentido algum no contexto da codificação sendo meramente projeção do conteúdo digital presente naquela célula de memória no momento da declaração.

A entrada de variáveis é feita com o comando **leia** ao passo que a saída pelo comando **escreva**. Vejamos o exemplo a seguir:

```
declare
   x: inteiro
leia x
escreva 'valor fornecido =', x
```

Neste exemplo **leia** provoca uma parada provisória da execução o que significa uma espera de entrada de informações via dispositivo padrão (teclado neste caso). Quando o usuário digita alguma coisa a execução ainda não prossegue porque uma confirmação é necessária (como a tecla ENTER). Uma vez confirmada, esta informação é transmitida e processada pela CPU, neste caso para se tornar um valor numérico inteiro. Novamente outra transmissão ocorre desta vez da CPU com destino final a memória principal, mais exatamente a célula vinculada à variável **x**. Até segunda ordem, ou até a finalização da execução, a variável **x** conterá o valor fornecido pelo usuário.

A próxima instrução executada é uma chamada a **escreva**. Ela não provoca espera, pelo contrário, envia à saída padrão (neste caso a tela) conteúdo textual. Neste exemplo a informação escrita tem duas partes separadas por uma vírgula: a primeira é uma mensagem de texto que será enviada para a tela tal como foi escrita; a segunda é uma referência a célula **x** e neste caso a CPU necessita recorrer à memória principal, buscar pelo conteúdo numérico em **x**, transformar este conteúdo em texto e por fim enviá-lo à saída padrão.

De fato apenas uma mensagem de texto é enviada à saída padrão: se duas ou mais partes separadas por vírgulas estão presentes o processamento em série individual será feito, várias componentes textuais geradas e por fim concatenadas numa única a qual será enviada. O termo **concatenação** surge

comumente em computação representando a fusão de informação numa só, normalmente textual.

As vírgulas possuem papel importante tanto em **leia** como em **escreva**. Para o comando **leia** a vírgula permite leitura de vários valores em uma única chamada do comando. Veja o exemplo a seguir:

```
declare
   x, y: inteiro
leia x, y
escreva 'A soma vale =', x+y
```

A linha de instrução contendo **leia** neste exemplo é exatamente equivalente a duas chamadas ao comando, uma para **x** e outra para **y**, em linhas distintas, como em:

```
leia x
leia y
```

Se na execução o usuário fornecer um valor e pressionar ENTER a execução não prosseguirá, pois **leia** aguarda por um segundo valor. Enquanto este não for digitado, confirmações com ENTER serão incapazes de permitir o prosseguimento. Em outras palavras: apesar de uma única chamada a **leia**, cada variável em sua lista (que são separadas por vírgulas) necessitará de sua própria confirmação. É possível digitar todos os valores pretendidos às variáveis e confirmar a entrada com um único ENTER? Sim. Neste caso todas as entradas devem ser digitadas numa única linha e separadas entre si por espaços. A seguir temos a saída obtida em tela para as duas abordagens descritas (<ENTER> apenas representa a tecla de confirmação sendo pressionada):

```
25 <ENTER>
32 <ENTER>
A soma vale = 57
```

Confirmações uma a uma

```
25 32 <ENTER>
A soma vale = 57
```

Confirmação única

A última linha impressa nos esquemas acima ilustra a flexibilidade do comando **escreva**. Ao passo que **leia** pode apenas manipular listas de variáveis, **escreva** pode manipular listas contendo mensagens, variáveis ou mesmo expressões. A expressão **x+y** do exemplo demanda da CPU processamento adicional que efetua a soma dos conteúdos das células **x** e **y** antes da

conversão em texto e da concatenação. Mais detalhes sobre as expressões serão vistos mais adiante.

O exemplo a seguir ilustra a leitura de um número e a exibição se seu quadrado:

```
declare
   n: inteiro
leia n
escreva n, ' ao quadrado vale ', n*n
```

Neste pseudocódigo o comando **escreva** concatena o valor de **n**, uma mensagem de texto e o valor da expressão, **n*n** (produto de n por n). Um exemplo de execução segue:

```
6 <ENTER>
6 ao quadrado vale 36
```

Caracteres podem ser lidos normalmente por **leia** e escritos normalmente por **escreva**. Entretanto alguns cuidados devem ser tomados. Se uma instrução como, **leia c**, com **c** caractere, é executada, muitos caracteres poderão ser escritos pelo usuário antes da confirmação: neste caso **c** só receberá o primeiro caractere e os demais serão desconsiderados. Se uma instrução como, **escreva c**, com **c** caractere é executada, será impresso um caractere contido em c, mas sem as aspas simples.

Valores lógicos não podem ser lidos ou escritos diretamente com **leia** e **escreva**. Mais adiante analisaremos exemplos que, entretanto, permitem aos usuários modificarem ou inspecionarem variáveis deste tipo.

Se o comando **leia** aguarda por valores numéricos e, por exemplo, texto é digitado, então uma conversão apropriada não poderá ser feita pela CPU e ocorrerá um **erro em tempo de execução**. Nesta abordagem sobre algoritmos, quando um erro ocorre necessariamente a execução é suspensa.

Saiba Mais

O link do light-bot traz um jogo muito interessante para o início da aprendizagem em programação. O nome do jogo é Light-bot o qual consiste de um robô capaz de executar tarefas (sequência de passos) com o objetivo de chegar ao quadrado azul e acendê-lo. Ao carregar a página do jogo, você deve clicar em play e logo em seguida clicar em *new game* para iniciar. Vamos lá ! Você verá que a prática aliada aos algoritmos que você desenvolverá para chegar ao quadrado azul o ajudarão na aprendizagem de programação.

Síntese do capítulo

Neste capítulo foi apresentado o conceito de algoritmo e suas formas básicas de representação: texto, fluxograma e pseudocódigo. Mostra também as possibilidades de execução que um algoritmo pode ter. Lembrando que as mesmas devem seguir uma sequência, mas que nem sempre todas podem ser processadas. Outros fatores importantes para algoritmos são: entrada e saída. O conceito de variável e os tipos podem ser: inteiro, real, caractere e booleano. A primeira parte do programa é a declarativa, a qual define os tipos de variáveis também contemplaram este capítulo.

Atividades de avaliação

1. Definir algoritmo utilizando suas próprias palavras.
2. Formule em forma de fluxograma três algoritmos que descrevem tarefas de seu dia-a-dia.
3. Discutir sintaxe e semântica em uma metodologia de construção de algoritmos.
4. O que são pseudocódigos e que aspectos de sintaxe estão envolvidos?
5. Qual a importância da indentação na construção de pseudocódigos?
6. Outras metodologias de construção de algoritmos utilizam blocos do tipo início-fim para promover o encapsulamento de código. Pesquise e responda por que a indentação neste caso não é necessária e qual o papel dela quando utilizada.
7. Pesquise e responda: qual a diferença entre uma instrução de código e uma instrução de máquina?
8. Conceituar entrada e saída padrão e discutir as questões de simplificação de hardware associadas a elas.
9. Quais as principais diferenças entre o gerenciamento de memória secundária e primária?
10. Que são variáveis e qual a relação entre elas, a memória principal e o sistema operacional?
11. Quais são e o que representam os tipos primitivos de variáveis?
12. Construir pequeno algoritmo que leia três valores da entrada padrão reais e escreva na saída padrão a média aritmética entre eles.

Leituras, filmes e sites

http://equipe.nce.ufrj.br/adriano/c/apostila/algoritmos.htm

http://armorgames.com/play/2205/light-bot

Referências

SEBESTA, Robert W. **Concepts of Languagues Programming.** 8th edition. University of Colorado. Addison-Wesley 2007. 752p. ISBN-10: 032149362.

FERNANDEZ, M; Cortés, M. **Introdução à Computação.** 1ª Edição. Fortaleza. RDS Editora, 2009.

Capítulo 2

Elementos construtivos

Objetivos

- Apresentar os elementos essenciais para a construção de algoritmos.

1. Operadores

Variáveis são vinculações com a memória real do computador e funcionam como meros armazenadores temporários de informação digital. Entretanto, para a construção de algoritmos funcionais, deve ser possível também operar entre si tais variáveis. Operar neste contexto pode significar tanto alterar o conteúdo de uma variável quanto associar duas ou mais destas numa expressão válida para se obter um novo valor (por exemplo, **x+y** envolve duas variáveis numa operação de soma cujo resultado é um novo valor).

Operadores são objetos de construção de algoritmos, de simbologia própria e semântica bem definida, destinados às operações com ou entre variáveis. Por exemplo, na expressão, **a+b**, o símbolo **+** é um operador e sua semântica é a execução da soma entre os valores nas variáveis **a** e **b**.

Operadores podem ser **unários** ou **binários**, ou seja, podem ou operar um ou dois operandos respectivamente. O exemplo mais comum de operador unário é o de mudança de sinal. A sintaxe é **-x**, onde o sinal de menos é o operador, x é a variável e o valor retornado é o valor na variável (numérica) com sinal invertido.

Outro exemplo de operador unário é valor absoluto. A sintaxe é **|x|**, onde as duas barras verticais representam o operador, **x** é a variável e o valor retornado é o módulo do valor na variável (numérica). Operadores aritméticos são exemplos de operadores binários. Na expressão, **a*b**, o asterisco denota um operador binário operando as variáveis **a** e **b** e cuja semântica é a multiplicação numérica. Nesta expressão o valor retornando é o produto entre o conteúdo existente nessas variáveis.

Operadores podem ser **infixos**, **préfixos** ou **pósfixos**. Esta classificação refere-se à posição relativa dos operadores nas expressões em que aparecem. Por exemplo, em, **a+b**, o operador de adição, **+**, é infixo, ou seja, é disposto entre seus operandos. Na expressão de mudança de sinal, **-a**, o operador é pré-fixo, ou seja, disposto antes do operando. Na expressão de cálculo de um fatorial, **x!**, o operador é pós-fixo, ou seja, é disposto depois do operando. Usualmente a maioria dos operadores binários é usada infixamente e dos unários pré-fixamente.

Há situações em que operadores tradicionalmente infixos são utilizados como pré ou pós-fixação. Em notação polonesa[2] inversa (PEREIRA; 2006), por exemplo, a expressão, **AB+**, é legal e denota a soma das variáveis A e B com operador de adição pós-fixado. Esta notação é conveniente aos computadores no que diz respeito a avaliação de expressões (úteis no projeto de compiladores, por exemplo).

Operadores são funções, ou seja, ao invés da notação apresentada até então uma dada operação pode ser expressa utilizando uma função nomeada com seus operandos entre parênteses e separados por vírgula. Assim, por exemplo, a expressão, **SOMA(A, B)**, tem exatamente o mesmo efeito de, **A+B**. Na notação funcional o símbolo + é substituído pela função **SOMA()**.

Em outras sintaxes propostas para algoritmos, ou mesmo em linguagens de programação, operadores são introduzidos apenas pela notação funcional. Se um operador não possui símbolo que permita prefixação, in-fixação ou pós-fixação, então a notação funcional é única opção. Exemplos: logaritmo natural **LN()**, função exponencial **EXP()**, funções trigonométricas **(SEN()**, **COS()** e etc.) entre outras.

[2] Notação Polonesa Inversa, também conhecida como **notação pós-fixada**, foi inventada pelo filósofo e cientista da computação australiano Charles Hamblin em meados dos anos 1950. Ela deriva da notação polonesa cujos operadores são prefixos dos operandos. Como exemplo a expressão, A+B, é escrita como +AB. A notação polonesa foi introduzida em 1920 pelo matemático polonês Jan Łukasiewicz. (Daí o nome sugerido de notação Zciweisakul).

1.1. A atribuição

O **operador de atribuição** é o elemento de construção de algoritmos que expressa mais genuinamente a essência da máquina de von Neuman[3] (FERNANDEZ; 2009). Este operador é o responsável pela modificação do conteúdo das células de memória (variáveis) alocadas. Cada **atribuição** corresponde a uma ação da CPU que envia dados de seus registradores para uma célula de memória alvo e cujo valor só se modificará novamente por outra atribuição.

Atribuições podem ser **explícitas** ou **implícitas**. Numa atribuição explícita o operador ← é utilizado para indicar que o lado direito da expressão é alvo do lado esquerdo, ou seja, uma variável do lado esquerdo terá seu valor modificado para o valor avaliado no lado direito. Veja o exemplo a seguir.

[3] A Arquitetura de von Neuman (de John von Neuman), é uma arquitetura de computador que se caracteriza pela possibilidade de uma máquina digital armazenar seus programas no mesmo espaço de memória que os dados, podendo assim manipular tais programas. A máquina proposta por Von Neumann reúne os seguintes componentes: (i) uma memória, (ii) uma unidade aritmética e lógica (ULA), (iii) uma unidade central de processamento (UCP), composta por diversos registradores, e (iv) uma Unidade de Controle (UC).

```
declare
   a: inteiro
   x, y: real
   c: caractere
a←120
x←3.253
y←a+x+23
c←'w'
escreva b, y, c
```

No pseudocódigo acima às variáveis **a** e **x** são atribuídos valores numéricos diretamente; à variável **y** é atribuído o resultado da avaliação prévia da expressão, **a+x+23** (a soma é calculada primeiramente para então **y** ser modificado); à variável c é diretamente atribuído o código ASCII de caractere `'w'`; por fim na última instrução, escreva converte para texto, concatena e imprime **b**, **y** e **c.**

Se uma variável é lógica, a atribuição explicita em código dos valores Verdadeiro e Falso é feita mediante as palavras chave **V** e **F** respectivamente. Exemplo:

```
declare
   a, b: lógico
a←V
b←F
```

De uma forma geral o lado direito de uma atribuição é primeiramente resolvido para então o resultado encontrado ser redirecionado para à célula de memória vinculada a variável do lado esquerdo desta atribuição. Esta forma de processamento permite que a seguinte expressão seja legal:

```
x←x+y
```

Em casos como este a variável que aparece em ambos os lados da atribuição possui um valor antes e outro depois da atribuição e comumente chamados de **valor antigo** e **valor novo** da variável respectivamente.

O processamento ocorre da seguinte forma: o conteúdo das células **x** e **y** são copiados para registradores distintos da CPU; em seguida a unidade lógica aritmética desta efetua a soma e a coloca num terceiro registrador; por fim o valor da soma é transmitido para a célula de memória vinculada a **x** onde ocorrerá sobreposição de valor. Este último passo corresponde a atribuição propriamente dita.

Atribuições implícitas ocorrem quando o comando **leia** é invocado. O mesmo mecanismo de envio de uma informação para uma célula de memória alvo acontece, porém, a origem da informação é a entrada padrão. Veja o exemplo:

```
declare
   x, y: inteiro
leia x
y←x+10
escreva 'Resposta: ', y
```

Neste pseudocódigo a variável **x** tem atribuição implícita ao passo que **y** tem atribuição explícita, ou seja, o conteúdo na célula vinculada a **x** muda por ação de leia ao passo que aquela vinculada a **y** muda por ação do operador ← após resolução da expressão, **x+10**. Funcionalmente a execução deste algoritmo implica na leitura de um valor dado pelo usuário e sua impressão acrescentada de 10. Veja o esquema de execução a seguir:

```
34 <ENTER>
Resposta: 44
```

Em alguns casos o operador ← pode participar diretamente na entrada de dados fornecidos pelo usuário. É o que acontece com o comando **tecla**. Quando uma expressão de atribuição contendo **tecla** é encontrada o processamento é suspenso provisoriamente aguardando entrada do usuário, da mesma forma que acontece com **leia**. Entretanto neste caso basta um único pressionar de tecla para continuar o processamento. A tecla pressionada neste intere é atribuída a variável do lado esquerdo da atribuição. Por esta razão tal variável deve ser de tipo caractere. Veja o exemplo a seguir:

```
declare
   ch: caractere
ch ← tecla
escreva 'tecla pressionada: ', ch
```

Neste pseudocódigo a execução da linha, **ch** ← **tecla**, implica suspensão provisória da execução. Quando o usuário pressionar uma tecla seu valor ASCII é repassado para a variável **ch** (caractere) a qual manterá este valor até uma nova atribuição. A mensagem final impressa é a concatenação da mensagem entre aspas e o grafema do caractere da tecla pressionada.

1.2. Operadores aritméticos

É conveniente padronizar o comportamento dos operadores aritméticos em qualquer estudo de algoritmos. Na tabela a seguir são apresentados os símbolos e a descrição adotados neste texto.

Operadores	Descrição
+	Soma de valores numéricos
–	Subtração de valores numéricos
*	Multiplicação de valores Numéricos
/	Divisão entre valores reais
div	Divisão entre valores inteiros
mod	Resto de divisão entre valores inteiros
abs	Módulo de valor numérico

Todos os operadores, com exceção de **abs**, são usualmente binários e infixos O operador **abs** é unário e com notação funcional. Por exemplo, a expressão, **abs(x)**, representa o módulo do valor em **x**. Os operadores, **+**, e, **–**, possuem também versões unárias e pré-fixas como nas expressões, **+x-3** (ênfase de sinal), e, **–a+b** (inversão de sinal).

O operador / é utilizado para dividir dois valores reais retornando um valor real de saída. Se algum dos operandos for inteiro então este será primeiramente convertido em real antes da divisão. Na expressão, **a**←**b/c**, se **b** e/ou **c** são inteiros, então eles são primeiramente convertidos em reais, depois divididos e o resultado da divisão atribuído a variável **a**. Se **a** é uma variável real, o valor integral da divisão é atribuído, por exemplo, se **b** contém 7 e **c** contém 2, então **a** receberá 3.5. Entretanto, se **a** é uma variável inteira ocorrerá **truncamento**, ou seja, o real oriundo da avaliação da expressão do lado direito da atribuição perderá a parte fracionária e o valor inteiro restante é atribuído à a. Neste caso o valor 3.5 é truncado para 3 para então haver atribuição.

As expressões, a←b **div** c, e, a←b **mod** c, só serão legais se **b** e **c** forem variáveis inteiras. Se **a** for real, o inteiro oriundo da operação no lado direito será convertido em real para então ocorrer a atribuição.

1.3. Operadores relacionais

Operadores relacionais são operadores binários e infixos cujo resultado da expressão que constituem é sempre um valor lógico, ou seja, Verdadeiro ou Falso. A tabela a seguir lista símbolos e nomes dos operadores relacionais utilizados neste estudo de algoritmos:

Operadores	Nomes
=	Igual a
>	Maior que
<	Menor que
≥	Maior ou igual a
≤	Menor ou igual a
≠	Diferente de

A seguir alguns exemplos de expressões relacionais legais, ou seja, cada uma delas quando avaliada será Verdadeira ou Falsa.

a > 100
400 ≤ y
a+b ≠ c
x+y < a+b

Nos dois últimos casos acima o operador relacional contém em um dos lados (ou em ambos) expressões aritméticas. Esta sintaxe é perfeitamente legal e será analisada logo adiante.

1.4. Operadores lógicos

Operadores lógicos são aqueles que operam valores lógicos e retornam valores lógicos. Utilizando a notação mais usual da álgebra booleana[4], montamos a listagem dos operadores lógicos, juntamente com seus nomes conforme tabela a seguir.

Operadores	Nomes
∧	Conjunção
∨	Disjunção
⊕	Disjunção Exclusiva
¬	Negação

Os operadores de conjunção, disjunção e disjunção exclusiva são todos binários e infixos ao passo que o de negação é unário e pré-fixo. A seguir algumas expressões lógicas legais:

```
X ∨ Y
B ⊕ (A ∧ B)
(A ∨ B ∧ C) ∧ (¬A ⊕ C)
```

[4] Álgebra booleana (ou Álgebra de Boole) trata das estruturas algébricas construídas a partir das operações lógicas E, OU e NÃO, bem como das operações soma, produto e complemento em teoria de conjuntos. É comumente aplicada em computação para operar números binários. Recebeu o nome de George Boole, matemático inglês, que foi o primeiro a defini-la como parte de um sistema de lógica em meados do século XIX.

Nos exemplos acima são utilizados apenas variáveis e operadores lógicos. As variáveis poderiam ser substituídas, por exemplo, por expressões relacionais. O resultado da avaliação destas expressões é também um valor lógico.

Como cada variável do tipo lógico só assume dois valores, Falso ou Verdadeiro (**F** ou **V**) é possível listar todas as possibilidades de resposta de uma expressão lógica (em contrapartida às expressões numéricas com inúmeras possibilidades). Tais listagens são conhecidas como **Tabelas-Verdade**. A seguir são listadas as tabelas verdades para as expressões lógicas elementares (que possuem apenas um operador).

A	B	A∧B	A∨B	A⊕B
F	F	F	F	F
F	V	F	V	V
V	F	F	V	V
V	V	V	V	F

2. Expressões

Expressões podem ser constituídas pela associação de diversas variáveis de tipos não necessariamente compatíveis, mas que se arranjem de forma lógica e avaliável. Uma vez definidas tais expressões, a avaliação implica num processo sistemático que finaliza num valor final cujo tipo dependerá da expressão.

Por exemplo, na expressão, **x+1 > y-p*p**, resolvem-se primeiramente as expressões aritméticas em cada lado do operador relacional para depois fazer a comparação que resultará num valor lógico. Este procedimento de escolha de "quem operar primeiro" numa expressão segue na verdade dois conjuntos de regras de operadores. São eles o de **precedência de operadores** e de **associatividade de operadores**. Os analisaremos a seguir.

2.1. Precedência de operadores

Nas primeiras lições sobre matemática ensina-se que numa expressão numérica primeiramente devem-se fazer as divisões e multiplicações para por fim fazer as adições e subtrações. Ensina-se também que se existem componentes entre parênteses elas são prioritárias na ordem de resolução. Sem tais regras uma expressão como, **2+4*7**, seria ambígua, ou seja, com duas possibilidades de resposta dependo de qual operação fosse realizada em primeiro lugar. Esta predefinição de quem opera primeiramente numa família de operadores é comumente chamada de **precedência de operadores**.

A ordem convencional de precedência dos operadores aritméticos, do de maior para o de menor, é a seguinte:

```
(maior precedência)      abs
                         +, - (unários)
                         *, /, mod, div
(menor precedência)      +, - (binários)
```

A precedência convencional dos operadores lógicos é a seguinte:

```
(maior precedência)      ¬
                         ∧
                         ∨
(menor precedência)      ⊕
```

Por exemplo, na expressão, **A** $\oplus$ **B** $\wedge \neg$ **C**, o booleano **C** é primeiro invertido, depois a conjunção entre este valor e **B** é feita; por fim a disjunção exclusiva fecha o processo de avaliação e um valor final lógico é obtido.

Convencionalmente não existe precedência entre operadores relacionais, o que será mais bem compreendido na sessão seguinte.

2.2. Associatividade de operadores

Permitir a **associatividade de operadores** significa permitir que dois operadores de uma mesma família (aritmético, relacional ou lógico) e de mesma precedência apareçam adjacentes em uma mesma expressão sem afetar operabilidade. Por exemplo, na expressão, **A + B - C**, quem deve ser feita primeira? Adição ou subtração? A forma usual de resolver a associatividade é definindo um sentido fixo de avaliação; por exemplo, da direita para a esquerda (no exemplo anterior a subtração seria então feita primeiro).

A associatividade nem sempre é possível. Para que dois operadores possam ser associados o tipo da saída do operador deve ser do mesmo tipo dos operandos, do contrário ocorrerá inconsistência. Por exemplo, a avaliação da expressão, **A + B - C**, da esquerda para a direita, possui o seguinte esquema:

```
A+B-C
<número>+<número>-<número>
<número>+<número>
<número>
```

Cada redução na resolução acima transforma um par de números num novo número que entra na etapa seguinte da avaliação. De uma forma geral, operadores aritméticos são sempre associáveis. O mesmo não acontece na expressão, **A > B > C**, onde **A**, **B** e **C** são inteiros. O processo de avaliação desta expressão, da esquerda para direita, é interrompido por uma inconsistência, como se vê a seguir:

```
A>B>C
<número> > <número> > <número>
<número> > <lógico>   //inconsistência! Não pode continuar
```

Este exemplo demonstra a ausência de associatividade entre operadores relacionais e conseqüentemente a inexistência de precedência entre eles. A associatividade entre operadores lógicos funciona similarmente aos operadores aritméticos.

2.3. Expressões booleanas

Expressões booleanas são expressões cujo valor da avaliação é um lógico, ou seja, Verdadeiro ou Falso. Expressões booleanas podem conter componentes aritméticas, relacionais, ou mesmo lógicas. A seqüência de avaliação por ordem de natureza de componentes é a seguinte:

(1). Componentes aritméticas

(2). Componentes relacionais

(3). Componentes lógicas

Se existem parênteses nas expressões então o conteúdo entre eles deve ser avaliado primeiro obedecendo naturalmente, neste escopo, a seqüência anterior. Considere o exemplo a seguir:

```
x+y>50 ⊕ w<100+r ∧ w≠y-5
```

A seqüência esquemática de avaliação desta expressão é a seguinte:

```
x+y > 50 ⊕ w < 100+r ∧ w ≠ y-5
<número> > <número> ⊕ <número> < <número> ∧ <número> ≠
<número>
<lógico> ⊕ <lógico> ∧ <lógico>
<lógico> ⊕ <lógico>
<lógico>
```

Caso a disjunção exclusiva deva ser avaliada antes da conjunção, então o uso apropriado de parênteses transforma a expressão em:

```
(x+y>50 ⊕ w<100+r) ∧ w≠y-5
```

Cujo esquema de avaliação agora é:

```
(x+y > 50 ⊕ w < 100+r) ∧ w ≠ y-5
(<número> > <número> ⊕ <número> < <número>) ∧ <número> ≠ <número>
(<lógico> ⊕ <lógico>) ∧ <lógico>   //prioridade de parênteses
<lógico> ∧ <lógico>
<lógico>
```

3. Estruturas de controle

Existem dois mecanismos construtivos que qualquer método de elaboração de algoritmos deve possuir: **decisão** e **repetição**. O conjunto de componentes de código que se associam para representar um processo de decisão ou de repetição é denominado **estrutura de controle**. Sem tais estruturas não seria possível a construção de algoritmos.

3.1. Decisão

Um processo de decisão consiste na escolha entre dois ou mais blocos de pseudocódigo para execução. Apenas um, entre os blocos existentes, é selecionado e assim tem seu conteúdo executado.

Uma decisão é o resultado da avaliação de uma expressão booleana. Adiante são estudados diferentes tipos de estruturas de decisão.

a) Decisão unidirecional

A **decisão unidirecional** é aquela que envolve apenas um bloco de código. Se a expressão de controle da decisão é avaliada como verdadeira, então o código do único bloco é executado. Do contrário, se a expressão resulta em falso, ocorre um **desvio** para a próxima instrução fora do bloco (caso exista).

A sintaxe básica da decisão unidirecional é:

```
se (...) então
   ...
```

Onde a primeira reticência entre parênteses refere-se à expressão booleana de controle, enquanto a segunda refere-se ao bloco de pseudocódigo. Para marcar o bloco utiliza-se pelo menos um nível de indentação no código de execução condicional. Diz-se que instruções de um bloco de decisão são **encapsuladas** pela estrutura de decisão. Veja o exemplo:

```
declare
   x: inteiro
leia x
se (x>0) então
   escreva x
   escreva 'o valor é positivo!'
```

As últimas três linhas deste algoritmo constituem um processo de decisão. A linha com se mantém a expressão booleana de controle, **x>0**. Logo abaixo, indentadas em um nível para formar um bloco, duas instruções de impressão são encapsuladas pela estrutura. A execução do algoritmo inicialmente requer um valor inteiro através da variável **x**. Se este valor é positivo as duas impressões são executadas em seqüência (primeiro a do valor em **x** e depois a mensagem de texto). Se o valor em **x** é negativo ou nulo nada é impresso na saída padrão e a execução se encerra (não há mais instruções após processo de decisão).

Outro exemplo:

```
declare
   a: inteiro
leia a
se (a mod 2 = 0) então
   escreva 'o número é par'
        escreva 'fim da execução!'
```

Neste algoritmo o único processo de decisão presente contém duas linhas (em negrito) e o bloco encapsulado apenas uma linha (ver a indentação). Se a expressão booleana de controle, a **mod 2=0** (resto da divisão do valor em **a** por 2 é igual a zero?), é avaliada como verdadeira, então a mensagem 'o número é par' é impressa. Do contrário ela não o será. A impressão da mensagem 'fim da execução!' não pertence ao processo de decisão (ver indentação) e logo ocorre incondicionalmente.

b) Decisão bidirecional

Na decisão **bidirecional** dois blocos de pseudocódigo competem para que um deles, e apenas um, seja executado. Caso a expressão booleana de controle seja avaliada como Verdadeira, então o primeiro bloco de pseudocódigo será executado, do contrário o segundo bloco que o será.

A sintaxe básica da decisão bidirecional é:

```
se (...) então
   ...
senão
   ...
```

Onde a primeira reticência entre parênteses refere-se à expressão booleana de controle, a segunda refere-se ao primeiro bloco de pseudocódigo e a terceira reticência refere-se ao segundo bloco de pseudocódigo. Veja o exemplo:

```
declare
   num, x: inteiro
leia num
se (num mod 2 = 0) então
   x ← num div 2
   escreva x
senão
   x ← (num+1) div 2
   escreva x + 1
```

As linhas com as palavras-chave **se** e **senão** são colocadas no mesmo nível de indentação para acusar que fazem parte da mesma estrutura. Os blocos encapsulados são dispostos um nível de indentação acima estando o primeiro antes e o segundo após o **senão**. Se a expressão booleana de controle, num **mod 2=0**, é avaliada como verdadeira então **x** recebe o resultado na expressão, num **div 2**, e tem em seguida o valor impresso. Do contrário **x** recebe o resultado da expressão, **(num+1) div 2**, e o valor, **x+1**, é que é impresso. Apenas uma atribuição explícita e uma impressão ocorrem.

Outro exemplo:

```
declare
   n: inteiro
leia n
se (n mod 2 = 0) então
   escreva 'número par!'
senão
   escreva 'número ímpar!'
```

Cada um dos blocos no algoritmo anterior é constituído por uma única linha de instrução.

Estudo de caso – Existência de Triângulo: Consideremos agora o algoritmo que determina se um triângulo existe ou não. Há duas maneiras de abordar o problema: (I) pode-se testar se cada um dos lados é menor que a soma dos outros dois e (II) pode-se testar se um lado qualquer é menor que soma dos outros dois e maior que o módulo da subtração entre os mesmos. O algoritmo para (I) será:

```
declare
   a, b, c: inteiro
leia a, b, c
se (a<b+c ∧ b<a+c ∧ c<a+b) então
   escreva 'O triâmgulo existe!'
senão
   escreva 'O triângulo não existe!'
```

A única diferença para o caso (II) está na expressão booleana de decisão que deve se tornar:

```
a<b+c ∧ a>abs(b-c)
```

O caso (I) ilustra o uso das expressões booleanas e a importância dos operadores lógicos quando expressões relacionais precisam ser conectadas. O caso (II) reafirma o caso (I) e ilustra o uso do operador módulo, **abs**, em componentes aritméticas de expressões booleanas.

c) Decisão múltipla

Em algoritmos, a **decisão múltipla** é um processo de decisão que envolve muitas entradas, ou seja, entre várias rotas possíveis de código será escolhida uma, e apenas uma, para ser executada. Devido à natureza binária das expressões booleanas, elas não se aplicam em processos da decisão múltipla. Ao invés disso é utilizada uma expressão aritmética com valor resultante inteiro (usualmente expressões de controle aparecem como uma única variável de tipo inteiro conhecida como **variável de controle**).

Espera-se naturalmente da expressão de controle que o valor proveniente da avaliação esteja numa faixa finita de valores inteiros. Cada um destes valores deve conduzir ao seu próprio bloco de código. Se a avaliação resulta um valor fora da faixa então nenhum bloco é executado e o processo de decisão encerrado. Blocos em decisões múltiplas precisam de um **rótulo** para indicar onde principiam. O rótulo deve conter o valor inteiro esperado pela avaliação da expressão de controle que induza a execução do bloco que rotula. A estrutura de decisão múltipla, conhecida como **caso...seja...**, possui a seguinte sintaxe geral:

```
caso <varável ou expressão de controle> seja
   <rótulo 1>: <bloco 1>
   <rótulo 2>: <bloco 2>
   ...
   <rótulo n>: <bloco n>
```

Cada rótulo da estrutura acima possui o próprio bloco e é separado dele pela marca, : (dois pontos). Todos os rótulos são dispostos um nível de indentação acima no nível onde esta a palavra chave **seja**. Blocos dispõem-se um nível de indentação acima daquele onde estão os rótulos.

Em síntese, após a avaliação da expressão (ou variável) de controle um valor inteiro é obtido, o rótulo com este valor selecionado e por fim o bloco executado. Quando a execução encerra, ocorre um **desvio** para fora da estrutura para que outros blocos não sejam executados.

O exemplo a seguir ilustra a estrutura **caso...seja**:

```
declare
   n: inteiro
leia n
caso (n) seja
   1: escreva 'Olá mundo!'
   2: escreva 'Hello world!'
   3: escreva 'Hallo Welt!'
```

A execução deste pseudocódigo implica impressão de uma entre três mensagens (caso o valor em **n** seja 1, 2 ou 3) ou em finalização sem impressão alguma (caso outros valores sejam fornecidos). Nunca duas mensagens são simultaneamente impressas.

A estrutura **caso..seja** pode ser estendida na sintaxe seguinte:

```
caso <varável ou expressão de controle> seja
   <rótulo 1>: <bloco 1>
   <rótulo 2>: <bloco 2>
   ...
   <rótulo n>: <bloco n>
   senão       <bloco n+1>
```

A palavra chave **senão** funciona neste contexto como um rótulo alternativo a todos os valores não rotulados anteriormente. Ou seja, se a avaliação da variável ou expressão de controle resulta um valor fora da faixa dos rótulos, então o bloco n+1 será executado. Neste contexto sempre algum bloco é executado. Uma prática bastante comum é o uso da sessão **senão** para imprimir mensagens de erro. Veja o exemplo:

```
declare
   n: inteiro
leia n
caso (n) seja
   1: escreva 'Olá mundo!'
   2: escreva 'Hello world!'
   3: escreva 'Hallo Welt!'
   senão
      escreva 'Opção inválida!!'
```

Se for fornecido a variável **n** um valor diferente de 1, 2 e 3 então a mensagem de erro 'Opção inválida!!' será impressa. O exemplo a seguir contém blocos, numa estrutura **caso..seja**, com mais de uma linha de instrução:

```
declare
   ch: caractere
   raio, base, altura, lado: inteiro
escreva 'pressione: C para cículo, T para triângulo, Q para
quadrado'
ch ← tecla
caso ch seja
   'C': escreva 'Valor do raio:'
        leia raio
        escreva 'área = ', 3.14159235*raio*raio
   'T': escreva 'Valores da base e da altura: '
        leia base, altura
        escreva 'área = ', base*altura/2
   'Q': escreva 'Valor do lado: '
        leia lado
        escreva 'área = ', lado*lado
   senão
        escreva 'Opção inválida!!'
```

Este algoritmo efetua cálculo da área de três categorias de figuras geométricas: círculo, triângulo ou quadrado. De acordo com a tecla pressionada a execução é direcionada para um rótulo apropriado. Cada bloco possui mensagem de entrada, leitura de variáveis e apresentação de resultados (área da figura). A variável de controle é de tipo caractere e como caracteres são tipos especiais de inteiros então podem ser usados em estruturas **caso...seja**.

O uso de uma decisão com **se..então..senão** em um bloco de **caso...seja** é perfeitamente legal e bastante usual. No exemplo seguinte simula-se uma calculadora cujos operandos são fornecidos primeiro e a operação (soma, subtração, multiplicação ou divisão) depois com um pressionar de tecla:

```
declare
   op: caractere
   x, y: inteiro
escreva 'fornecer valores: '
leia x, y
escreva 'forcecer opção : + - * /'
op ← tecla
caso op seja
   '+': escreva 'Soma: ', x+y
   '-': escreva 'Subtração: ', x-y
   '*': escreva 'Multiplicação: ', x*y
   '/': se (b≠0) então
           escreva x/y
        senão
           escreva 'divisão por zero!'
   senão escreva 'Opção inválida!!'
```

Se a tecla com o caractere '/' for pressionada então a operação de divisão é selecionada. Isso inicia um processo de decisão com **se..então..então** que trata a possibilidade de divisão por zero. A mensagem 'divisão por zero!' é impressa quando o valor em **b** é nulo. A mensagem 'Opção inválida!!' é impressa quando uma tecla operacional inválida é pressionada.

3.2. Aninhamento de decisões

A estrutura **se...então(...senão)** permite o **aninhamento**, ou seja, a possibilidade de construção de outras estruturas **se...então(...senão)** dentro de seus blocos. Este encapsulamento de decisões pode se estender em muitos níveis dependendo das unicamente das necessidades de expressão do algoritmo. A seguir um exemplo esquemático:

```
se (...) então
   // bloco
   se (...) então
      // bloco
senão
   se (...) então
      // bloco
   senão
      se (...) então
         // bloco
      senão
         // bloco
```

Estudo de Caso – IMC: A sigla IMC significa índice de massa corpórea. Trata-se de um número calculado pela razão entre o peso em kg (na verdade a massa!) e o quadrado da altura (em metros) de uma pessoa. O IMC determina se esta pessoa está em um dos estados seguintes: magra (o IMC é menor que 20), gorda (o IMC é maior que 25) ou em bom peso (o IMC está no intervalo fechado de 20 a 25). O pseudocódigo seguinte resolve o problema do IMC:

```
declare
   peso, altura, imc: real
escreva 'Fornecer peso e altura: '
leia peso, altura
imc ← peso /(altura*altura)
se (imc<20) então
   escreva 'Você está magro!'
senão
   se (imc>25) então
       escreva 'Você está gordo!'
   senão
       escreva 'Você está num bom peso!'
```

A execução deste algoritmo ocorre da seguinte forma: o primeiro **se** divide seu caminho entre os valores de IMC que são menores que 20 e os que não são. Caso seja menor que 20 a mensagem, '*Você está magro*!', é impressa e a execução finaliza. Caso seja maior ou igual a 20 um novo processo decisório aninhado é iniciado com se testando se o IMC é maior que 25. Caso seja Verdadeira a mensagem, '*Você está gordo!*', será impressa, do contrário

o conteúdo encapsulado pelo último **senão** será executado. Neste caso o valor de IMC certamente é maior ou igual a 20, por causa do primeiro **se**, mas é menor ou igual a 25, por causa do segundo se. Assim a impressão da mensagem, 'Você está num bom peso', sempre ocorrerá no intervalo fechado de IMC entre 20 e 25 (como esperado) dispensando o uso de outro se para o último bloco.

O aninhamento de decisões permite a programação da decisão múltipla de forma diferenciada daquela montada com **caso..seja**. Nesta estrutura as tomadas de decisão são necessariamente pontuais, isto é, um valor específico para cada rótulo de entrada. Com o aninhamento cada decisão fica atrelada a um intervalo de valores. Estes intervalos compõem o domínio de todas as entradas possíveis. Assim, por exemplo, com o aninhamento é possível construir decisão múltipla utilizando valores reais como no caso do IMC.

Estudo de Caso – Menor de Três Números: Consideremos o problema de determinação do menor de três números. O algoritmo seguinte efetua esta tarefa para três inteiros fornecidos via entrada padrão:

```
declare
   A, B, C, menor: inteiro
leia A, B, C
se (A<B ∧ A<C) então
   menor ← A
senão
   se (B<C) então
      menor ← B
   senão
      menor ← C
escreva 'Menor valor = ', menor
```

A idéia do algoritmo anterior é a seguinte: dados valores às variáveis **A, B** e **C**, se o valor em **A** é menor que em **B** e também menor que em **C,** então ele é o menor dos três e o problema fica resolvido. Caso contrário então o menor entre os três estará obviamente em **B** ou **C**. Assim se o valor em **B** é menor que em **C**, **B** contém o menor, do contrário, **C** é que contém. Uma das três atribuições ocorrerá devido aos testes de decisão e colocará na variável **menor** a solução.

Estudo de Caso – Equação do Segundo Grau: Uma equação do segundo grau tem forma, $a \cdot x2 + b \cdot x + c = 0$, onde **a, b** e **c** são coeficientes constantes e **x** a variável independente. Tais equações possuem raízes reais se o coeficiente **a**

é não nulo e ainda se é maior ou igual a zero, onde, $= b^2 - 4 \cdot a \cdot c$. O algoritmo seguinte resolve o problema de determinação das raízes de uma equação de segundo grau cujos coeficientes são fornecidos via entrada padrão:

```
declare
   a, b, c, delta, x0, x1: real
escreva 'Forneça coeficientes da equação: '
leia a, b, c
se (a≠0) então
   delta ← b*b – 4*a*c
   se (delta≥0) então
      x0 ← (-b + raiz(delta))/(2*a)
      x1 ← (-b - raiz(delta))/(2*a)
      escreva 'Raízes: ', x0, x1
   senão
      escreva 'Raízes Complexas!'
senão
   escreva 'A equação não é de segundo grau!'
```

Neste algoritmo o primeiro processo de decisão verifica nulidade do coeficiente em a. Se o valor em a é nulo a mensagem 'A equação não é de segundo grau!' é impressa e o algoritmo se encerra. Caso não seja nulo significa que se trata de uma equação de segundo grau e que o primeiro bloco indentado será executado. Este bloco inicia com o calculo do delta da equação o qual é colocado na variável **delta**. Com este valor um novo processo de decisão é iniciado (aninhamento de decisão) o qual trata da existência de raízes reais. Se o valor em **delta** é maior ou igual a zero então as raízes são calculadas (a função **raiz**() calcula a raiz quadrada de um valor real), impressas e o processo finalizado. Do contrário a mensagem 'Raízes Complexes!' é impressa e o algoritmo se encerra.

Estudo de Caso – Tipo de Triângulo em Relação aos Lados: Um triângulo pode possuir os três lados idênticos (eqüilátero), dois lados idênticos e o terceiro diferente (isósceles) e os três lados diferentes (escaleno). O algoritmo seguinte resolve o problema de determinação do tipo de um triângulo cujos lados são fornecidos via entrada padrão:

```
declare
   a, b, c: inteiro
leia a, b, c
se (a<b+c ∧ b<a+c ∧ c<a+b) então
   se (a=b ∧ a=c) então
      escreva 'Eqüilátero!'
   senão
      se (a=b ∨ a=c ∨ b=c) então
         escreva 'Isósceles!'
      senão
         escreva 'Escaleno!'
senão
   escreva 'O triângulo não existe!'
```

Este algoritmo possui três níveis de indentação de decisões aninhadas. O primeiro avalia a existência do triângulo exibindo a mensagem 'O triângulo não existe!' caso os lados (lidos através das variáveis **a**, **b** e **c** na entrada padrão) não formem um triângulo válido. Se o triângulo existe o processo de decisão do segundo nível de indentação decide entre triângulos eqüiláteros e não eqüiláteros.

Caso seja, a mensagem 'Equilátero!' é impressa e o algoritmo se encerra. Caso não, outro processo de decisão se inicia no terceiro nível de indentação o qual decidirá se o triângulo não-equilátero é isósceles ou escaleno. Para ser isósceles basta haver igualdade de pelo menos um par de lados (por isso a expressão booleana, **a=b** ∨ **a=c** ∨ **b=c**). Neste caso a mensagem 'Isósceles!' é impressa e o algoritmo finaliza. Do contrário (nenhum par de lados se iguala) a mensagem *'Escaleno!'* é impressa e o algoritmo se encerra.

3.3. Repetição

As estruturas de decisão permitem a modelagem de apenas parte dos problemas resolvidos por algoritmos. As estruturas de controle de **repetição** estendem largamente esta capacidade de modelagem. Elas fornecem meios de repetição de execução de blocos de códigos específicos. A associação entre capacidades de decisão e repetição constitui prática comum na construção de algoritmos.

Uma estrutura de controle de repetição encapsula apenas um bloco de código e o repete mediante um critério que notavelmente muda durante o processo (Naturalmente este bloco pode conter internamente outros blocos com nível de indentação acima e que também terão execução repetida). Cada re-

petição é conhecida como **iteração**. Estruturas de repetição são comumente chamadas de **laços** ou **loops**. A seguir são estudadas as principais estruturas de laços utilizadas na construção de algoritmos.

a) Laços controlados por contador

Um laço é controlado por contador quando uma variável auxiliar, chamada contador, usualmente de tipo inteiro, é responsável pela contabilização das iterações. A estrutura de controle para laço com contador é, **para...até...faça**, e possui a seguinte sintaxe geral:

```
    para <contador> ← <limite inferior> até <limite superio>
faça
        <bloco>
```

Esta estrutura efetua a repetição sistemática do código presente no bloco encapsulado posto nas linhas logo abaixo do cabeçalho da estrutura e indentado um nível acima deste. A execução deste laço se faz pela progressão aritmética do contador desde o limite inferior ate o limite superior. Esta progressão é automática e acrescenta uma unidade ao contador ao final de cada iteração. Conseqüentemente em cada iteração todo o bloco encapsulado executa com o mesmo valor de contador que terá uma unidade a mais na iteração seguinte. Quando o laço encerra o último valor atribuído a variável de contagem é o do limite superior. O operador ← é constituinte desta estrutura enfatizando que em cada iteração do laço ocorre uma atribuição (ao contador). O exemplo a seguir ilustra o uso desta estrutura:

```
declare
   i: inteiro
para i ← 1 até 100 faça
   escreva i
```

A execução deste algoritmo imprime os números de 1 a 100 na saída padrão. O bloco da estrutura é formado por uma única linha de instrução. Um outro exemplo:

```
declare
   i, x: inteiro
para i ← 1 até 50 faça
   x ← 2*i
   escreva x
```

A execução deste algoritmo imprime na saída padrão os números pares entre 2 e 100. Em cada iteração o valor em **x** é recalculado em função do valor do contador **i**.

Uma alternativa sintática ao laço com contador, e que melhora o desempenho de muitos algoritmos, é o uso do **passo**. Por definição o passo é um valor inteiro que deve ser somado ao contador quando encerrar uma iteração de laço e iniciar outra. O passo nos laços mostrados anteriormente valem 1. A sintática básica do laço com contador e passo é a seguinte:

```
para <contador> ← <limite inferior>, <limite superio>, <passo>
faça
   <bloco de pseudocódigo>
```

As mudanças nesta nova versão em relação a anterior foram: removeu-se a palavra chave até e no seu lugar colocou-se uma vírgula; após o limite superior marcou-se uma nova vírgula seguindo-a com o valor do passo; por fim a palavra chave **faça**. O exemplo dos números pares pode ser reescrito da seguinte forma:

```
declare
   i: inteiro
para i ← 2, 100, 2 faça
   escreva i
```

Neste algoritmo o contador **i** inicia com valor 2 e em cada iteração é incrementado de 2 (passo) até atingir 100.

Note que dependendo do passo o contador pode passar por cima do limite superior. Quando isso acontece o laço encerra naturalmente quando o contador atinge o primeiro valor acima do limite superior, mas sem processá-lo. Exemplo:

```
declare
   i: inteiro
para i ← 7, 30, 3 faça
   escreva i, ' '
```

Este laço imprimirá:

```
7 10 13 16 19 22 25 28
```

O valor 31 não será impresso porque extrapola o limite superior do laço.

Quando o valor do passo é negativo é possível montar laços com contador decrescente porque neste caso ocorre sucessivamente a subtração ao invés da adição. Exemplo:

```
declare
   i: inteiro
para i ← 20, 0, -1 faça
   escreva i, ' '
```

Este último laço imprimirá na saída padrão os números de 0 a 20 em ordem decrescente e separados por um espaço.

Estudo de Caso – Fatorial: O fatorial de um número inteiro é igual ao produto entre ele e todos os seus anteriores positivos maiores que zero. O algoritmo seguinte resolve o problema do fatorial:

```
declare
   n, i, fat: inteiro
leia n
fat ← 1
para i ← 1 até n faça
   fat ← fat*i
escreva fat
```

O funcionamento deste algoritmo é explicado a seguir. A variável **n** é implicitamente modificada pelo comando **leia** e mantém seu valor até o fim da execução. A variável fat é inicializada com o elemento neutro da multiplicação, **1**. Em cada iteração do laço para o valor em fat é multiplicado pelo valor no contador **i** e o resultado atribuído à própria variável **fat**. Assim, em cada iteração, **fat** irá conter o fatorial do valor corrente no contador **i**. Como o contador de estende até o valor em **n**, então o último valor atribuído a **fat** é o fatorial do valor em **n**.

Este processo de constante atualização de uma variável em um laço por operações de multiplicação é denominado de **acumulação por produto** e é arquétipo de diversos outros algoritmos. A variável modificada normalmente é inicializada com o elemento neutro da multiplicação para não afetar o resultado gerado no processo iterativo.

Estudo de Caso – Potência com Expoente Inteiro: Seja a potência, **be**, com base **b** real e expoente e inteiro. O algoritmo seguinte resolve o problema de cálculo de tais potências dados base e expoente na entrada padrão:

```
declare
   i, b, e, pot: inteiro
leia b, e
pot ← 1
para i ← 1 até e faça
   pot ← pot*b
escreva pot
```

Este algoritmo constitui outro exemplo de acumulação por produto. Sua única diferença para o algoritmo de cálculo de fatorial é o valor pelo qual a variável de acumulação, neste caso **pot**, é multiplicada. Para o fatorial tal valor varia e é igual ao valor no contador. No cálculo de potência, este valor é constante e igual a base (na variável **b**).

Estudo de Caso – Somatório Seletivo: Quantos números menores que um milhão são divisíveis por sete e não são divisíveis por três? Ilustramos dois algoritmos para resolução deste problema. Primeiro algoritmo:

```
declare
   n, cnt: inteiro
cnt ← 0
para n ← 1 até 1000000 faça
   se (n mod 7 = 0 ∧ n mod 3 ≠ 0) então
      cnt ← cnt + 1
escreva 'Total: ', cnt
```

Neste algoritmo o laço de contador em **n** efetua um milhão de iterações. Este laço encapsula a estrutura de decisão **se** (um nível de indentação logo acima de **para**). Em cada iteração, se o resto da divisão do valor em **n** por sete é nulo (ou seja, divisível por 7) e o resto da divisão por 3 for diferente de nulo (ou seja, não divisível por 3), então o valor na variável **cnt** é incrementado em uma unidade (atribuição colocada um nível de indentação acima da estrutura de decisão). Como antes do laço **cnt** foi iniciada em zero (elemento nutro da adição), então após finalização do laço cnt conterá o valor procurado.

Este processo de constante atualização de uma variável em um laço por operações de adição é denominado de **acumulação por soma** e é arquétipo de diversos outros algoritmos. A variável modificada normalmente é inicializada com o elemento neutro da adição para não afetar o resultado gerado no processo iterativo.

A segunda forma de resolver o mesmo problema tem melhor desempenho que a primeira. O algoritmo proposto é o seguinte:

```
declare
   n, cnt: inteiro
cnt ← 0
para n ← 0, 1000000, 7 faça
   se (n mod 3 ≠ 0) então
      cnt ← cnt + 1
escreva 'Total: ', cnt
```

Neste algoritmo o laço possui passo igual a 7 e limite inferior igual a 0. Isso faz com que o contador assuma apenas valores divisíveis por 7 e obviamente reduza o número de iterações a um sétimo do total no algoritmo anterior. Como o valor do contador já é divisível por 7 em todas as iterações, então o aumento do valor em **cnt** fica condicionado apenas a não divisibilidade do valor em **n** por 3. Por essa razão a expressão booleana se reduz a um único teste relacional.

b) Laços controlados por teste

Cada iteração de um laço está associada a um teste que será responsável pela finalização ou continuação deste. Nos laços com contador este teste está embutido na própria estrutura: ele consiste em verificar se o contador já alcançou ou ultrapassou o limite superior. Há, entretanto laços onde tais testes são explícitos e aparecem como expressões booleanas. Há dois tipos importantes de laços com teste: de pré-teste e de pós-teste. Analisaremos cada um deles nas linhas seguintes.

Laços de **pré-teste** são aqueles que no começo de cada iteração realizam um teste para determinar sua continuidade. A estrutura de controle para pré-teste é, **enquanto...faça**, e possui sintaxe geral como segue:

```
enquanto <expressão booleana> faça
   <bloco>
```

A semântica funcional desta estrutura é descrita a seguir. O bloco encapsulado é repetido diversas vezes (iterações). Antes de cada iteração a expressão booleana à entrada é avaliada. Se seu valor for Verdadeiro significa que a próxima iteração deverá ocorrer do contrário o laço deverá encerrar-se. Nenhuma instrução do bloco é executada se na primeira avaliação o resultado for Falso. Segue um exemplo:

```
declare
   x: inteiro
x ← 0
enquanto (x<20) faça
   escreva x, ' '
   x ← x + 2
```

Neste algoritmo, enquanto a expressão, **x<20**, for Verdadeira as duas instruções encapsuladas serão executadas. É importante observar que a variável n funciona como contador, apesar de não ser integrante da estrutura. Por essa razão o valor nesta variável precisa ser explicitamente inicializado fora do laço (**x←0**, na terceira linha) e modificado manualmente dentro do laço (**x←x+2**, na última linha). A saída deste algoritmo é a seguinte:

```
0 2 4 6 8 10 12 14 16 18
```

O valor 20 não é impresso porque não satisfaz a expressão booleana (20 é igual a 20 e não menor).

Observe que, ao contrário do que ocorre nos laços por contador, é possível mudar o local onde a variável de controle se modifica. Se, por exemplo, as duas linhas do bloco encapsuladas por enquanto no algoritmo anterior fossem trocadas de posição o código ainda seria válido, mas a saída impressa se modificaria para:

```
2 4 6 8 10 12 14 16 18 20
```

Esta saída deixa de imprimir o 0 e imprime 20 simplesmente porque as impressões agora vêem depois da modificação da variável **x**..

Laços de **pós-teste** são aqueles que ao final de cada iteração realizam um teste para determinar sua continuidade. A estrutura de controle para pós-teste é, **repita...até**, e possui sintaxe geral como segue:

```
repita
   <bloco>
até <expressão booleana>
```

A semântica funcional desta estrutura é descrita a seguir. O bloco encapsulado é repetido diversas vezes (iterações). Após o final de cada iteração a expressão booleana à saída é avaliada. Se seu valor for Falso significa que

a próxima iteração deverá ocorrer do contrário o laço deverá encerrar-se. O bloco encapsulado é executado pelo menor uma vez independente da expressão booleana. Ao passo que a estrutura **enquanto...faça** executa código enquanto algo é verdade, **repita..até** executa código até que algo seja verdade. Segue um exemplo:

```
declare
   x: inteiro
x ← 0
repita
   escreva x, ' '
   x ← x + 2
até (x>20)
```

O funcionamento deste algoritmo e similar ao último algoritmo com pré-teste apresentado. Como o teste fica na saída, o valor em **x** é sempre impresso na primeira iteração (mesmo que fosse maior que 20). O laço só é interrompido quando o valor em **x** atinge 22 (o qual não é impresso).A saída deste algoritmo é:

```
0 2 4 6 8 10 12 14 16 18 20
```

c) Aninhamento de laços

Aninhar laços significa encapsular laços dentro de outros. Se um laço A é encapsulado por um laço B então para cada iteração de B o laço A efetuará sistematicamente todas suas possíveis iterações naquele contexto. Por exemplo:

```
declare
   i, j: inteiro
para i ← 1 até 10 faça
   para j ← 1 até 10 faça
      escreva i+j
```

Os dois laços no algoritmo anterior são construídos para executar 10 vezes cada. O primeiro tem contador em **i** e o segundo contador em **j**. Entretanto o primeiro lado encapsula o segundo laço (posto um nível de indentação acima). Isso faz com que cada iteração do primeiro laço execute todas as 10 iterações do segundo laço e conseqüentemente execute 100 vezes a instrução **escreva** (posta um nível de indentação acima do segundo laço).

Os limites dos contadores dos laços podem ser expressos como função de qualquer variável inteira do programa (exceto o próprio contador). Isso inclui também os contadores de laços hierarquicamente acima em um dado aninhamento. Veja o exemplo:

```
declare
   i, j, cnt: inteiro
cnt ← 0
para i ← 1 até 10 faça
   para j ← i até 10 faça
      cnt ← cnt + 1
escreva cnt
```

Neste algoritmo o segundo laço é aninhado no primeiro. O limite inferior do segundo laço (de contador **j**) é o valor corrente do contador do primeiro laço, **i**. À proporção que o primeiro laço avança o número de iterações do segundo laço diminui como indica a tabela a seguir:

Valor de **i**	1	2	3	4	5	6	7	8	9	10
Número de iterações do segundo laço	10	9	8	7	6	5	4	3	2	1

O algoritmo utiliza acumulação por soma para guardar na variável **cnt** o total de iterações. Este valor também pode ser calculado pela somatória dos valores na segunda linha da tabela anterior, ou seja, 10+9+8+7...+1 = 55. Este será o valor impresso na saída padrão.

Não há restrições sobre os tipos de estruturas de repetição aninhadas. Assim, por exemplo, laços de contagem podem estar aninhados a laços de teste ou vice-versa. Também não há restrições sobre aninhamento entre estruturas de decisão e de repetição. De fato a associação coerente de todas estas estruturas é que possibilita a criação de algoritmos aplicáveis.

d) Quebra e continuação de laços

Um laço controlado por teste é denominado **laço infinito** quando a expressão booleana de controle é sempre avaliada como Verdadeira (no caso de enquanto...faça) ou sempre como Falsa (no caso de **repita...até**). Laços infinitos são aplicados em situações onde o critério de parada não se encaixa na entrada (ou na saída) do laço. Quando instruções internas ao bloco de laço promovem sua suspensão forçada (sem uso da expressão booleana de controle) diz-se que houve **quebra de laço**. Uma quebra de laço é construída utilizando o comando **pare**. Veja o exemplo:

```
declare
   ch: caractere
enquanto (V) faça
   ch ← tecla
   caso (ch) seja
      'x': escreva 'olá mundo!!'
      'y': escreva 'hello world!'
      'z': escreva 'hallo Welt!!'
      senão pare
```

No algoritmo anterior o laço construído com **enquanto** é infinito devido à expressão de controle ter sido substituída por um V (Verdadeiro). Em cada iteração uma **tecla** é solicitada pelo comando tecla e seu caractere armazenado na variável **ch**. Caso o valor em ch seja '**x**', '**y**' ou '**z**' então uma de três mensagens é enviada a saída padrão e o laço inicia novamente com nova solicitação de tecla. Se a tecla pressionada não é nenhuma das três então o **pare** do bloco rotulado **senão** é executado suspendendo o laço.

As regras de funcionamento do comando **pare** são as seguintes:

- Se pare for executado fora de laços, mesmo encapsulado por alguma estrutura de decisão, ocorrerá suspensão da execução do algoritmo como um todo.
- Se pare for executado dentro de um laço sem aninhamento com outros laços, mesmo que esteja encapsulado por alguma estrutura de decisão, apenas tal laço será suspenso, mas o algoritmo não necessariamente.
- Se pare for executado em um aninhamento de laços, mesmo havendo aninhamento de estruturas de decisão, apenas o laço hierarquicamente mais próximo a este comando será suspenso, mas o algoritmo não necessariamente.

Estudo de Caso – Números Primos: Um número primo é aquele que é divisível apenas por um e por ele mesmo. Como um número não é divisível pelos que o sucedem é razoável acreditar que um dado inteiro n, que se quer testar se é primo ou não, deva ser dividido pelos números no intervalo fechado [2, n-1] sendo primo se todas estas divisões obtiverem resto não nulo e não primo se pelo menos uma for nula (encontrada uma divisão com resto nulo o número não é mais primo e as demais divisões, se ainda restarem, tornam-se desnecessárias). Matematicamente, entretanto, tantas divisões não são necessárias Um intervalo suficiente de testes pode ser reduzido para [2, d] tal que $d2 \leq n$.

Com base no critério anunciado acima foi construído um algoritmo capaz de escrever, na saída padrão, todos os números primos menores que cinco mil. Segue:

```
declare
   n, d: inteiro
   b: lógico
pata n ← 2 até 5000 faça
   b ← V
   d ← 2
   enquanto (d*d ≤ n) faça
      se (n mod d = 0) então
         b ← F
         pare
      d ← d+1
   se (b) então
      escreva n, ' '
```

O laço **para** neste algoritmo submete em cada iteração o valor de seu contador **n** a um teste para verificar se ele é ou não valor primo. O teste possui quatro etapas:

(I) Atribuição à variável **b** do valor Verdadeiro indicando a primeira hipótese sobre a natureza do valor em **n** (ou seja, é primo até que se prove o contrário).

(II) A variável **d** recebe valor 2 indicando o início do intervalo de valores pelos quais o valor em **n** deve ser dividido.

(III) O laço **enquanto** testa a divisibilidade de **n** pelos valores no intervalo [2,3,..,d,..,D], com $\mathbf{D^2}$ **n**, interrompendo o laço quando o primeiro resto igual a zero é encontrado. Se esta interrupção acontece o valor em **b** muda para Falso (foi provado o contrário: o numero não é primo) e o laço **enquanto** é suspenso por um **pare** (não há mais sentido testar outras divisibilidades).

(IV) O valor em **b** é verificado para ver se o valor corrente no contador **n** é primo. Se for (ver expressão booleana formada apenas por **b**) ele é impresso na saída padrão.

Neste último exemplo, utilizando-se limite inferior igual a 3 e passo igual a 2 no laço **para**, testam-se apenas valores ímpares (o único primo par é 2) aumentando-se a velocidade do algoritmo.

Síntese do capítulo

Neste capítulo foram apresentados elementos essenciais para a construção de algoritmos. Os operadores são objetos de semântica bem definida e que são classificados como: lógicos, relacionais e aritméticos. Em seguida, abordamos elementos construtivos: estruturas de decisão e de repetição que são fundamentais para a construção de algoritmos. Finalizando, ilustramos seis estudos de casos que utilizam todos os elementos abordados no capítulo.

Contamos que você já esteja fera no jogo sugerido no final do capítulo 1 e que agora com os conceitos aprendidos neste capítulo você consiga progredir e utilizar as estruturas de decisão e repetição abordadas aqui.

Atividades de avaliação

1. Faça um algoritmo para calcular o quadrado de um número.
2. Faça um algoritmo para calcular a média das notas de uma turma de 20 alunos.
3. Construir um algoritmo que receba três valores inteiros quaisquer e os imprima em ordem crescente.
4. Faça um algoritmo para calcular o fatorial de um número N inteiro dado.
5. Faça um algoritmo que, dados números de zero até um máximo, mostre a soma de todos os não-primos, subtraída da soma dos primos.
6. Faça um algoritmo que determine se um número é divisível ou não por outro.
7. Construir algoritmos para o calculo do número de dígitos dos valores: (a) 100! (b) 2^{1000}
8. Considere uma moeda que contenha cédulas de 50,00, 10,00, 5,00 e 1,00. Considere ainda que caixas eletrônicos de um banco operem com todos os tipos de notas disponíveis, mantendo um estoque de cédulas para cada valor. Os clientes deste banco utilizam os caixas eletrônicos para efetuar retiradas de uma certa quantia. Escreva um algoritmo que, dado o valor da retirada desejada pelo cliente, determine o número de cada uma das notas necessário para totalizar esse valor, de modo a minimizar a quantidade de cédulas entregues.
9. Faça um algoritmo para calcular a soma de todos os números inteiros menores que 100 e que não sejam divisíveis por 3.
10. O reverso de um número natural é o novo número obtido pela inversão de seus dígitos, ex: o reverso de 127 é 721. Escreva algoritmo que receba um número natural e escreva seu reverso.

Leituras, filmes e sites

http://armorgames.com/play/2205/light-bot

Referências

PEREIRA, SILVIO DO LAGO. **Estruturas de Dados Fundamentais:** Conceitos e Aplicações. 9ª Edição. Editora Érica 2006. ISBN: 8571943702 .

FERNANDEZ, M; Cortés, M. **Introdução à Computação**. 1ª Edição. Fortaleza. RDS Editora, 2009.

Capítulo 3

Variáveis indexadas

Objetivos

- Apresentar a estrutura matriz, a qual é utilizada para manipulação de variáveis com capacidade de armazenar na memória mais de um valor ao mesmo tempo. Para manipular as variáveis utiliza-se o mecanismo de indexação.

1. Matrizes

1.1. Introdução a matrizes e vetores

Uma **variável escalar** é uma entidade abstrata que se refere a uma célula de memória e que possui um tipo relacionado, uma faixa restrita de representação e a capacidade de armazenar um único valor. Há, entretanto, a possibilidade de definir variáveis com capacidade de guardar mais de um valor ao mesmo tempo. Tais variáveis são denominadas **matrizes**.

Naturalmente os valores armazenados por matrizes não compartilham a mesma célula de memória: de fato mais de uma célula estará presente na definição de matrizes. Cada uma destas células funciona como uma variável escalar independente o que faz das matrizes aglomerados de variáveis escalares. Todas essas variáveis constituintes são do mesmo tipo de forma que podemos ter, por exemplo, matrizes de inteiros, lógicos ou de reais, mas nunca a mistura deles. Denomina-se **tipo base** o tipo de dados comum a todos os aglomerados da matriz.

Uma variável matriz, assim como todas as demais, possui apenas um nome. Então como manipular cada uma das variáveis escalares (células) que a constitui? O mecanismo para se referenciar cada uma das células de uma matriz é denominado **indexação**. Assim, por exemplo, se M é uma matriz dos primeiros dez ímpares então M[1] = 1, M[2] = 3, M[3] = 5 e assim por diante: aqui os colchetes retratam hospedeiros dos índices que neste caso vão de um a dez acessando independentemente cada valor em M. O vetor M ainda pode ser descrito desta forma: M[k]=2k-1, com o índice k no intervalo {1..10}.

Matrizes podem contar com um ou mais índices. Cada índice de uma matriz está associado a uma **dimensão** da matriz. Assim matrizes unidimensionais precisam de um índice, bidimensionais de dois e assim por diante. As matrizes na matemática se confundem com as matrizes bidimensionais aqui descritas.

Matrizes unidimensionais são comumente denominadas de **vetores**. Os vetores representam a forma de matriz mais comumente empregada em algoritmos.

O total de aglomerados de um vetor representa o **comprimento do vetor**. O **tamanho de um vetor** é a memória total que ele ocupa, ou seja, seu comprimento multiplicado pelo tamanho em bytes do tipo base. Assim, por exemplo, se H é um vetor formado de 5 inteiros de 2-bytes, então o comprimento de H é 5, seu tipo base é inteiro, e seu tamanho vale 10-bytes.

1.2. Matrizes na memória

Como estão dispostas as matrizes na memória? Se pensarmos em termos de vetores é coerente imaginar que as células constituintes são dispostas vizinhas umas as outras: de fato é exatamente isso o que acontece. Quando um vetor precisa ser alocado, então a memória requisitada deverá ser um bloco uniforme de bytes com o tamanho total do vetor. Encontrado o bloco ele é então marcado como usado e vinculado a variável vetor no programa.

Seja, por exemplo, um vetor formado de 50 células de inteiros de 2-bytes vinculado à variável X. Assim, após alocação, um bloco de 100 bytes estará vinculado a X e as 50 células poderão ser acessadas independentemente por X[1], X[2], X[3],..., X[50]. O único endereço real disponível é o do início do bloco de memória. E o de cada célula? Como funciona uma chamada a X[k], com k inteiro e entre 1 e 50? A resposta é simples: através de aritmética elementar, ou seja, manipular X[k] significa manipular diretamente o conteúdo no endereço de memória $\xi+(k-1)\cdot 2$.

O acesso a células individuais via vetores é o mais eficaz se comparada a outras formas de armazenamento linear (como listas encadeadas).

E as matrizes de mais de uma dimensão? Também tem células contiguas em memória? A resposta é sim. Quando mais de uma dimensão está envolvida a memória total utilizada será o produto entre o tamanho do tipo base e todos os comprimentos das dimensões. Por exemplo, seja Q uma matriz bidimensional 3×3 de inteiros como segue:

```
|12     -7     14|
|1      -9     -5|
|11     78     25|
```

Se Q é composto por inteiros de 2-bytes então o bloco de memória alocado terá 18 bytes. O aspecto em memória (mais comumente adotado) para o exemplo anterior é:

12	−7	14	1	−9	−5	11	78	25

1.3. Declaração, inicialização e uso de vetores

A sintaxe geral para declaração de vetores é a seguinte:

```
declare
   <variável>[comprimento]: <tipo>
```

Segue um exemplo:

```
declare
   M[20], i: inteiro
para i ← 1 até 20 faça
   M[i] ← i*i
```

Neste código duas variáveis são declaradas: um inteiro **i** e um vetor de inteiros **M**. A variável **M** comporta vinte valores inteiros e é utilizada para armazenar, por ação de um laço, os quadrados dos números de um a vinte. Vale observar que o contador i do laço funciona tanto como índice do vetor quanto como componente de expressão.

Há duas aplicações para o par de colchetes. A primeira é **declarativa**, ou seja, eles abrigam o comprimento do vetor declarado conforme se verifica na definição da sintaxe geral. A segunda é **evocativa**, ou seja, o conteúdo entre os colchetes é um valor de índice entre 1 e seu comprimento (definido na declaração) e possibilita o acesso a uma célula específica do vetor. Tal acesso permite tanto leitura quanto modificação do que está na célula indexada.

Consideremos o exemplo seguinte:

```
declare
   M[10], i, imax: inteiro
para i ← 1 até 10 faça
   leia M[i]
imax ← 1
para i ← 2 até 10 faça
   se (M[i] > M[imax]) então
      imax ← i
escreva M[imax]
```

Este algoritmo solicita dez valores do usuário e imprime como saída o maior entre eles. O primeiro laço é o responsável pela entrada de dados. Em cada iteração o comando **leia** modifica uma célula distinta do vetor **M** de modo a preenchê-lo convenientemente. O restante do código é um processo de busca. Tal busca começa com a hipótese de que o maior valor está na primeira posição (assim imax recebe 1).

As demais posições são testadas progressivamente pelo laço para: cada vez que uma célula verificada possui valor maior que aquele de índice **imax**, então **imax** recebe o valor do contador. Quando o laço encerra imax possui a posição contendo o maior valor e assim **M[imax]** é impresso. Não há aninhamento de laços neste exemplo e por essa razão os dois laços estão no mesmo nível de indentação.

Apesar de não estar disponível na maioria das linguagens de programação introduzimos nessa abordagem uma opção de inicialização de vetores diretamente do código. A sintaxe geral é a seguinte:

```
declare
   <variável _ 1>[c1]: <tipo>
//...
<variável _ 1> ← {<valo1 _ 1>, <valo1 _ 2>, <valo1 _ 3>,..., <valo1 _
c1>}
//...
```

Um exemplo de aplicação:

```
declare
   M[10], i, imax: inteiro
M ← {12, -9, 33, 8, -23, 17, 6, -5, 31, 2}
imax ← 1
para i ← 2 até 10 faça
   se (M[i] > M[imax]) então
      imax ← i
escreva M[imax]
```

Nesta sintaxe de inicialização de vetores, o par de colchetes não é necessário, as chaves (que envolvem os elementos) são obrigatórias e ainda deve haver vírgulas entre os elementos. A execução deste algoritmo imprime 33 na saída padrão.

Estudo de Caso – Dois maiores valores de um Vetor. O algoritmo seguinte encontra num vetor dado os dois maiores valores que ele contém.

```
declare
   M[10], i, im1, im2: inteiro
M ← {-12, 9, 30, 18, 13, 7, -6, 5, 31, 1}
im1 ← 1
im2 ← 2
se (M[im1] < M[im2]) então
   i ← im1
   im1 ← im2
   im2 ← i
para i ← 3 até 10 faça
   se (M[i] > M[im1]) então
      im2 ← im1
      im1 ← i
   senão
      se (M[i] > M[im2])
         im2 ← i
escreva M[im1], ' e ', M[im2]
```

Este algoritmo modifica **im1** e **im2** para conterem respectivamente, no final do processo, os índices do maior e do segundo maior valor em **M**. A primeira hipótese feita no algoritmo é a de que o maior valor está na posição 1 e o segundo maior está na posição 2.(atribuições, **im1←1** e **im2←2**). No passo

seguinte a estrutura de decisão, **se**, verifica e possivelmente corrige esta hipótese (troca de valores) considerando apenas os dois primeiros valores em **M**.

As demais posições do vetor (3 a 10) são verificadas no laço para e tratadas da seguinte forma: quando contiverem um valor maior que o maior até então (em **im1**) então a nova posição do segundo maior valor passa a ser a em **im1** (**im2←im1**) e a nova posição do maior valor passa a ser a do contador (**im1←i**); mas se o valor em **i** não for maior que aquele em **im2**, então ainda poderá ser maior que aquele em **im2** e neste caso faz-se, **im2←i**. As variáveis im1 e im2 valem no final respectivamente 9 e 3 e a mensagem impressa é '31 e 30'.

Estudo de Caso – Ordenação de Vetores. O algoritmo seguinte implementa a ordenação de vetores pelo **método da seleção**:

```
declare
   M[10], i, j, t, x: inteiro
M ← {-12, 9, 30, 18, 13, 7, -6, 5, 31, 1}
//Ordenação por seleção:
para i ← 1 até 9 faça
   t ← i
   para j ← i+1 até 10 faça
      se (M[j] < M[t]) então
         t ← j
   x ← M[t]
   M[t] ← M[i]
   M[i] ← x
// Impressão:
para i ← 1 até 10 faça
   escreva M[i]
```

A ordenação por seleção é construída pelo aninhamento de dois laços. O laço externo percorre o vetor da primeira à penúltima posição. O laço interno busca pela posição, entre a posição corrente do laço externo e o fim do vetor, contendo o menor valor. Esta busca usa a variável **t** e principia com a hipótese de que o menor valor tem posição em **i** (assim, **t←i**). A execução do laço interno (e conseqüentemente de seus testes de decisão) modifica **t** (possivelmente) para conter o índice do menor valor entre o valor em **i** e 10 (observe que este laço inicia em **i+1** para excluir a hipótese inicial). As últimas três linhas do laço externo (a indentação mostra que elas não fazem parte do laço interno) fazem a troca entre o elemento na posição em t e aquele na posição em **i**, ou seja, na posição em **i** fica o menor elemento do intervalo [$\mathbf{i}_{\text{corrente}}$, 10]. O laço externo provoca 9 processos de busca que acabam por

ordenar o vetor. O último elemento do vetor não é considerado no laço externo porque não há elementos à sua frente para que ocorram trocas. O laço final imprime a seqüencia seguinte na saída padrão:

```
-12   -6     1      5      7      9      13     18     30     31
```

1.4. As cadeias de caracteres

Se um vetor é constituído por caracteres então ele é denominado uma **cadeia de caracteres** ou simplesmente **string**. Cadeias de caracteres são utilizadas para representar mensagens ou mesmo textos completos. Há duas características importantes das strings que as destacam em relação aos vetores em geral. A primeira é o conceito de **comprimento de cadeia** e a segunda são as formas peculiares que elas possuem de inicialização, atribuição, escrita e leitura.

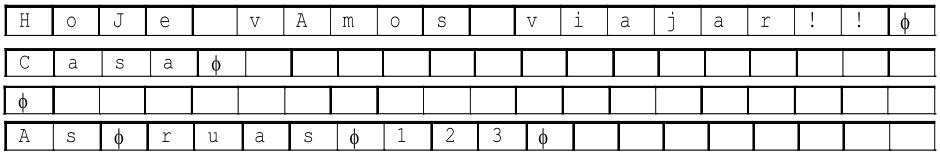
O comprimento de uma cadeia de caracteres é a quantidade de caracteres medida do primeiro caractere até o caractere que precede a primeira ocorrência do caractere nulo. (aquele cujo ASCII é zero). Denotaremos este caractere neste texto por ϕ. Esta estratégia é responsável pela capacidade que strings têm de armazenar mensagens de tamanho variável, ou pelo menos com tamanho mínimo (zero) e máximo (comprimento do vetor menos um). Para entender melhor esta questão consideremos um vetor de comprimento 20 onde se queira armazenar a mensagem 'Hoje vamos viajar'. O aspecto dos caracteres no vetor segue:

H	O	j	e		v	a	m	o	s		v	i	a	j	a	r	ϕ		

O total de caracteres da mensagem é 17, incluindo espaços. Dispô-la no vetor demanda 18 caracteres porque o nulo é posto logo em seguida. Os 2 finais são ignorados, por exemplo, num processo de impressão. O comprimento desta cadeia é exatamente 17 e poderá se estender até no máximo 19 por causa do nulo.

Pelo menos um caractere nulo deverá estar presente num vetor de caracteres: o final da cadeia ocorrerá sempre no caractere que antecede a primeira aparição de ϕ.

Outras situações para o vetor anterior:



Respectivamente essas strings tem comprimentos 19, 4, 0 e 2.

Para inicializar uma cadeia de caracteres pode-se utilizar a sintaxe apresentada anteriormente. Exemplo:

```
declare
   S[10]: caractere
S ← {'a', ' ', 'c', 'a', 's', 'a', ϕ}
```

Neste exemplo cada caractere foi colocado em sua respectiva posição no vetor, incluindo o nulo. Para guardar a mensagem 'a casa', entretanto, são necessários apenas 7 caracteres (incluindo ϕ). A lógica é simples: não excedendo o comprimento do vetor, qualquer disposição de caracteres terminada em nulo é viável. Caso seja inferior ao comprimento do vetor, o restante, apesar de ainda disponível, é simplesmente ignorável.

Como já mencionado, as cadeias de caracteres possuem inicialização peculiar. Ao invés de usar chaves, vírgulas e caracteres soltos, convencionaremos que as cadeias poderão ser inicializadas por atribuição direta de uma mensagem. Veja como fica o exemplo anterior:

```
declare
   S[10]: caractere
S ← 'a casa'
```

Neste caso não aparece explicitamente o nulo, mas convencionamos que a inicialização por esta forma efetua automaticamente sua inserção.

Exclusivamente para as cadeias de caracteres, os comandos **leia** e **escreva** funcionam análogos às suas versões para variáveis escalares. Exemplo:

```
declare
   S[100]: caractere
leia S
escreva S
```

Neste algoritmo o usuário fornece uma cadeia via entrada padrão e depois ela é simplesmente reimpressa na saída padrão. Esta é uma forma conveniente de entrada para leia haja vista que pouco processamento adicional é necessário (não esqueça que a entrada padrão gera uma cadeia de caracteres). Cada caractere digitado é convenientemente colocado no vetor. A confirmação de entrada (ENTER no teclado) concatena adicionalmente o nulo à cadeia.

Se o comando **escreva** deve imprimir uma cadeia de caracteres, então ele fará impressão caractere por caractere até que um nulo (não impresso) é encontrado. Exemplo:

```
declare
   S[100]: caractere
S ← 'olá mundo!'
escreva S
```

Apesar do vetor **S** comportar até 100 caracteres, a cadeia de caracteres que ele armazena possui comprimento 10: assim apenas estes caracteres serão impressos por **escreva**. O comportamento de **escreva**, com cadeias de caracteres, é reproduzido com utilização de um laço:

```
declare
   S[100]: caractere
   i: inteiro
leia S
i ← 1
enquanto (S[i] ≠ ϕ) faça
   escreva S[i]
   i ← i + 1
```

Neste algoritmo a cadeia é impressa caractere a caractere sendo um caractere por iteração de laço. Cada um dos caracteres é denotado pela expressão **S[i]** como convém a qualquer vetor. Se o nulo é encontrado, então o laço para (não é impresso). Outra aplicação de leitura caractere a caractere de uma cadeia é mostrada a seguir:

```
declare
   S[100]: caractere
   t, i: inteiro
leia S
t ← 0
enquanto (S[t+1] ≠ ϕ) faça
   t ← t+1
para i ← t, 1, -1 faça
   escreva S[i]
```

Este algoritmo recebe uma cadeia via entrada padrão e a escreve de trás para frente. A idéia é medir primeiramente o comprimento da cadeia usando a variável **t** cujo valor inicial é zero. Com o fim do primeiro laço (**enquanto**) **t** conterá o comprimento da cadeia e um laço por contagem (**para**) poderá fazer o trajeto reverso do vetor a fim de imprimi-lo de trás para frente.

1.5. Vetores dinâmicos

Todas as declarações de vetores nas sessões anteriores possuíam **vinculação estática** de espaço em memória, ou seja, o tamanho que o vetor possuirá em tempo de execução precisa ser definido no código pelo programador. O efeito direto disso é a inflexibilidade, ou seja, se mais memória for necessária não será possível expandir o vetor durante a execução.

Para resolver esta limitação, introduzimos os comandos **alocar** e **limpar**. Eles interagem diretamente com o gerenciador de memória requisitando e devolvendo respectivamente quantidades de memória. Cada solicitação de memória com **alocar** está associada a uma devolução com **limpar**. Memória não devolvida mantém o status de ocupada não podendo ser usada por outros processos. Em máquinas reais, a ocorrência desse quadro causa queda de desempenho ou mesmo parada do sistema (*crash*).

Denominamos **vetor dinâmico** ao vetor com capacidade de definir seu comprimento em tempo de execução. A sintaxe geral de um vetor dinâmico é a seguinte:

```
declare
   <vetor>[]: <tipo base>
```

A diferença para vetores estáticos é a ausência de um comprimento entre os colchetes. Exemplos:

```
declare
   X[]: inteiro
   VET[]: inteiro
```

Para alocar e limpar memória de um vetor dinâmico utiliza-se respectivamente os comandos **alocar** e **limpar**. A sintaxe básica segue:

```
alocar <vetor>[<comprimento>]
...
limpar <vetor>
```

O exemplo a seguir ilustra o uso de vetores dinâmicos:

```
declare
   X[], n, i: inteiro
leia n
alocar X[n]
para i ← 1 até n faça
   X[i] ← 2*i-1
limpar X
```

No exemplo acima o vetor dinâmico **X** tem seu comprimento alocado com um valor de comprimento, **n**, fornecido via entrada padrão. Após a alocação o vetor é carregado com a máxima quantidade de números ímpares que suporta e finalmente desalocado ao final.

Estudo de Caso – Números Pandigitais: um número natural é denominado pandigital se seus algarismos forem todos distintos entre si. O algoritmo seguinte determina se um valor fornecido pelo usuário é ou não pandigital:

```
declare
   Q[], t, n, i, j: inteiro
   b: lógico
leia n
// PARTE I
t ← 0
j ← n
enquanto (j>0) faça
   t ← t + 1
   j ← j div 10
// PARTE II
alocar Q[t]
// PARTE III
j ← n
para i ← t, 1, -1 faça
   Q[i] ← j div 10
   j ← j div 10
// PARTE IV
b ← V
para i ← 1 até t-1 faça
   para j ← i+1 até t faça
      se (Q[i] = Q[j]) então
         b ← F
         pare
   se (¬ b) então
      pare
// PARTE V
se (b) então
      escreva 'o número é pandigital!'
senão
      escreva 'o número NÃO é pandigital!'
limpar Q
```

O algoritmo anterior é seccionado em seis partes explicadas a seguir.

(I) O comprimento em dígitos do valor em **n** é medido. Isso é feito pelo laço **enquanto** que, utilizando uma cópia de **n**, neste caso em **j**, efetua sucessivas divisões por dez. Obviamente este laço se encerra quando um total de iterações igual ao número de dígitos do valor em **n** é realizado (o que conduz **j** a zero e logo ao encerramento do laço). Haja vista que **t** é anulado fora do laço e incrementa em um em cada iteração deste, então o valor final de **t** é o comprimento em dígitos do valor em **n**.

(II) O vetor **Q** tem seu espaço alocado. A estratégia é dedicar uma célula para cada dígito do valor em **n**.

(III) Os dígitos do valor em **n** são dispostos no vetor **Q**. Para realizar tal tarefa, novamente uma cópia de **n** é feita em **j** que mais uma vez, por cíclicas operações de divisão por dez, é reduzido à zero. O laço não é mais por teste e sim de contagem com t iterações (isso provoca o mesmo efeito). Antes de cada divisão, o valor **j mod 10** é colocado na posição **i** de **Q**. Esta operação consiste exatamente em ler o último digito do inteiro em **j**. Como o valor em **j** reduz em cada iteração, então **j mod 10** retorna sistematicamente cada último dígito do número. Assim tais dígitos são tomados de trás para frente justificando o laço de contagem decrescente (observe que nele o contador **i** inicia com o valor em **t** e decresce até 1 causando acesso de trás para frente no vetor). Com o fim do laço o vetor Q conterá os dígitos do atual valor

em **n** na seqüência correta (para simples verificação de pandigitalidade isso não faz diferença, mas ilustramos aqui por ser útil em outros problemas).

(IV) É testado se o vetor **Q** de dígitos possui ou não elementos repetidos. Os dois laços aninhados efetuam sistemáticas comparações entre elementos sendo suspensos (ambos) se alguma igualdade é encontrada. A variável **b**, inicialmente marcada como Verdadeira, indica a hipótese inicial de que há repetições. Se uma igualdade é encontrada tal variável recebe Falso. As chamadas a **pare** que suspendem os laços estão associadas ao valor em **b**. A expressão ¬**b** representa negação do valor em **b** e é necessária para testar se o laço externo (de contador **i**) deve ser ou não suspenso.

(V) Uma mensagem apropriada é impressa na saída padrão de acordo com o último valor em **b**. O vetor **Q** é desalocado.

Estudo de Caso – Transformação em Hexadecimal: a base hexadecimal conta com 16 distintos algarismos para representação de valores numéricos. Os dez primeiros coincidem com aqueles da base decimal ao passo que os seis restantes são denotados pelas cinco primeiras letras do alfabeto. Como há letras e números envolvidos, então hexadecimais são preferencialmente representados como cadeias de caracteres.

Para transformar um decimal em hexadecimal deve-se submetê-lo a sucessivas divisões pela base 16. Os restos destas divisões são progressivamente tomados e seus correspondentes caracteres colocados na string resposta. Se o resto de divisão está entre 0 e 9 então os caracteres correspondentes são respectivamente '0' a '9'. Se o resto está entre 10 e 15 então os caracteres são respectivamente 'A' a 'F'. Os caracteres devem ser postos na string resposta de trás para frente porque os caracteres constituintes são obtidos na ordem inversa.

Para resolver o problema descrito é o sugerido o algoritmo a seguir:

```
declare
   H[], B[17]: caractere
   t, n: inteiro
   b: lógico
// PARTE I
B ← '0123456789ABCDEF'
leia n
// PARTE II
t ← 0
j ← n
enquanto (j>0) faça
   t ← t + 1
   j ← j div 16
      fim
// PARTE III
alocar H[t+1]
H[t+1] ← ϕ
// PARTE IV
j ← t
enquanto (n>0) faça
   H[j] ← B[ 1 + (n mod 16)  ]
   n ← n div 16
   j ← j - 1
// PARTE V
escreva H
limpar H
```

Este algoritmo divide-se em cinco partes descritas a seguir:

(I) Define-se o vetor máscara B e um inteiro é solicitado ao usuário via n para que seja convertido para o formato hexadecimal.

(II) O comprimento em dígitos (na base 16) do valor em n é calculado. Para tanto o valor em j (que é iniciado com uma cópia de **n**) é sucessivamente dividido por 16, no laço **enquanto**, até ser anulado. Em cada iteração **t** (previamente anulado) é incrementado em um. Ao final, o valor em **t** é a quantidade de dígitos do valor em **n** na base 16.

(III) Memória é alocada no vetor dinâmico de caracteres, **H**. O valor de comprimento **t+1** deve-se a necessidade do caractere nulo.

(IV) O laço **enquanto** dispõe, no vetor **H**, cada caractere em sua posição apropriada. Esta posição apropriada é controlada pela variável **j** que inicia com o valor em **t** e decai até 1 no laço (preenchimento reverso). Já os caracteres, que serão postos em **H**, são extraídos da cadeia **B** mediante uso da expressão, **1 + n mod 16**. Como o valor em **n** decai em cada iteração (**n ← n div 16**) então o valor da expressão, n mod 16, representa cada último algarismo (na base 16) extraído do que resta em n e conseqüentemente o valor da expressão, **1 + n mod 16**, é exatamente o índice em **B** que contém o caractere correspondente.

(V) A string é impressa e depois removida da memória.

1.6. Usando mais de uma dimensão

A sintaxe geral para declaração de matrizes com mais de uma dimensão é a seguinte:

```
 declare
      <variável>[comprimento _ 1,  comprimento _ 2,...,comprimento _ n]:
<tipo>
```

Segue um exemplo:

```
declare
   X[3,3]: real
   i, j: inteiro
para i ← 1 até 3 faça
   para j ← 1 até 3 faça
      leia X[i, j]
```

No exemplo acima uma matriz quadrada de ordem 3 é definida e lida com o auxílio de dois laços aninhados. O comando **leia** executado exatamente um total de nove vezes.

Para criar uma matriz dinâmica adotaremos a seguinte sintática: a declaração será idêntica a de um vetor dinâmico, entretanto no momento da alocação serão especificados os comprimentos das dimensões separados por vírgulas. O uso dessas matrizes é idêntico ao de uma estática. Veja o exemplo:

```
declare
   W[], n, i, j: integer
leia n
alocar W[n,n]
para i ← 1 até n faça
   para j ← 1 até n faça
      leia W[i, j]
limpar W
```

No exemplo acima **W** é uma matriz cuja declaração é idêntica a de um vetor dinâmico. Somente na alocação são definidas as dimensões. O uso pós-alocação é idêntico ao de uma matriz comum. Para limpar esta matriz dinâmica a sintaxe é a mesma dos vetores dinâmicos.

Devido a natureza linear da memória alocada, uma matriz poderá ser tratada como vetor mesmo que tenha mais de uma dimensão. Para ilustrar isso consideremos o exemplo a seguir:

```
declare
   VET[], m, n, i, j: integer
leia m, n
alocar VET[m,n]
para i ← 1 até m faça
   para j ← 1 até n faça
      VET[i,j] ← i*j
para i ← 1 até m*n faça
   escreva VET[i]
limpar VET
```

Apesar de **VET** ser dinamicamente definida como uma matriz de m colunas e n linhas e de seus valores serem carregados com o uso de dois índices (pelo aninhamento de dois laços), isso não impede que um único laço trate esta variável como um vetor cujo único índice está no intervalo fechado [1; **m*n**]. Se o usuário fornecer 3 e 2 como entrada para este algoritmo a saída será:

```
3 2 <ENTER>
1 2 2 4 3 6
```

Síntese do capítulo

Neste capítulo foi apresentada a estrutura de matriz a qual é utilizada para manipulação de variáveis com capacidade de armazenar na memória mais de um valor ao mesmo tempo. Matrizes unidimensionais são comumente denominadas de **vetores**. Os vetores representam a forma de matriz mais comumente empregada em algoritmos.

Atividades de avaliação

1. Construa algoritmo que receba um vetor de inteiros e imprima como saída o maior e o menor valor contidos nele.
2. Construa algoritmo que busque num vetor os dois maiores valores contidos.
3. Construa um algoritmo que receba um vetor de comprimento conhecido e altere seus valores de forma a ficarem em ordem inversa.
4. Construir algoritmo que carregue em um vetor de comprimento n os primeiros n valores primos.
5. Rotação é o procedimento de deslocamento dos itens de um vetor para casas vizinhas. Numa rotação a direita todos os itens migram para posições logo a frente, exceto o último que vai para a posição 1, ex: se v = {1,2,3,4}, uma rotação a direita modifica v para {4,1,2,3}. De forma análoga uma rotação a esquerda de v geraria {2,3,4,1}. Construir algoritmos para os dois processos de rotação.
6. Construa um algoritmo que calcule a média dos números menores que 5000 que sejam divisíveis por 7 mas não sejam divisíveis por 3.
7. Construa um algoritmo que receba uma lista de números inteiros e rearranje seus elementos de forma a ficarem em ordem crescente
8. Uma cadeia de caracteres é dito palíndromo se a seqüência dos caracteres da cadeia de esquerda para direita é igual à seqüência de caracteres da direita para esquerda. Exemplos: ARARA, RADAR, AKASAKA, ANA. Faça um algoritmo que reconheça se uma cadeia de caracteres é palíndromo.
9. O número 585 é palíndromo nas bases 10 e base 2 (1001001001). Construir algoritmo que determine a soma de todos os números com esta propriedade que sejam menores que um milhão.

Leituras, filmes e sites

http://www.academicearth.org/lectures/array

http://www.academicearth.org/lectures/multi-dimensional-arrays

Referências

SEBESTA, Robert W. **Concepts of Languagues Programming**. 8th edition. University of Colorado. Addison-Wesley 2007. 752p. ISBN-10: 032149362.

Capítulo 4

Programação modular

Objetivos

- Apresentar o conceito de programação modular, isto é, como dividir um programa em subprogramas.

1. Modularização

1.1. O que é modularizar?

Há dois importantes níveis de abstração em computação: abstração de dados e abstração de processos. A abstração de dados explora como diferentes tipos de informações podem ser representados e ainda quais as restrições de operabilidade entre eles. A abstração de processos trata da simplificação do processo de execução de uma dada aplicação pela divisão deste em partes menores (sub-processos) que podem ser testadas separadamente.

A **modularização** está estreitamente ligada à abstração de processos. Modularizar significa dividir um programa em subprogramas cujos funcionamentos podem ser testados separadamente. Os subprogramas são nada mais que abstrações com alguma sintaxe adicional (em nível de código).

1.2. Funções, procedimentos, processos e controle

Modularizar algoritmos significa separar o pseudocódigo em **funções**. As funções, módulos ou subprogramas, como são conhecidos, podem aparecer em grande quantidade em um mesmo algoritmo, mas apenas uma destas funções tem papel de acionador, ou seja, sempre a partir dela se iniciará a execução. É a chamada **função principal**. Todas as demais funções são secundárias.

Cada função tem um nome e, num mesmo algoritmo, todas as funções devem ter nomes distintos entre si. Além do nome uma função pode conter parâmetros. **Parâmetros**, ou **argumentos**, são variáveis especiais que agem como entrada de dados para estas funções. A sintaxe geral de uma função é a seguinte:

```
<nome _ da _ função>  (<arg _ 1>:  <tipo _ 1>,  <arg _ 2>:  <tipo _ 2>,
..., <arg _ n>: <tipo _ n>)
  //declaração de variáveis da função
  //instruções da função
```

Na sintaxe de funções os argumentos são arranjados em uma lista de variáveis com seus respectivos tipos e ainda separados por vírgula. Apesar de não existir a palavra **declarar**, os argumentos de entrada são, por definição, variáveis ali declaradas. A primeira linha da função (composta por seu nome e seus argumentos entre parênteses) é chamada de **cabeçalho da função**. O **corpo da função** é todo o código abaixo do cabeçalho e cuja indentação é pelo menos um nível acima (relação de encapsulamento de código em funções). O corpo de uma função pode possuir uma sessão declarativa (variáveis da função) e possui obrigatoriamente (logo abaixo) a sessão de código (bloco de código).

Se um grupo de argumentos de entrada tiver mesmo tipo eles podem ser declarados com a seguinte sintaxe:

```
<nome _ da _ função>       (<arg _ 1>,<arg _ 2>,<arg _ 3>,...,<arg _ n>:
<tipo>)
  // corpo da função
```

Para que seja executada, uma função precisa ser **chamada**. Uma chamada de função é realizada pela invocação de seu nome juntamente com valores para seus parâmetros. Para executar uma dada função secundária, ela deve ser chamada de dentro da função principal ou então de dentro de outra função secundária. Cada função de um algoritmo deve ser chamada pelo menos uma vez em algum lugar daquele algoritmo. Se uma dada função nunca é chamada ela pode ser removida do projeto. A execução de uma aplicação é nada mais que a execução de sua função principal.

Quando uma aplicação inicia sua execução diz-se que a função principal recebe o **controle de fluxo** ou simplesmente que recebe o controle. Se a função principal chama uma função secundária então esta função receberá o controle do fluxo enquanto estiver executando e o devolverá a função principal quando sua execução se encerrar.

De modo geral quando uma função A, seja ela principal ou não, chama outra função B ocorre repassagem provisória do controle de fluxo de A para B e depois seu retorno para A quando B se encerra. Sistematicamente o controle migra de função para função durante toda execução. Em algum momento cada uma das funções do projeto manterá o controle. Se uma mesma função for chamada várias vezes ela manterá o controle em momentos diferentes. Apenas uma, entre as várias funções, se existirem, terá o controle durante a execução da aplicação. (Uma aplicação deverá conter pelo menos a função principal).

Funções podem retornar valores. O **retorno** (ou saída) de uma função pode ser, por exemplo, um número, um caractere ou um booleano. A ação de retorno de uma função é sumária, ou seja, quando invocada a função automaticamente se encerra devolvendo o controle à função chamadora. Para provocar o retorno usa-se a palavra chave **retorne** seguida do valor que se deve retornar. Veja o exemplo a seguir:

```
soma (x, y: inteiro)
   declare
      s: inteiro
   s ← x + y
   retorne s

principal()
   declare
      a, b, c: inteiro
   leia a, b
   c ← soma(a,b)
   escreva c
```

Neste exemplo há duas funções: a principal (que sempre denotaremos pelo nome **principal**) e a função secundária **soma()**. A função secundária possui dois parâmetros, **x** e **y** e sua chamada é feita na função principal passado como argumentos os valores nas variáveis a e b. O uso da atribuição, **c ← soma(a, b)**, entra em concordância com o **retorne** da função **soma()**, ou seja, só é possível atribuir a **c** algum valor porque este valor existe e é o retorno da função em questão. As variáveis **x** e **y** são denominadas **argumentos formais** da função **soma()** ao passo que a e b são denominadas de **argumentos reais** da mesma função.

Os argumentos reais de uma função podem ser tanto constantes como variáveis. Já os argumentos formais aparecem no próprio corpo construtivo da função e são representados apenas por variáveis. Para cada argumento real deve haver um formal equivalente. A ordem em de disposição dos argumentos formais deve ser mantida pelos argumentos reais, por exemplo, se uma dada função possui três argumentos cujos tipos são respectivamente inteiro, real e booleano, então os argumentos reais repassados no momento da chamada devem também estar listados nesta ordem de tipo, ou seja, inteiro, real e booleano.

Consideremos outro exemplo:

```
e _ primo (n: inteiro)
   declare
      d: inteiro
   d ← 2
   enquanto (d*d ≤ n) faça
      se (n mod d = 0) então
         retorne F
      d ← d + 1
   retorne V

principal()
   declare
     k: inteiro
   para k ← 1 até 200 faça
      se ( e _ primo(k) ) então
         escreva k
```

A função **e_primo()** acima é uma versão modularizada do algoritmo apresentado no estudo de caso Números Primos. O objetivo desta função é receber um inteiro e retornar um booleano onde retorno Verdadeiro significa que a entrada é um valor primo e retorno Falso significa o inverso. Se durante os testes de divisibilidade internos ao laço **enquanto** algum dos restos for nulo, então não haverá mais sentido continuar testando outras divisões porque o número já não é mais primo.

A estratégia do estudo de caso anterior fora o uso de um **pare** e uma variável booleana auxiliar. Neste novo contexto se invoca apenas um, **retorne F**, porque este comando força o encerramento da função independente de onde ele apareça dentro da função. Como um **F** (Falso) segue o **retorne** então a função retorna convenientemente para valores não primos. Caso o valor seja primo o laço se encerrará normalmente e a instrução, **retorne F**, jamais será executada. Em contrapartida a continuação da execução encontrará a linha, **retorne V**, que forçará o encerramento e retorno Verdadeiro: de fato apenas valores primos causarão o alcance desta linha de instrução. A função principal faz uso intenso da função **e_primo()** testando quais números entre 1 e 200 são ou não primos. Caso sejam, eles são impressos.

A expressão booleana do **se** é a própria chamada da função **e_primo()** haja vista que o retorno dela é um valor booleano e desta forma não desobedece a sintaxe da estrutura.

Há funções que não precisam retornar. Elas são denominadas de **procedimentos**. Veja o exemplo a seguir:

```
print _ primo (t: inteiro)
   declare
      i: inteiro
   para i ← 1 até t faça
      se ( e _ primo(i) ) então
         escreva(i)
```

O procedimento anterior deve ser parte integrante do algoritmo apresentado anteriormente, pois faz uso da função **e_primo()**. Este procedimento simplesmente imprime na saída padrão todos os primos entre 1 e o valor no argumento de entrada **t**. Obviamente o conteúdo impresso pode ser interpretado como saída da função. Entretanto, por convenção, diremos que uma função possui saída se ela gerar alguma informação que puder ser repassada adiante por uma ação conjunta de um **retorne** e um operador de atribuição. A função principal a seguir imprime os primos menores que 100 pela chamada de **print_primos()**.

```
principal ()
   print _ primos(100)
```

Como **print_primos()** não possui saída não há nenhuma atribuição atrelada à chamada desta função.

1.3. Escopo de variáveis

O escopo de uma variável é definido como o lugar no algoritmo onde esta variável pode ser manipulada. De forma geral o escopo de todas as variáveis mostradas até este ponto é exatamente o corpo das funções onde são definidas. Assim uma variável, definida dentro de uma dada função, não será visível dentro de outras funções. Exemplo:

```
func _ 1 ()
   declare
      x: inteiro
   x ← 100

func _ 2 ()
   escreva x // erro: x não foi definido neste escopo
```

A variável **x** não foi definida no escopo da função **func_2()** e desta forma não pode ser impressa. Analogicamente duas variáveis em escopos diferentes podem ter o mesmo nome e não estarem relacionadas (posições distintas em memória). Exemplo:

```
func _ 1 ()
   declarar
      x: inteiro
      y: booleano
   // corpo da função func _ 1()

func _ 2 ()
   declarar
      x: real
      y: booleano
   // corpo da função func _ 2()
```

Neste exemplo as variáveis **x** e **y** estão em escopos diferentes. Em cada escopo **x** tem um tipo diferente, mas **y** mantém o mesmo tipo (booleano). Tais variáveis só possuem em comum o nome. Operar **x** e **y** em **func_1()** não afeta **x** e **y** em **func_2()** e vice=versa.

Os parâmetros ou argumentos de uma função são as únicas variáveis que podem ser alteradas a partir de outro escopo. Por exemplo, se **x** é argumento formal da função **f()** então **x** está no escopo de **f()**; entretanto ao fazer uma chamada de **f()**, uma constante ou variável fora do escopo de **f()** terá que agir como argumento real para carregar **x**. Argumentos formais são portas de entrada para as funções e devem se comportar desta forma.

Tanto os argumentos formais quanto as variáveis declaradas internamente a uma função estão no escopo desta função e são comumente chamadas de **variáveis locais**. Todas as variáveis apresentadas nos exemplos até aqui são variáveis locais. Uma variável não local é denominada **variável global**. Uma variável global é aquela visível a todas as funções de um projeto de algoritmo. Para criar variáveis globais uma seção de declaração externa a todas as funções deve ser criada como no exemplo seguinte:

```
declare
   x: inteiro

proc ()
   escrever x

principal()
    x ← 23
    proc()
```

O projeto acima possui uma variável global, **x,** um procedimento **proc()** sem entrada (o par de parentes vazio é mantido aqui para indicar que se trata de um módulo) e a função principal. O módulo **proc()** pode escrever **x** porque ela é global ao passo que **principal()** também pode mudar **x** com uma atribuição direta pela mesma razão. Visto que **proc()** é chamado em **principal()** então o valor 23 será impresso na saída padrão.

Caso exista alguma variável local em alguma função que tenha mesmo nome de uma variável global a ambigüidade será tratada de forma análoga à maioria das linguagens de programação: valerá a variável local e não será possível o acesso a variável global. Veja o exemplo:

```
declare
   x: inteiro

proc ()
   declare
      x: inteiro
   x ← 23

principal()
   x ← 35
    proc()
    escreva x
```

O valor impresso na saída padrão será 35 porque a variável **x** mudada com a execução de **proc()** não é a versão global. Se a declaração de **x** em **proc()** fosse removida o valor impresso na saída padrão seria 23.

1.4. Passando argumentos por valor e por referência

As regras de escopo permitem que argumentos formais sejam modificados para os valores de seus respectivos argumentos reais no momento da chamada da função. De fato cada argumento formal recebe uma **cópia** do conteúdo de seu argumento real equivalente. Este modelo é conhecido como **passagem de argumento por valor**. Utilizando tal modelo, o argumento formal não tem qualquer influência sobre o argumento real o que em alguns casos não é interessante. Suponhamos, por exemplo, uma função **inc()** que contenha apenas um argumento e cujo objetivo é aumentar em uma unidade o valor no argumento real. Antes de escrever o código para a função **inc()** codifiquemos uma função principal que teste **inc()**, como segue:

```
principal()
   declare
      j: inteiro
   j ← 5
   inc(j)
   escreva j // deve imprimir 6
```

O comportamento esperado no código acima é a impressão do valor 6 na saída padrão. Entretanto, pelo modelo da passagem por valor tal função não pode ser escrita. Vejamos uma tentativa para **inc()**:

```
inc(k: inteiro)
   k ← k + 1
```

Caso esta função seja utilizada o valor impresso na saída padrão será 5 e não 6 simplesmente por que quem incremente em uma unidade é **k** e não **j**. Como **k** e **j** não se relacionam a função não se comporta como desejado.

Para resolver o problema apresentamos outro modelo de passagem de argumento conhecido como **passagem por referência**. Uma referência por definição é um apelido de variável que pode ser manipulada como se fosse a própria variável que ela apelidou. Por exemplo, se **r** é uma referência da variável **x** então uma atribuição a **r** será de fato uma atribuição para **x**. Na passagem por referência alguns ou todos os argumentos formais de uma função são transformados em referências: assim quando os argumentos reais são repassados não ocorre cópia alguma e sim a definição de apelidos em outro escopo! Uma vez de posse de tais referências a função pode mudar variáveis fora de seu escopo.

Para transformar um argumento formal numa referência basta preceder a declaração da variável com a palavra chave **ref**. Assim a função **inc()** pode ser agora corretamente escrita:

```
inc(ref k: inteiro)
   k ← k + 1

principal()
   declare
      j: inteiro
   j ← 5
   inc(j)
   escreva j
```

Nesta nova versão **k** deixa de ser uma variável de tipo inteiro e torna-se uma referência para valor inteiro. Assim quando **j** é repassado como argumento, então **k** torna-se provisoriamente uma referência para ele. A adição de uma unidade à referencia **k** na função inc() afeta de fato a variável **j**. O valor impresso na função principal é então 6.

1.5. Passando vetores como argumentos

Algumas linguagens de programação não permitem que vetores sejam repassados por valor. Isso acontece porque não é possível garantir que o vetor repassado sempre tenha o mesmo comprimento do vetor argumento de entrada. Mesmo que seja possível impedir a execução de uma função quando se tentar repassar a ela um vetor de comprimento inválido, isso implicaria em restrições de uso desta função. A saída utilizada nestas linguagens é o uso de referências. Adotaremos estas mesmas regras nos algoritmos que seguem.

Para utilizar referências na repassagem de vetores seguiremos a seguinte sintaxe geral:

```
<nome _ da _ função>(ref <arg>[]: <tipo>)
   //corpo da função
```

Apesar de um par de colchetes vazios, o argumento não se trata de um vetor dinâmico. A associação entre **ref** e **[ ]** no argumento de entrada (e apenas aí) significa vetor repassado por referência.

Quando repassado por referência um vetor torna-se modificável no escopo da função. Entretanto a função aparentemente perde uma informação importante: o comprimento do vetor. Para resolver o problema utilizaremos uma importante função chamada **comp()** que recebe como argumento um vetor, ou referencia de vetor, e retorna seu comprimento. Exemplo:

```
soma(ref M[]: inteiro)
   declare
      i, res: inteiro
   res ← 0
   para i ← 1 até comp(M) faça
      res ← res + M[i]
   retorne res
```

A função **soma()** recebe um vetor de valores inteiros e retorna a somatória de seus elementos. A variável **res** é um acumulador que recebe zero no princípio e depois muda em cada iteração do laço até conter a soma de todos os elementos. A função **comp()** retorna o comprimento do vetor recebendo-o como argumento (sem colchetes).

Estudo de Caso – Números Pandigitais. A modularização do problema dos pandigitais fornece uma solução mais natural como indica o algoritmo seguinte. Nesta nova abordagem procura-se determinar quantos números pandigitais existem abaixo de um milhão.

```
ndig(num: inteiro)
   declare
      s: inteiro
   s ← 0
   enquanto (num>0) faça
      s ← s + 1
      num ← num div 10
   retorne s

carregar(ref R[], num: inteiro)
   declarar
      i: inteiro
   para i ← comp(R), 1, -1 faça
      R[i] ← num div 10
      num ← num div 10
```

```
e _ pandig(ref R[]: inteiro)
   declare
      i, j: inteiro
   para i ← 1 até comp(R)-1 faça
      para j ← i+1 até comp(R) faça
         se (R[i] = R[j]) então
            retorne F
   retorne V

principal()
  declare
     cnt, Q[], t, n: inteiro
  cnt ← 0
  para n ← 0 até 1000000 faça
     alocar Q[ ndig(n) ]
     carregar(Q, n)
     se ( e _ pandig(Q) ) então
        cnt ← cnt + 1
     limpar Q
  escreva cnt
```

O algoritmo acima possui três funções secundárias e a função principal. Na função principal o laço **para** testa cada valor que o contador **n** assume. Cada teste consiste em: (I) alocar dinamicamente o vetor **Q** com comprimento igual à quantidade de dígitos do atual valor de **n**; (II) carregar **Q** com os dígitos de **n** e; (III) testar se o vetor **Q** possui elementos repetidos; caso seja o contador **cnt** (iniciado com zero fora do laço) é incrementado em uma unidade. Quando o laço se encerra o valor em **cnt** é o total de pandigitais menores que um milhão.

As quantidades de dígitos de **n** em cada iteração são calculadas por chamadas a **ndig()** cujo modelo de passagem de seu único argumento é por valor (por isso a inexistência de uma variável auxiliar **j** como no estudo de caso anterior sobre pandigitais).

O procedimento **carregar()** distribui os dígitos do valor em num (segundo argumento) dentro do vetor referenciado por **R** (primeiro argumento). Quando invocada dentro do laço para na função principal, esta função configura **Q** para conter os dígitos do atual valor em **n**.

A função **e_pandig()** testa se um dado vetor referenciado por seu único argumento de entrada, **R** (referência a vetor em outro escopo), possui repe-

tição de elementos. Quando isso acontece o par de laços aninhado são encerrados por um, **retorne F**; do contrário a instrução, **retorne V**, é alcançada. Assim, apenas números pandigitais distribuídos no vetor referenciado por **R**, farão **e_pandig()** retornar Verdadeiro. Como a saída desta função é um booleano então ela é usada como expressão de decisão no **se** da função principal.

1.6. Recursividade

Recursividade é a técnica que permite a uma função (ou procedimento) fazer, no corpo de seu código, uma chamada a si própria. Para ilustrar a recursividade apresentamos a seguir duas versões de funções para cálculo de fatorial: uma não-recursiva (iterativa) e outra recursiva.

```
//versão iterativa
fat(n: inteiro)
   declare
      res: inteiro
   enquanto (n>0) faça
      res ← res * n
      n ←n - 1
   retorne res

// versão recursiva
fat(n: inteiro)
   se (n<2) então
      retorne 1
   senão
      retorne n*fat(n-1)
```

A versão não recursiva não trás novidades, mas ilustra a potencialidade dos laços para resolver problemas como o cálculo de fatoriais. Na versão recursiva o laço desaparece, pelo menos aparentemente. O processamento da versão recursiva da função fatorial é descrito a seguir. Quando uma chamada explícita a **fat()** é feita seu argumento formal recebe o valor do qual se deseja calcular o fatorial. Caso este seja menor que 2 a função se encerra retornando 1. Caso não seja, ela tenta retornar o produto, **n*fat(n-1)**.

Como a expressão de retorno contém a própria função, então uma nova chamada a **fat()** é realizada. Enquanto esta nova chamada não for resolvida a chamada anterior não poderá retornar. Assim a primeira chamada fica a espera do retorno da segunda chamada. Essa por sua vez pode desencadear o

mesmo processo e ter-se-á um primeiro aguardando por um segundo e esse por sua vez por um terceiro. O processo só não se torna infinito porque existe a decisão bidirecional, se, que desvia para retorno 1 sempre que valores menores que 2 forem repassados.

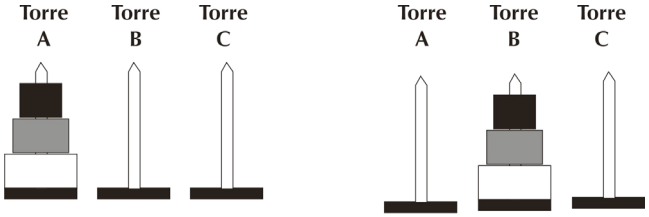
Em suma a recursão cria vários níveis de espera que são resolvidos de trás para frente no momento em que uma delas não recorre mais a recursão. A presença de um mecanismo de parada como o descrito no exemplo do fatorial (uso de se) impede naturalmente o problema de laço recursivo infinito.

Um outro exemplo clássico que demonstra o uso da recursão é o problema conhecido como Torre de Hanoi. Nesse problema, deseja-se mover um conjunto de n discos, de tamanhos diferentes, de uma torre para outra, na presença de uma terceira torre auxiliar, respeitando-se algumas restrições, entre as quais:

I. Somente um disco pode ser movido por vez;

II. Em nenhum momento, um disco de tamanho maior pode estar acima de um de tamanho menos.

Inicialmente, como mostra a figura XX, todos os discos encontram-se empilhados em uma das torres, digamos, A, em ordem decrescente de tamanho. A partir de movimentos válidos, ou seja, movimentos que respeitem as restrições acima, deseja-se empilhar todos esse discos em uma outra torre, digamos, B. Para isso, pode-se usar a torre extra, C, como auxiliar.



Configuração Inicial

Configuração Final

Intuitivamente, vamos discutir de que forma esse problema pode ser resolvido. Para mover todos os discos da torre A para a torre B, sabemos que precisamos inicialmente colocar o disco maior (nesse caso, o disco branco) na base de B, visto que a restrição II exige que os discos maiores fiquem na base das torres. No entanto, para movermos o disco maior de A, precisamos remover todos os discos que estão acima dele (discos cinza e preto).

Como vamos mover esse disco maior para B, essa torre não poderá conter nenhum outro disco. Sendo assim, precisamos mover todos os discos de A, com exceção do maior, da torre A para a torre C, para podermos pos-

teriormente mover o disco maior de A para B. Depois disso, "basta" mover os discos que estão em C para B. Veja que, para movermos os discos de A para C, estamos resolvendo o mesmo problema original da Torre de Hanoi, dessa vez tendo como origem a torre A e como destino a torre C, e desconsiderando o disco maior. Logo, podemos usar essa característica pra gerar uma solução recursiva, como se segue:

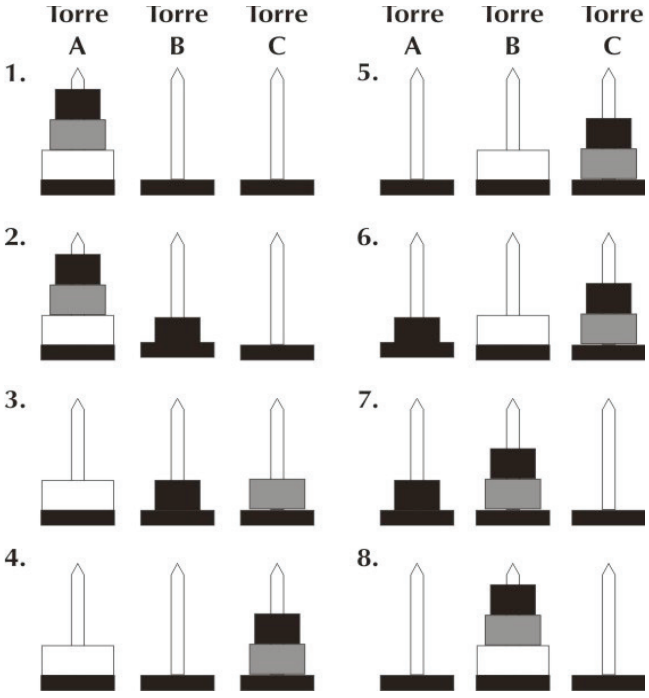
```
hanoi(n: inteiro, A, B, C: <tipo _ torre>)
   se (n>0) então
    hanoi(n – 1, A, C, B);
                move disco de A para B
          hanoi(n – 1, C, B, A);
```

O algoritmo recursivo acima apresenta um procedimento, chamado hanoi(), que move os discos, de acordo as restrições apresentadas anteriormente. Esse procedimento move n discos, da torre A para a torre B, usando a torre C como auxiliar. Como na descrição intuitiva mostrada acima, o algoritmo recursivo da Torre de Hanoi resolve esse problema em três etapas: movendo, recursivamente, n-1 discos da torre A para C, usando B como auxiliar; movendo o disco restante de A para B e movendo, também recursivamente, os n-1 discos da torre C para a torre B.

Abaixo segue um diagrama com a aplicação do algoritmo recursivo mostrado acima para o problema da Torre de Hanoi com 3 discos. Nesse exemplo, para movermos os discos da torre A para B, primeiro precisamos mover o disco branco (nessa caso, o maior) para B. Mas antes disso, vamos mover os dois discos acima dele (cinza e preto) para C.

Para isso, usaremos agora a torre B como auxiliar para executar o mesmo algoritmo recursivo. Logo, precisaremos mover o disco cinza para C. Mas antes temos que mover o disco preto para a torre auxiliar B. Fazendo isso, como mostrado no passo 2, podemos mover o disco cinza para C e o disco preto também para C (passos 3 e 4). Pronto, já temos o disco cinza livre para movermos para B (passo 5). Repetindo-se o procedimento, temos como mover os discos de C para B (passos 6 – 8).

Analogamente, o mesmo procedimento recursivo pode ser usado para resolver o problema com mais de 3 discos.



A recursão se apresenta como técnica de projeto de algoritmos com muitas aplicações. Notadamente, a recursão tem sido usada como estratégia na geração de soluções para o problema da pesquisa, como é o caso do algoritmo de busca binária, para o problema de ordenação, com os algoritmos *quicksort* e intercalação, em problemas de grafos, e muitos outros problemas computacionais.

Síntese do capítulo

Neste capítulo foi apresentado o conceito de modularização. Este termo é muito utilizado na programação, aliado ao método de dividir para conquistar, o qual permite dividir um problema em partes menores. Modularizar significa dividir um programa em subprogramas cujos funcionamentos podem ser testados separadamente.

As funções, módulos ou subprogramas, como são conhecidos, podem aparecer em grande quantidade em um mesmo algoritmo, mas apenas uma destas funções tem papel de acionador, ou seja, sempre a partir dela se iniciará a execução. É a chamada função principal. Todas as demais funções são secundárias. Finalizando, é abordado o conceito de recursividade: técnica que permite a uma função fazer, no corpo de seu código, uma chamada a si própria.

Atividades de avaliação

1. Faça uma função que recebe um número inteiro por parâmetro e retorna verdadeiro se ele for par e falso se for ímpar.
2. Escreva um procedimento de nome tipo_triangulo que receba 3 parâmetros representando os lados de um triângulo e imprima o tipo dele (eqüilátero, isósceles ou escaleno).
3. Escreva um algoritmo em pseudocódigo para receber o tamanho dos 3 lados de um triângulo e usar o procedimento tipo_triangulo para exibir o tipo dele.
4. Escreva um algoritmo para receber 10 números reais e armazená-los em um vetor. Depois disso, mostre o somatório dos números, através do uso da função somatorio, que não recebe parâmetro nenhum, acessa o vetor definido globalmente e retorna o somatório dos elementos do vetor.

Leituras, filmes e sites

http://www.lsd.ic.unicamp.br/projetos/e-lane/introPascal/aula9.html

http://www.inf.pucrs.br/~pinho/LaproI/Funcoes/AulaDeFuncoes.htm

Referências

SEBESTA, Robert W. **Concepts of Languagues Programming**. 8th edition. University of Colorado. Addison-Wesley 2007. 752p. ISBN-10: 032149362.

Sobre os autores

Ricardo Reis Pereira: Possui graduação em Engenharia Mecânica pela Universidade Federal do Ceará (2000) e mestrado em Programa de Mestrado em Engenharia Química pela Universidade Federal do Ceará (2004). Atualmente é professor assistente do curso Sistemas de Informação do Campus de Quixadá da Universidade Federal do Ceará. Ministra as disciplinas de fundamentos de programação, cálculo diferencial e integral, estruturas de dados e computação gráfica e atua nas áreas de algoritmos, computação gráfica e mais recentemente em visão computacional.

Jerffeson Teixeira de Souza: recebeu o título de Ph.D. em Ciência da Computação pela *School of Information Technology and Engineering* (SITE) da *University of Ottawa*, Canadá. Éé professor adjunto da Universidade Estadual do Ceará (UECE) e Pró-reitor de Pós-graduação e Pesquisa da UECE, já tendo trabalhado como Diretor de Pesquisa da UECE, Coordenador do Mestrado Acadêmico em Ciência da Computação da UECE (MACC) e Coordenador Geral do Mestrado Integrado Profissional em Computação Aplicada UECE/IFCE (MPCOMP). Atualmente é Coordenador dos Grupos de Otimização em Engenharia de Software da UECE (GOES.UECE) e Padrões de Software da UECE (GPS.UECE) e Coordenador da Especialização em Engenharia de Software com Ênfase em Padrões de Software da UECE (EES). É ainda membro eleito do *Steering Commitee do International Symposium on Search Based Software Engineering* e membro nominado do *Hillside Group*. Trabalhou ainda como Co-presidente da Quarta Conferência Latino-americana em Linguagens de Padrões para Programação (SugarLoafPLoP 2004), como Co-presidente do Comitê de Programa do SugarLoafPLoP 2007, como Presidente do SugarLoafPLoP 2008 e é atualmente Co-presidente do Comitê de Programa do 4th *International Symposium on Search Based Software Engineering* (SSBSE´2012). É um dos precursores da área de *Search Based Software Engineering* no Brasil e um dos principais pesquisadores dessa área no país, tendo sido ainda o idealizador e Coordenador Geral do I Workshop Brasileiro de Otimização em Engenharia de Software (WOES), que foi posteriormente nomeado para Workshop Brasileiro Engenharia de Software baseada em Busca (WESB). Seus interesses de pesquisa são: Otimização em Engenharia de Software, Documentação e Aplicação de Padrões de Software e Estudo de Técnicas e Aplicação de Algoritmos de Mineração de Dados.

Jeandro de Mesquita Bezerra: Possui graduação em Computação pela Universidade Estadual do Ceará (2003) e mestrado em Computação Aplicada MPComp pela Universidade Estadual do Ceará (2007). Atualmente é professor Assistente II da Universidade Federal do Ceará (Campus de Quixadá).

Tem experiência na área de Ciência da Computação, com ênfase em Redes de Computadores, atuando principalmente nos seguintes temas: Redes Sem Fio, Avaliação de Desempenho e Computação em Nuvem.