Construção de Algoritmos

Jander Moreira
DC-UFSCar

Sumário

Lista de figuras

Lista de algoritmos

Lista de tabelas

Unidade 1 - Conceitos básicos sobre algoritmos

1.1 Primeiras palavras

Os alunos que chegam a um curso universitário possuem, em geral, formação relativamente heterogênea. Em particular, no que tange a algoritmos e programação, muitos nunca tiveram contato com programas ou mesmo com computadores diretamente; alguns conhecem o computador como usuários eventuais, conhecem a Internet, já usaram os mensageiros instantâneos e salas de bate-papo; outros, ainda, por curiosidade própria ou por terem feito um curso técnico, já programaram e adquiriram uma noção e conhecimento do que é programar e de como desenvolver programas.

Esta disciplina é voltada a todos os alunos e, portanto, cada aluno utilizará o material apresentado de forma diferente.

Para começar, é importante delimitar os objetivos da disciplina. Um primeiro ponto a destacar aponta para aquilo que a disciplina **não é**: esta não é uma disciplina que ensina a programar. Seu objetivo é um pouco mais independente dos programas de computador, embora esteja claramente relacionado a eles. A disciplina **Construção de algoritmos** visa orientar o aluno a resolver um determinado problema usando ferramental apropriado e propondo uma solução que, no final, poderá ser codificada em uma linguagem de programação e se tornar um programa de verdade.

Assim, pretende-se delinear uma metodologia de como abordar um problema, como entendê-lo e como propor uma solução algorítmica de qualidade que resolva o problema proposto.

1.2 Algoritmos no cotidiano

Para iniciar, é preciso que se entenda o que é um algoritmo. Genericamente, pode-se ver um algoritmo como uma coleção de passos que orientam as ações para se chegar a um resultado. Por exemplo, ao chegar de uma viagem à praia, o motorista fica curioso (ou ansioso) para saber o consumo médio do automóvel. Como foi cuidadoso, lembrou-se de anotar todas as vezes que abasteceu, além de, ao partir, zerar o hodômetro parcial ao encher o tanque. De tanque cheio novamente, pede ao filho caçula que faça as contas para ele. Com os papéis na mão, o filho os olha e depois encara o pai, com óbvia cara de dúvida. A orientação do pai é a seguinte: "some os valores dos litros de combustível e veja o total; depois, veja o quanto andamos olhando o número indicado no hodômetro; divida a distância total pela quantidade total de litros; faça a conta, que é o consumo médio, e me diga quanto deu". Esta orientação é um algoritmo, pois define claramente quais os passos que o filho deve seguir para gerar o resultado esperado (ou seja, a solução para o problema).

Da mesma forma, outros tipos algoritmos se apresentam no dia-a-dia, como, por exemplo, na parte do manual de um telefone celular que ensina a cadastrar um novo contato na agenda, na receita que orienta em como fazer um brigadeiro ou mesmo nas instruções que alguém solicita na rua quando está perdido.

1.3 Conceitos básicos de algoritmos e sua lógica

1.3.1 Delimitações necessárias aos algoritmos formais

A noção de um algoritmo é conhecida por todos, mesmo que desconheçam que tais instruções possam ser chamadas de algoritmo. Entretanto, alguns pontos devem ser destacados, pois são importantes para que programas com objetivos computacionais sejam escritos.

O primeiro ponto é a definição do problema. De volta ao exemplo da viagem à praia, o problema em questão poderia ser colocado da seguinte forma: "dados os valores, em litros, de combustível usado nos diversos abastecimentos feitos em uma viagem, juntamente com o total de quilômetros percorrido, calcule e reporte o consumo médio". (É claro que, nestas palavras, fica um pouco menos poético, mas o problema é o mesmo.) Deve ser observado que o problema é bastante específico: já se tem o total de quilômetros percorridos e os valores individuais de todos os abastecimentos, dados em litros. O problema seria diferente caso se tivesse a quilometragem percorrida por partes, entre cada um dos abastecimentos. Também seria diferente se o custo dos abastecimentos fosse envolvido. Entender qual é o problema permite propor uma solução adequada. Assim, se fosse preciso somar as distâncias dos diversos trechos para se obter a distância total, a orientação dada pelo pai não seria adequada, pois, seguida à risca, o filho dividiria somente o último trecho (indicado no hodômetro) pelo total de litros. Ou seja, a solução deveria ser outra, pois o problema é diferente.

A solução proposta pelo pai é clara o suficiente para que o filho, que já está em idade escolar, faça as contas e produza o resultado. Se o filho fosse mais novo, talvez tivesse grandes dificuldades para seguir as orientações. Poderia ter dúvidas, como, por exemplo, "pai, o que é dividir?". Um algoritmo deve ter claro quem (ou o quê) o executará, sendo que as instruções devem ter clareza suficiente e necessária para se chegar ao final sem duvidar que o procedimento foi feito corretamente. Para um filho mais velho, a orientação do pai talvez fosse, simplesmente, "calcule o consumo; as anotações estão no porta-luvas".

O uso do imperativo é um indicativo que cada instrução é uma ordem. Assim, verbos como *calcule*, *some*, *multiplique* devem ser usados.

Outro ponto, também muito importante, é a clareza de cada instrução em si. Para isso, pensem na avó, que gosta de cozinhar doces e quer ensinar o neto a fazer um brigadeiro. Em algum momento, a panela já no fogo, a orientação é "deixe ferver até dar ponto". Como muitos, o neto deve ter pensado "dar ponto?!? Como eu posso saber que *deu ponto*?". Portanto, em um algoritmo mais formal, a instrução de cada passo deve ser precisa e, seguida à risca, deve produzir o mesmo resultado. Boa sorte com o brigadeiro.

1.3.2 Ordem de execução

Uma característica básica de um algoritmo, embora um tanto óbvia, deve ser destacada. A ordem em que os passos são seguidos é importante. Para ilustrar isso, o Algoritmo 1-1 apresenta o cálculo do consumo médio.

Um formato um pouco mais formal já é usado, mas não se deve preocupar com isso no momento. Além disso, a numeração das linhas é feita apenas para que cada comando possa ser referenciado no texto e, assim, não faz parte dos algoritmos.

Algoritmo 1-1

```
1  { cálculo do consumo médio de um veículo, dados a distância total e
2    os valores de litros consumidos em cada abastecimento realizado }
3
4  algoritmo
5        calcule o total de litros perguntando por cada valor e somando-os
6        pergunte pela distância total percorrida
7        calcule a divisão da distância total pelo total de litros
8        apresente o resultado do cálculo
9  fim-algoritmo
```

Não é difícil ver que, caso a ordem de algumas instruções fossem alteradas, o resultado não se modificaria. Este é o caso de inverter as linhas 5 e 6 do algoritmo. Porém, a alteração de outras linhas produziria um resultado incorreto, como seria o caso de trocar as linhas 6 e 7, ou seja, tentar calcular a divisão antes de se ter o valor da quilometragem.

Para os algoritmos formais utilizados nesta disciplina, a regra de execução de cada instrução vem descrita pelos itens:

1) As instruções são executadas seqüencialmente e na ordem em que estão apresentadas;
2) A próxima instrução somente é executada quando a anterior tiver terminado.

Considere o Algoritmo 1-2. Leia-o com atenção e veja se o resultado desejado (o consumo médio) pode ser corretamente obtido a partir dele.

Algoritmo 1-2

```
1  { cálculo do consumo médio de um veículo, dados a distância total e
2    os valores de litros consumidos em cada abastecimento realizado }
3
4  algoritmo
5        pergunte pela distância total percorrida
6        calcule a divisão da distância total pelo total de litros
7        calcule o total de litros perguntando por cada valor e somando
8        apresente o resultado do cálculo
9  fim-algoritmo
```

Deve-se notar que, se as instruções forem seguidas por uma pessoa, o resultado desejado pode até ser calculado corretamente. Não seria difícil para alguém ler a instrução da linha 7 antes de terminar a da linha 6, deduzindo que o total de litros é obtido pela soma das quantias individuais. Seria como se as linhas 6 e 7 fossem executadas simultaneamente. Neste caso, a linha 7 funciona quase que como uma observação para a linha 6.

Segundo as regras de execução apresentadas, o algoritmo seguinte não é executado corretamente. A precisão da linguagem adotada é importante, pois não há margens para deduções ou interpretações pessoais. Deve-se sempre ser lembrado que um algoritmo é um conjunto preciso de instruções, e não um conjunto de boas intenções.

Em particular, o termo **lógica de programação** é usado para indicar que a seqüência de ações que são definidas para resolver um dado problema segue um conceito geral de causa e efeito. Resultados produzidos em um passo podem ser aproveitados nos passos subseqüentes, todos cooperando para produzir o resultado final esperado.

1.3.3 Definição para algoritmos

Como visto, os algoritmos existem em formas e características variadas, desde em manuais mais ou menos rígidos até em informais receitas culinárias. Para esta disciplina, entretanto, o termo **algoritmo** terá um uso bastante restrito, sendo preciso, portanto, estabelecer seus limites.

Definição: *Um algoritmo é um conjunto finito de instruções que, se executadas, permitem a manipulação de um conjunto finito de dados de entrada para produzir um conjunto finito de dados de saída, dentro de um tempo finito.*

Desta definição, destaca-se inicialmente o termo **finito**. Os algoritmos devem ter um tamanho limitado, ou seja, ser composto por um número finito de instruções. Também a quantidade de dados que está disponível assim como a quantidade de dados produzida como resposta devem ser limitadas. Note que se pode desconhecer, em princípio, qual o volume de dados, mas esta quantidade não é infinita. Da mesma forma, mesmo demorando, um algoritmo produzirá um resultado.

Outra questão é a associação desta definição com os problemas que devem ser resolvidos. Os chamados dados de entrada são os dados que estão disponíveis para serem usados. A função do algoritmo é, a partir destes dados e de manipulações adequadas, produzir um resultado (dados de saída) que corresponda à resposta do problema proposto.

1.4 Apresentação visual dos algoritmos

Adicionalmente ao que já foi apresentado, ainda há uma questão de organização. Como se deseja que os algoritmos sejam claros e precisos, sua organização visual é essencial. Portanto, será exigido que algumas regras básicas sejam seguidas, embora algumas delas possam ser contornadas, pois podem remeter a estilos pessoais de organização.

O básico destas regras inclui:

1) Todo algoritmo escrito deve conter uma documentação, que é um comentário (escrito sempre entre chaves) em seu início, indicando sua finalidade e fazendo uma breve descrição do problema que o algoritmo deve resolver; se este algoritmo for ser enviado a outra pessoa, deve também conter autor(es) e data;

2) Linhas em branco devem sempre ser inseridas para separar grupos de instruções relacionadas, como será utilizado nos exemplos deste texto e apresentados no AVA;
3) Cada instrução deve ocupar uma linha diferente; caso não haja espaço na linha para especificá-la, pode ser continuada na linha de baixo, mas com tabulação maior, para indicar que é uma continuação;
4) O uso da tabulação para deixar claros os limites dos comandos deve ser sempre empregado; este é um dos itens "não opcionais";
5) Comentários (entre chaves) deverão ser inseridos quando necessário, para deixar claros os objetivos dos trechos do algoritmo; usado em algoritmos mais longos, são essenciais para a clareza e devem sempre obedecer à organização visual dos demais comandos;
6) Opcionalmente alguns comentários podem ser inseridos à direita, na mesma linha das instruções, mas apenas quando forem curtos e se esclarecerem uma questão específica daquela instrução em particular.

O Algoritmo 1-3 é um algoritmo "fictício", ou seja, que não propõe nenhuma solução, mas apenas ilustra alguns comandos (em modo informal), a documentação, comentários e outros elementos importantes da organização visual.

Algoritmo 1-3

```
1  { esta é descrição de um algoritmo; ela relata seu propósito, o qual,
2    no caso, é apenas ilustrar esta própria documentação e também as
3    regras de organização visual para os algoritmos
4
5    Jander Moreira
6    Novembro de 2007 }
7
8  algoritmo
9        { este comentário é para a parte inicial }
10       apresente mensagem instruindo quais dados devem ser fornecidos
11       pergunte pelos dados principais
12
13       { este comentário é para o primeiro processamento }
14       calcule algumas coisas com os dados principais, gerando dados
15                 secundários    { note aqui a continuação em outra linha }
16                                { note, também, comentários à direita }
17       apresente alguns resultados preliminares calculados
18
19       { comentário para outra parte do processamento }
20       calcule mais algumas coisas
21       se for necessário, pergunte por dados complementares
22
23       { este comentário é para a parte que apresenta os resultados,
24         mas como é longo, pode ser dividido em duas linhas }
25       apresente os resultados principais
```

```
26       se resultados adicionais foram também calculados, também devem
27                 ser apresentados
28
29       { comentário sobre o término }
30       apresente uma mensagem indicando o fim do processamento
31 fim-algoritmo
```

No algoritmo, as linhas 1 a 6 mostram a documentação básica, que descreve o propósito do algoritmo. As linhas 8 e 31 delimitam os comandos do algoritmo às linhas internas (de 9 a 30). Os diversos "blocos" de comandos, ou seja, conjuntos de comandos que delineiam ações específicas são separados por linhas em branco (linhas 12, 18, 22 e 28), sendo que cada um deles contém um comentário identificando sua função (linhas 9, 13, 19, 23 e 29). Deve ser notado também que todos os comandos em si são precedidos por uma tabulação, deixando-os mais à direita do par **algoritmo**/**fim-algoritmo** e mostrando que estão inclusos nestes delimitadores. Alguns comandos não cabem em uma única linha e são continuados na linha de baixo (linhas 15 e 16 e também 26 e 27), assim como o comentário que faz a descrição geral no início do algoritmo e o comentário que se inicia na linha 23 e termina na linha 24. Deve-se notar que sempre que há continuação na outra linha, a organização visual não é prejudicada, mantendo-se as tabulações necessárias. Outros comentários, ainda, sendo específicos a um comando, estão à direita, como é o caso dos situados nas linhas 15 e 16.

Cada um possui um estilo para a organização visual, o que admite que pequenas variações nas regras possam ser feitas. O mais importante é que, havendo uma visualização rápida do algoritmo, seja sempre possível observar claramente os comandos, suas continuações em outras linhas e os blocos, sem que seja necessário grande esforço para isso. Se mais do que poucos segundos forem necessários para saber se o comando usa mais que uma linha ou onde está o delimitador final de um comando, então a organização visual não está boa. O mesmo ocorre se não há clareza nas tabulações. Nestes casos, é preciso mudar a organização visual.

1.5 Considerações finais

Algoritmos, como observado, propõem soluções para problemas. Para fins computacionais, os algoritmos devem obedecer a um modo formal de apresentação e possuir uma organização visual relativamente rigorosa. O uso de comentários e da correta disposição visual dos comandos é um hábito e o estudante não deve se iludir com a idéia de que isso é uma "etapa posterior"; todos os aspectos visuais devem ser feitos durante o desenvolvimento e não após. Nunca é demais lembrar que o que é provisório quase sempre fica permanente.

Além disso, para que uma solução proposta não dependa de interpretações subjetivas de quais ações os comandos descrevem, os algoritmos devem ser rigorosos também neste aspecto. Devem, então, cumprir regras de execução que indicam a ordem de execução de cada um dos comandos (seqüencial, começando pelo primeiro), além de estabelecer

que um comando só é executado após o anterior ter sido totalmente concluído, sem voltas ou "jeitinhos".

A compreensão destes aspectos é de importância crucial para que os desenvolvimentos futuros sejam feitos adequadamente. Portanto, mesmo que alguns deles pareçam ser simples, ou mesmo óbvios, todos merecem uma revisão, certificando-se de que foram corretamente compreendidos.

Unidade 2 - Conceitos básicos para o desenvolvimento de algoritmos

2.1 Primeiras palavras

Para resolver problemas utilizando computadores, é necessário que uma solução seja projetada de forma adequada, ou seja, de forma que um computador tenha condições de realizá-la.

Assim, um primeiro passo para se aprender a propor uma solução algorítmica é lembrar que um algoritmo é simplesmente um manipulador de dados. Em outras palavras, os algoritmos representam informações importantes para um dado problema (como nomes de pessoas, valores de salários e taxas, medidas estatísticas ou até figuras) na forma de dados. Estes dados são lidos, modificados e reestruturados para produzir os resultados desejados. Neste contexto, esta unidade trabalha o conceito de representação de dados, começando sobre quais os tipos de dados que qualquer computador já é usualmente capaz de armazenar e relacionando esta representação ao modo como o computador interpreta os algoritmos (ou seja, os passos para gerar as soluções de um problema).

A "execução" das instruções de um algoritmo requer que se tenha conhecimento do computador que o executará, ou seja um modelo deste computador. Deste modo, pode-se então propor uma formalização da notação para os algoritmos aqui tratados, permitindo introduzir os primeiros comandos formais.

2.2 Por que dados têm tipos

O entendimento do conceito de **tipos abstratos de dados** pode ser relacionado a conceitos usados cotidianamente. Por exemplo, pense em sua idade. Pense, agora, na idade de seus pais, ou de seus irmãos, ou de seu namorado ou sua namorada. Para começar, é bem provável que as idades que pensou sejam medidas em anos (e não em meses ou dias). Outro ponto é que, raramente, você pensou na sua idade como 19,7 anos, por exemplo, pois é natural que idades sejam valores inteiros. Valores de idades, geralmente, estão associados a números como 8, 10, 18, 25, 55, 58 ou 70 e menos freqüentemente a números como 2,93 ou -14, por exemplo. Idades usualmente são valores inteiros e, quase sempre, positivos. Similarmente, pense em uma conta ou saldo bancário. Não faz sentido que seja um valor inteiro, mas que tenha também uma parte decimal, que na nossa moeda é dada em centavos. Pense também no consumo médio de combustível de um veículo: será algo como 9,6 quilômetros por litro. Alguns valores são, intrinsecamente, reais (isto é, têm também parte decimal).

Existem, além destes, outros tipos de dados, que são associados também a textos (como nomes, endereços ou placas de carros) ou até a condições, como se um saldo está negativo ou não, se um funcionário possui ou não filhos ou se dois conjuntos são iguais.

2.3 Representação das informações e processamento dos algoritmos

Compreender como um algoritmo manipula as informações recai em se ter um conhecimento prévio de quais os tipos de informações (dados) podem ser utilizados e como um computador (ou um modelo dele) interpreta estas informações e as instruções que o algoritmo indica fazer.

2.3.1 Tipos abstratos de dados

As informações que precisam ser armazenadas e manipuladas por algoritmos recaem em um conjunto bastante restrito de **tipos abstratos de dados**. Estes tipos incluem os **inteiros**, os **reais**, os **literais** e os **lógicos**, além de um outro tipo complementar denominado **ponteiros**.

O tipo de dados denominado **inteiro** compreende, como o nome implica, os valores numéricos que não possuem parte decimal. Valores como 3, 15, 334565, -443, 0 e 2000 são representantes dos inteiros. Além do conjunto de valores, o tipo abstrato inteiro inclui o elenco das operações válidas para valores inteiros. Dentre estas operações podem ser destacadas as já conhecidas soma e multiplicação, a título de exemplo. O conjunto de operações sobre os diversos tipos de dados é coberto na Unidade 3, que trata das expressões algorítmicas.

Outro tipo de dados é o tipo **real**. O conjunto de valores relativo aos reais inclui, por exemplo, 2,7, -3,8, 6,223, 400,0 e 0,0, entre outros. Para diferenciar valores inteiros dos reais, os decimais iguais a zero são indicados explicitamente. Como será visto mais adiante, nem sempre esta diferenciação é necessária. As operações sobre o tipo real são similares àquelas sobre o tipo inteiro e também serão vistas na próxima Unidade.

Além dos tipos numéricos, há ainda três outras classes de tipos de dados.

Para o armazenamento e manipulação de valores textuais, como nomes, códigos alfanuméricos e textos com e sem pontuação, existe o tipo abstrato denominado **literal**. Um literal corresponde, portanto, a uma cadeia de caracteres, seja lá quais forem. Além disso, para que não haja qualquer confusão, todas as constantes literais, sem exceção, são indicadas entre aspas. Por exemplo: "Aristóteles", "José de Arimatéia", "Avenida Carlos Gomes, 1233", "Qual sua senha?", "símbolos: @#$%¨&*" e "*** ATENÇÃO ***".

Um tipo de dado mais singular é o tipo **lógico**. O conjunto de valores lógicos inclui apenas dois elementos: o **verdadeiro** e o **falso**. Dados lógicos estão normalmente associados a condições, como a presença ou ausência de um atributo ou se um dado item foi ou não localizado em uma lista. Assim como os demais, há também operações que se aplicam ao tipo lógico.

Finalmente, há um tipo em destaque, denominado **ponteiro**. Devido ao seu uso muito particular, este tipo é tratado em separado, na Unidade 7.

Os valores escritos explicitamente são chamados valores **constantes** e a cada constante está associado um tipo específico, dentre os citados anteriormente. Para deixar mais claro, 10, 18, -3 e 9 são constantes

inteiras, 1,0, -0,001, 14,2 e -99,001 são constantes reais, "sadio", "solteiro", "custo do empreendimento" e "18" são constantes literais e, finalmente, **verdadeiro** e **falso** são os únicas constantes lógicas.

Uma observação importante deve ser feita, ainda. A diferença entre as constantes 18 e "18" deve ser bastante clara. Enquanto 18 refere-se a um número inteiro, quantificando o equivalente a dezoito unidades, "18" é um texto formado por dois caracteres (o "1" e o "8") e não se refere a grandeza numérica nenhuma. Esta distinção é importante, pois não pode haver ambigüidade nas representações. Outro exemplo, 43,8 e "43,8" formam outro par com propriedades similares ao exemplo anterior; agora, 43,8 é um número real, correspondendo a uma quantia, e "43,8" é um texto formado por quatro caracteres ("4", "3", "," e "8") e tem tanto valor numérico quanto teria "ABC", "Juvenal Urbino" ou "a1b2c3d4"; ou seja, nenhum.

2.3.2 Modelo de computador

Como apresentado na Unidade 1, um algoritmo deve ter claro quem ou o quê o executará, o que interfere no grau de detalhamento das instruções.

Para os algoritmos desta disciplina, a execução será feita por um computador fictício, similar em funcionalidade a um computador real, mas sem que se necessite explorar em demasia seus detalhes.

Este computador fictício define, assim, um **modelo de computador**, que de agora até o fim do texto, será tratado apenas por computador. Quando for necessária a distinção do modelo de computador de um computador real, o texto deixará isto bastante claro.

Deste modo, o computador que executa os algoritmos deve ter as seguintes características:

1) Poder guardar o algoritmo e ser capaz de executá-lo, instrução a instrução;
2) Poder armazenar os dados necessários para resolver o problema por meio do algoritmo;
3) Permitir que dados sejam transferidos para dentro do computador;
4) Permitir que dados sejam transferidos para fora do computador.

O esquema do computador é apresentado na Figura 2-1, destacando seus quatro módulos principais: unidade de processamento, memória e unidades de entrada e de saída.

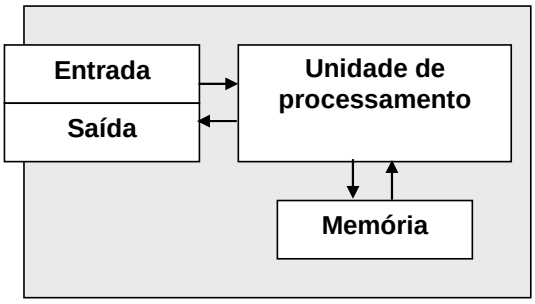


Figura 2-1. Unidades constituintes do modelo de computador.

A **unidade de processamento** é o "cérebro" do computador e é responsável por controlar tudo o que acontece. É o dispositivo capaz de interpretar as instruções dos algoritmos e executar as ações especificadas. O algoritmo é armazenado na **memória**, que, além do algoritmo, armazena também os dados que este utiliza. A unidade de processamento, quando necessário, armazena e recupera dados e instruções guardados na memória.

A interface com o "mundo real" é feita pelas **unidades de entrada e saída**. A unidade de entrada é capaz de pegar dados externos ao computador e, por intermédio da unidade de processamento, armazená-los na memória. Como dispositivos associados à unidade de entrada estão, por exemplo, o teclado, um leitor óptico, o modem (que recebe dados via linha telefônica), o mouse, o disco rígido (do qual são lidos dados) e um escâner, entre outros. Dispositivos ligados à unidade de saída são, por exemplo, a tela (monitor de vídeo), a impressora, um modem (quando este envia dados), o disco rígido (quando neles são armazenados dados) etc. Assim, o mundo externo é tudo que não pertence ao computador e o mundo interno é o que está armazenado na memória e o que é manipulado pela unidade de processamento.

2.3.3 Formalização da notação de algoritmos

O modelo de computador, assim como o computador real, tem suas limitações de entendimento de dados e de interpretação de instruções. Os algoritmos, desta forma, devem ser compostos de um conjunto finito de instruções, que recebem o nome de **comandos**. Cada comando tem um comportamento específico e claramente definido, de modo a não introduzir ambigüidade na interpretação da ação associada ao comando.

Desta maneira, os algoritmos são tratados segundo um modelo bastante formal, seguindo uma notação relativamente rígida.

Para começar, todos os algoritmos atendem ao formato geral apresentado pelo Algoritmo 2-1. As partes em texto destacadas (em negrito com itálico) são partes que variam dependendo da solução que o algoritmo apresenta; os delimitadores **algoritmo** e **fim-algoritmo** indicam a localização das instruções pelas quais a unidade de processamento inicia a execução dos comandos. É importante apontar que não é permitido misturar as instruções de declarações locais com os comandos, ou seja, são feitas primeiramente todas as declarações e, depois, especificados os comandos.

Algoritmo 2-1

```
1  documentação_do_algoritmo
2
3        declarações_globais
4
5  algoritmo
6        declarações_locais
7
8        lista_de_comandos
9  fim-algoritmo
```

Os algoritmos estão disponíveis para a unidade de processamento, ficando armazenados na memória. A explicação de como os algoritmos são transferidos para a memória e o modo como ficam armazenados nela está fora do escopo desta disciplina e não serão tratados neste texto.

2.3.4 Entrada, saída e atribuição

Até este momento, sabe-se que toda a execução de um algoritmo é de responsabilidade da unidade de processamento e que o algoritmo fica armazenado na memória. O algoritmo, por sua vez, indica como os dados, que também devem ficar armazenados na memória, são transformados para produzir a resposta a um problema. Sabe-se, também, que o conjunto de comandos que pode ser usado nos algoritmos é limitado.

Para iniciar, porém, é preciso definir como os dados são armazenados na memória e quais os comandos necessários para efetivar estes armazenamentos.

O conceito de **variáveis** é o ponto chave para entender a manipulação de dados. Uma variável corresponde a um dado armazenado na memória. Como muitas coisas são armazenadas na memória, a interpretação correta deste dado depende de identificar seu tipo. Portanto, uma variável sempre tem um tipo associado a ela. Para identificar os diversos dados existentes, é preciso dar um nome a eles. O nome de uma variável é chamado **identificador**.

O Algoritmo 2-2 ilustra a declaração de duas variáveis.

Algoritmo 2-2

```
1  { leitura de nome e idade com escrita de mensagem usando estes dados }
2
3  algoritmo
4        declare
5             nome: literal
6             idade: inteiro
7
8        { leitura de nome e idade do teclado }
9        leia(nome)
10       leia(idade)
11
```

```
12        { saída da mensagem na tela }
13        escreva(nome, " tem ", idade, " anos.")
14  fim-algoritmo
```

Na linha 4 existe a indicação da **declaração de variáveis.** Uma declaração equivale a informar que uma dada parte da memória deve ser reservada para guardar um determinado dado. Associado a estes dados está seu tipo, que pode ser inteiro, real, literal, lógico ou ponteiro.

Nas linhas 5 e 6 estão as declarações propriamente ditas. O identificador **nome**, do tipo literal (isto é, capaz de armazenar textos), é associado a um dado na memória. Da mesma forma, outro dado da memória, que será tratado como um inteiro, recebe o identificador **idade**. As linhas 4, 5 e 6 podem ser lidas como "declare as variáveis nome, como um dado literal, e a variável idade, como um dado inteiro". Deve ser observado que, uma vez definidas as áreas de memórias para variáveis e seus nomes, nenhum destes elementos se altera até o final do algoritmo.

Uma declaração, em sua forma geral, é dada conforme a especificação seguinte.

```
          declare lista_de_identificadores: nome_do_tipo
```

Na declaração, a lista de identificadores é uma lista separada por vírgulas. Caso haja separação entre as declarações (por tipos diferentes, por exemplo), pode-se omitir o **declare** nas declarações subseqüentes, como foi feito no algoritmo ilustrado.

Na linha 9 encontra-se o primeiro comando que é apresentado. O comando **leia** instrui o processador a realizar a seguinte ação: copiar do mundo externo um dado e o armazenar na(s) variável(eis) relacionada(s). No exemplo, o mundo externo pode ser visto como alguém digitando dados no teclado do computador. O comando indica que um nome (literal) que for digitado deve ser armazenado na memória, sob o identificador **nome**. A linha 10 indica que um valor numérico (inteiro) digitado deve ser armazenado na memória na parte referenciada por **idade**. A unidade de processamento aguarda que cada comando **leia** seja finalizado para prosseguir.

Suponha que o usuário (pessoa que usa o computador) tenha digitado "Sigmund" e pressionado ENTER. A esta ação, o valor literal "Sigmund" é copiado para a memória referenciada por **nome**. Em seguida, é digitado "21" e digitado ENTER novamente, o que faz a unidade de processamento interpretar o que foi escrito e transferir o valor inteiro 21 para parte da memória chamada **idade**.

Seguindo seu tratamento usual, o próximo comando feito pela unidade de processamento é o **escreva** da linha 13. Um comando **escreva** instrui à cópia dos dados indicados para o dispositivo de saída. Neste caso, pode-se assumir que a saída é a tela do computador, na qual o usuário pode visualizar o resultado. No exemplo, o comando **escreva** indica que quatro itens sejam copiados para a tela: o conteúdo da variável **nome**, um valor literal constante (" tem "), o conteúdo da variável **idade** e, por fim,

uma outra constante literal (" anos."). O resultado é, neste caso, "Sigmund tem 21 anos.".

O comando **leia** tem o formato indicado abaixo.

```
leia(lista_de_variáveis)
```

A lista de variáveis corresponde a uma coleção de um ou mais identificadores, separados por vírgulas e indicados entre os parênteses.

O comando **escreva** tem o formato indicado na seqüência.

```
escreva(lista_de_expressões)
```

Similarmente ao comando leia, cada item é especificado dentro dos parênteses e cada um separado dos demais por vírgulas. Porém, enquanto no comando **leia** os itens são os identificadores, no **escreva** aceita-se uma lista de expressões, que podem ser valores de variáveis, constantes de quaisquer tipos e mesmo expressões propriamente ditas. Como um exemplo simples, se o comando fosse **escreva(idade + 1)**, o valor 22 seria escrito. Mais detalhes sobre expressões serão tratados na Unidade 3.

Retomando um problema de consumo médio como o visto na Unidade 1, porém um pouco mais simples, é proposto o seguinte: "Escreva um algoritmo que calcule o consumo médio de um veículo, dadas as quantidades em litros de exatamente 2 abastecimentos realizados (um no meio da viagem e outro no final) e também a quilometragem total percorrida, apresentando o resultado. Considere que o veículo saiu de tanque cheio e em ambos os abastecimentos o tanque foi novamente completado".

O Algoritmo 2-3 propõe uma possível solução para este novo problema (que é diferente do visto na Unidade 1).

Algoritmo 2-3

```
1  { cálculo do consumo médio de um veículo, dados a quilometragem total
2    percorrida e o número de litros utilizados nos dois únicos
3    abastecimentos realizados }
4
5  algoritmo
6        declare
7             quilometragem, abastecimento1, abastecimento2: real
8
9        { obtenção da quilometragem e litros consumidos }
10       leia(quilometragem)
11       leia(abastecimento1, abastecimento2)
12
13       { escrita do resultado do consumo }
14       escreva(quilometragem/(abastecimento1 + abastecimento2), " km/l")
15 fim-algoritmo
```

O algoritmo declara três variáveis (**quilometragem**, **abastecimento1** e **abastecimento2**) para guardar os dados necessários ao cálculo, como apresentado nas linhas 6 e 7. As linhas 10 e 11 especificam dois comandos de leitura, para que o usuário digite os valores, todos reais, para as três variáveis. Quando o usuário as digitar, seus valores serão transferidos para a memória, segundo os identificadores especificados. No final, na linha 14, está o comando de escrita, que indica que seja transferido para a tela o valor equivalente ao conteúdo da variável **quilometragem** dividido pela soma do consumo de litros, indicados nas variáveis **abastecimento1** e **abastecimento2**, seguido por um texto (constante literal) com valor " km/l".

Por exemplo, assumindo-se que tenha sido digitado 218 para o valor total da quilometragem, indicando 218 km, e 11,7 e 8,3 respectivamente para os dois os abastecimentos (representando 11,7 e 8,3 litros), o resultado produzido seria "10,9 km/l".

Até agora, a única forma de colocar um dado na memória, na área reservada para uma variável, é usando o comando **leia**. Em várias situações, é preciso colocar os valores em variáveis, mas estes valores não estão no "mundo exterior" (ou seja, não serão digitados, por exemplo), mas foram calculados pelo próprio algoritmo. O comando de **atribuição** é usado para esta finalidade.

Para continuar o exemplo, suponha que a alternativa para o cálculo do consumo fosse, ao invés de usar o comando **escreva** com a especificação da expressão do cálculo, armazenar o resultado da conta na memória, para depois escrevê-lo. Nesta situação, o formato fica como no Algoritmo 2-4.

Algoritmo 2-4

```
1  { cálculo do consumo médio de um veículo, dados a quilometragem total
2    percorrida e o número de litros utilizados nos dois únicos
3    abastecimentos realizados }
4
5  algoritmo
6        declare
7             quilometragem, consumo,
8             abastecimento1, abastecimento2: real
9
10       { obtenção da quilometragem e litros consumidos }
11       leia(quilometragem)
12       leia(abastecimento1, abastecimento2)
13
14       { cálculo do consumo }
15       consumo ← quilometragem/(abastecimento1 + abastecimento2)
16
17       { saída do resultado calculado }
18       escreva(consumo, " km/l")
19 fim-algoritmo
```

A linha 15 mostra como o valor de uma expressão (o cálculo do consumo) é alocado para a memória indicada pelo identificador **consumo**, que está associado a um dado real. Para que possa ser usada, a variável **consumo** também teve que ser declarada (linha 7).

A **atribuição** é sempre indicada com o identificador à esquerda na linha de comando, seguido de uma seta para a esquerda (←) que indica que toda a expressão que estiver à direita será calculada e o valor resultante será armazenado na memória da variável especificada pelo identificador.

O comando de atribuição tem sempre o formato indicado na seqüência.

```
identificador ← expressão
```

É sempre importante destacar que deve haver uma compatibilidade de tipos entre a expressão e a variável. Assim, não faz sentido a expressão ser real se a variável tiver o tipo literal ou lógico. Como regra geral, o tipo resultante da expressão deve ser igual ao da variável. A única exceção é quanto a reais e inteiros pois, como ambos são numéricos, há a promoção para real ao se atribuir um valor inteiro para uma variável real, assim como a perda da parte fracionária ao se atribuir um dado real para uma variável inteira. Importante observar que, neste último caso, não é feito o arredondamento, mas sim a simples eliminação da parte fracionária, de modo que 5,02 e 5,999, ao serem atribuídos a um inteiro, produziriam o valor 5.

2.3.5 Regras para a formação de identificadores

Para que não haja dúvidas de interpretação nem outros problemas decorrentes, algumas regras são definidas para que um identificador seja criado. São elas

1) Todo identificador é formado apenas por letras do alfabeto (*a* a *z*), por dígitos (*0* a *9*) e pela sublinha (_);
2) O primeiro caractere é sempre uma letra ou a sublinha; nunca um dígito;
3) O identificador escolhido deve ser significativo.

Decorrentes da primeira regra, não são permitidos espaços, símbolos (&, #, ! etc.) ou outros caracteres. Também não são permitidas letras ou símbolos de outros alfabetos, como o grego ou o cirílico.

A terceira regra obriga que uma variável tenha nome significativo, ou seja, que o nome ajude a entender sua função no algoritmo. Por exemplo, **pia** é um identificador corretamente formado, mas não faz sentido que seja usado se, no algoritmo, a variável for usada para guardar a taxa de juros da poupança. No Algoritmo 2-3 são usados identificadores que deixam clara a função de cada variável no contexto do algoritmo.

Recomenda-se, ainda, que seja adotado o seguinte estilo para os algoritmos:

1) O uso de minúsculas deve ser privilegiado;

2) Quando um identificador for formado pela justaposição de duas ou mais partes, a partir da segunda deve-se usar uma maiúscula para indicar a transição de uma palavra para outra.

O uso de minúsculas é mais legível e agradável à leitura, pois produz linhas menos densas, além de, atualmente, escritas em maiúsculas terem uma conotação de "falar alto" ou "gritar", como é usual nos fóruns e salas de bate-papo. A única tolerância no uso de maiúsculas é no caso da segunda regra, que abrange nomes compostos. Segundo esta sugestão, usada internacionalmente pelos programadores, são recomendados identificadores como **salárioMínimo**, **consumoTotal** e **taxaCpmf**, por exemplo. Todos os algoritmos deste texto utilizam estas convenções e é fortemente recomendado que todos façam sua adesão ao estilo.

Como exemplo (ou contra-exemplo), o Algoritmo 2-5 mostra o Algoritmo 2-4 usando apenas maiúsculas, o que o torna desagradável de ler.

Algoritmo 2-5

```
1  { CÁLCULO DO CONSUMO MÉDIO DE UM VEÍCULO, DADOS A QUILOMETRAGEM TOTAL
2    PERCORRIDA E O NÚMERO DE LITROS UTILIZADOS NOS DOIS ÚNICOS
3    ABASTECIMENTOS REALIZADOS }
4
5  ALGORITMO
6        DECLARE
7             QUILOMETRAGEM, CONSUMO,
8             ABASTECIMENTO1, ABASTECIMENTO2: REAL
9
10       { OBTENÇÃO DA QUILOMETRAGEM E LITROS CONSUMIDOS }
11       LEIA(QUILOMETRAGEM)
12       LEIA(ABASTECIMENTO1, ABASTECIMENTO2)
13
14       { CÁLCULO DO CONSUMO }
15       CONSUMO ← QUILOMETRAGEM/(ABASTECIMENTO1 + ABASTECIMENTO2)
16
17       { SAÍDA DO RESULTADO CALCULADO }
18       ESCREVA(CONSUMO, " KM/L")
19 FIM-ALGORITMO
```

2.4 Considerações finais

Um modelo de computador foi definido, cuja responsabilidade é executar os algoritmos. Para tanto, este modelo define uma interface com o mundo, permitindo a entrada de dados (obtenção de dados do mundo externo) e também a saída (que fornece dados do algoritmo para o meio externo). Além disso, existe a unidade de processamento, que é capaz de interpretar as instruções dos algoritmos e realizar suas ações, usando para isso a memória, a qual é responsável por armazenar tanto o algoritmo em si quanto os diversos dados utilizados.

A memória pode ser vista como um conjunto de dados, cada qual ocupando uma parte dela. Para saber exatamente como interpretar os dados de cada parte da memória, um tipo de dados deve ser associado a ele, sendo os tipos inteiro, real, lógico, literal e ponteiro os tipos básicos válidos.

Uma área de memória, à qual se associa um tipo, pode ser manipulada pelo algoritmo por meio de um identificador. Esta parte da memória, por poder ter seu valor modificado com o tempo, recebe o nome genérico de variável. É através da manipulação dos conteúdos armazenados nas variáveis que os algoritmos, por meio de instruções adequadas, resolvem um problema.

As principais formas de se modificar o conteúdo de uma variável é por meio de um comando de leitura (comando **leia**), ou seja, solicitando à unidade de processamento que transfira dados externos (teclado, por exemplo) para a memória no local definido pelo identificador, ou através da atribuição de um valor diretamente à variável (usando o comando de atribuição).

Unidade 3 - Expressões algorítmicas

3.1 Primeiras palavras

Se algoritmos são manipuladores de dados utilizados para que um determinado problema seja tratado e uma resposta correta produzida, o modo como os dados são tratados e estruturados assume papel importante neste contexto. Nas Unidades anteriores foram apresentadas as maneiras como os comandos de um algoritmo são executados e como as variáveis são definidas. É nas variáveis que são armazenados os dados e, em conseqüência, é nelas que os comandos agem e os dados são transformados.

Para que sejam especificadas corretamente as transformações dos dados, é preciso que sejam definidas as expressões que os manipulam sem que haja ambigüidade nas interpretações. Deste modo, esta Unidade trata das **expressões algorítmicas** e suas propriedades.

3.2 Matemática e precisão da notação

Existe muito em comum entre as expressões algorítmicas e as expressões matemáticas que são aprendidas desde os primeiros anos de estudo. Expressões matemáticas são indicadas por operandos e operadores. Na expressão $2 \times 3 + 7$, por exemplo, são indicadas as operações de multiplicação e de soma; os operandos da multiplicação são os valores 2 e 3 e os da adição, o resultado da multiplicação (6) e o 7. O resultado, claro, é 13.

Outra expressão, como $7 + 2 \times 3$, é... a mesma expressão. Na realidade, é uma expressão equivalente. Nesta nova expressão, as operações e resultados são os mesmos, mas uma diferença deve ser notada: a multiplicação, embora colocada após a adição, é calculada antes. Expressões matemáticas possuem, naturalmente, prioridade entre os operadores, o que tem evidente importância no que a expressão representa.

Expressões algorítmicas são similares às expressões matemáticas. Possuem operadores, operandos e ordem correta de avaliação. Também têm algumas outras restrições, que as tornam adequadas aos algoritmos, e devem tratar distintamente os vários tipos dados.

3.3 Expressões em algoritmos

As expressões em algoritmos, do mesmo modo que as matemáticas já conhecidas de todos, têm como característica uma formalidade que evita ambigüidades. Assim, apresentada uma expressão qualquer, as regras envolvidas determinam em que ordem cada operação deve ser avaliada para que, ao final, um resultado específico seja calculado. Esta Unidade trata da introdução, nos algoritmos, dos diversos tipos de expressões válidos para que sejam utilizados nas soluções propostas.

3.3.1 Linearidade das expressões, operadores e precedência

A formalização, tão necessária desde a primeira unidade, começa por definir as regras básicas para que as expressões sejam construídas corretamente:

1) Todas as expressões são **lineares**, ou seja, são escritas na altura de uma linha;
2) Os operadores são sempre explícitos e devem ser coerentes com os tipos de dados envolvidos;
3) As relações entre os tipos de dados devem ser mantidas de forma coerente.

O fato de expressões serem lineares implica que não há liberdade de usar visualmente o espaço, como é feito na matemática. A Figura 3-1(a-c) apresenta algumas expressões matemáticas cujos elementos não ficam todos em uma única linha. Nos algoritmos, isto é uma necessidade, de forma que modificações são requeridas para que fiquem na forma linear. A Figura 3-1(d-f) mostra as mesmas expressões em forma linear.

a/b	$a\dfrac{b}{c+d}$	$a+\dfrac{b}{c/d}$
(a)	(b)	(c)
a/b	$a*b/(c+d)$	$a+b/(c/d)$
(d)	(e)	(f)

Figura 3-1. (a-c) Exemplos de expressões matemáticas. (d-f) Versões algorítmicas das mesmas expressões.

A segunda regra exige que todos os operadores sejam explícitos. Portanto, expressões que incluam a multiplicação, que geralmente é omitida na matemática, como na Figura 3-1(b), devem estar explicitamente apresentadas, como na versão algorítmica mostrada na Figura 3-1(e). Deve-se notar, em particular, que a multiplicação é indicada por um asterisco.

Tratando dos tipos de dados envolvidos, a terceira regra exige coerência entre os valores e resultados. Estes aspectos de coerência são tratados nas seções seguintes, para os vários tipos de expressões que aparecem em algoritmos: as **aritméticas**, as **literais**, as **relacionais** e as **lógicas**.

Já na Unidade 2, no Algoritmo 2-3, é exemplificada uma expressão escrita na forma linear e seguindo as orientações aqui especificadas.

3.3.2 Expressões aritméticas

Os tipos de dados inteiro e real são tratados conjuntamente nas **expressões aritméticas**. Este tipo de expressão está diretamente

vinculado ao conceito de expressões matemáticas, como as exemplificadas na seção anterior.

As expressões aritméticas utilizam o conjunto de operadores apresentado na Figura 3-2. A primeira observação é sobre os dois operadores que usam o mesmo símbolo: o sinal de menos. O primeiro, junto como %, é o operador unário, usado em expressões como *–b*, e o segundo é o binário, que exige dois operandos, como em *a - b*. Outro ponto é a ordem de prioridade, sendo que os de maior prioridade (avaliados antes) estão na parte superior e os de menor prioridade na parte inferior. Assim, em uma expressão com os operadores % (módulo) e * (multiplicação), o módulo é calculado antes. Alteração na ordem de precedência dos operadores é feita com o uso de parênteses, que podem ser usados em qualquer tipo de expressão.

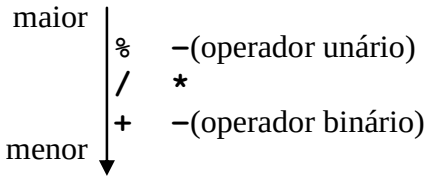


Figura 3-2. Operadores aritméticos.

Ainda na questão sobre a precedência dos operadores, operadores que estão na mesma linha na Figura 3-2, isto é, que possuem a mesma precedência, são avaliados da esquerda para a direita. Por exemplo, *3 * 8 / 2* tem a multiplicação do 3 pelo 8 sendo feita antes da divisão deste resultado intermediário por 2. Na expressão anterior, pode-se argumentar que fazer a divisão antes da multiplicação não alteraria o resultado, o que é verdadeiro neste caso. Embora, em geral, esta ordem pareça não fazer diferença, quando são usadas divisões o resultado pode não ser o pretendido, como no caso da expressão *3 / 8 / 2*. Nesta nova situação, ambas as divisões têm a mesma precedência, mas dividir 3 por 8 para então dividir por 2 é diferente de dividir 8 por 2, para então dividir 3 por este resultado. Confira as contas!

Todos os operadores, exceto o de módulo (%), podem ser usados tanto em operandos inteiros quanto reais. Assim, adota-se a regra que, para dois operandos do mesmo tipo, o resultado segue o mesmo tipo. Portanto, a soma de dois inteiros resulta em inteiro e a multiplicação de dois reais resulta em um real. Um destaque deve ser feito à divisão, pois a divisão de dois inteiros resulta em um inteiro, de forma que a expressão *5 / 2* tem valor 2, e não 2,5. Esta é uma característica que requer atenção e deve ser tratada com cuidado, sob a pena de especificar expressões incorretas.

Quando há mistura de tipos, adota-se a chamada **regra da promoção**, que promove o tipo mais "simples" para o mais "completo". Isto quer dizer que, no caso de operações que envolvam inteiros e reais, primeiro o inteiro é "promovido" a real, e então a operação é feita. Por exemplo, *10,2 * 2* tem o 2 (inteiro) promovido para 2,0 (real); então a multiplicação de real por real é efetuada, resultando neste caso em 20,4. Enquanto a expressão *21 / 2 * 2* resulta 20 (lembrando que o inteiro 21

dividido pelo inteiro 2 resulta no também inteiro 10), *21 / 2,0 *2* resulta 21,0, pois o 21 é promovido a real antes da divisão pelo real 2,0 (com resultado intermediário 10,5); então o inteiro 2 é promovido a real, antes da multiplicação final. Atenção especial deve sempre ser dada a estas situações.

Finalmente, o chamado **operador modular**, indicado pelo símbolo % e usado exclusivamente para operandos inteiros, resulta no resto de uma divisão inteira. Por exemplo, *10 % 3* resulta em 1, pois 10 dividido por 3 resulta em 3, com resto 1. Similarmente, *6 % 2* resulta 0, *7 % 2* resulta 1, *254 % 100* resulta 54. Pode-se observar que o resto de divisão por um valor *k* está sempre entre os valores 0, 1, 2..., *k*–1.

Não somente operações estão disponíveis para as expressões, mas também algumas funções de uso geral, como as apresentadas na Tabela 3-I.

Tabela 3-I. Lista das funções usuais para expressões aritméticas.

Função	Parâmetros	Valor de retorno	Descrição
`sen(r), cos(r), tan(r)`	`real`	`real`	Funções trigonométricas, com parâmetros em radianos
`asen(r), acos(r), atan(r)`	`real`	`real`	Funções trigonométricas inversas, retornando o valor do arco em radianos
`ln(r)`	`real`	`real`	Logaritmo neperiano (base *e*)
`log(r)`	`real`	`real`	Logaritmo base 10
`exp(r)`	`real`	`real`	*e* elevado à potência *r*
`pot(v, r)`	`real/inteiro, real`	tipo do primeiro argumento	Potência v^r
`sinal(i)`	`inteiro`	`inteiro`	Retorna 1 se o valor for positivo ou –1 se for negativo; zero se for nulo
`trunca(r)` ou `int(r)`	`real`	`inteiro`	Retorna somente a parte inteira do argumento passado
`frac(r)`	`real`	`real`	Retorna somente a parte fracionária do argumento
`arred(r)`	`real`	`inteiro`	Retorna o inteiro mais próximo do valor real passado

abs(v)	real/inteiro	tipo do primeiro argumento	Retorna o valor absoluto (sem sinal) do argumento, mantendo o tipo especificado no parâmetro
resto(r, i)	real, inteiro	inteiro	Retorna o resto da divisão inteira de *r* por *i*
quoc(r, i)	real, inteiro	inteiro	Retorna o quociente da divisão inteira de *r* por *i*

O Algoritmo 3-1 é uma proposta de solução para o seguinte problema: "Uma lanchonete precisa, em um determinado momento, calcular o total da conta de uma mesa e, dado o valor pago pelos clientes, calcular o troco. Os dados disponíveis, além do valor pago pelos clientes, são o valor total consumido em bebidas e o total consumido em alimentos. Sabe-se também que, como padrão, também é incluído na conta a gorjeta do garçom, no valor correspondente a 10% do total da conta. É preciso escrever um algoritmo que reporte o valor total da conta, o valor total pago pelos clientes e o valor do troco.".

Algoritmo 3-1

```
1  { apresentar, para uma conta de lanchonete, o valor total da conta,
2    o valor pago e o troco devolvido, dados o total consumido de bebidas
3    e alimentos, além de se saber que a porcentagem do garçom é 10% }
4
5  algoritmo
6        declare totalBebidas, totalAlimentos,
7                valorPago, troco,
8                porcentagemGarçom: real
9
10       { obtenção dos dados necessários }
11       leia(totalBebidas, totalAlimentos, valorPago)
12
13       { cálculos necessários }
14       porcentagemGarçom ← (totalBebidas + totalAlimentos) *
15                                                  10/100,0 { 10% }
16       troco ← valorPago – (totalBebidas + totalAlimentos +
17                                              porcentagemGarçom)
18
19       { apresentação dos resultados }
20       escreva("Valor da conta:",
21                       totalBebidas + totalAlimentos + porcentagemGarçom)
22       escreva("Valor pago:", valorPago)
23       escreva("Troco:", troco)
24 fim-algoritmo
```

Deve-se notar que, no algoritmo, todas as variáveis são reais. Um potencial problema com tipos de dados foi resolvido na linha 15, ao se calcular o valor de 10%. Caso a expressão fosse escrita apenas *10/100*, o

valor recebido pelo garçom seria de zero, pois ambos os valores são inteiros e, portanto, o resultado da divisão daria zero. O uso de um dos operandos com valor real exige a promoção do outro e o resultado desejado (0,1) é obtido. Também podem ser observadas as expressões utilizadas em outros locais, principalmente com o uso de parênteses.

3.3.3 Expressões literais

Os tipos literais são mais restritos em termos de operações, pois são constituídos somente por texto. É definida, portanto, apenas uma operação sobre literais, que é a concatenação. Desta forma, dados os literais "texto inicial" e "texto final", a expressão *"texto inicial"* + *"texto final"* resulta no literal "texto inicialtexto final". Havendo várias concatenações, assume-se que sejam feitas da esquerda para a direita, embora isso não dê diferença no resultado final: a expressão *"início"* + " " + *"fim"* resulta "início fim", notando-se que " " indica um literal contendo um espaço em branco (diferentemente de "", que indica um literal vazio, sem nenhum caractere).

Algumas funções sobre literais também estão disponíveis, com as mais usuais apresentadas na

Tabela 3-II. Lista das funções usuais para expressões literais

Função	Parâmetros	Valor de retorno	Descrição
`comprLiteral(s)`	`literal`	`inteiro`	Retorna o comprimento da cadeia de caracteres, incluindo os espaços
`posLiteral(s1, s2)`	`literal, literal`	`inteiro`	Retorna a primeira posição onde a cadeia *s1* ocorre na cadeia *s2*, sendo que o primeiro caractere é considerado como posição 0 (zero); retorna -1 se não ocorrer nenhuma vez.
`subLiteral(s1, n1, n2)`	`literal, inteiro, inteiro`	`literal`	Retorna a sub-cadeia de *s1*, começando na posição *n1* e comprimento total *n2*
`valLiteral (s)`	`literal`	tipo equivalente ao valor numérico representado	Retorna o valor numérico de uma cadeia de caracteres que contém um número válido; o tipo resultante depende do valor representado no literal

Seja o problema: "Dados o primeiro nome e o sobrenome de uma pessoa, compor o nome em dois formatos: '*Fulano Tal*' e '*Tal, F.*' (como na lista telefônica).". O Algoritmo 3-2 ilustra uma possível solução para este problema, ilustrando o uso de expressões literais.

Algoritmo 3-2

```
1  { dados, separadamente, prenome e sobrenome, escrevê-los em dois
2    formatos: "Fulano Tal" e "Tal, F." }
3
4  algoritmo
5        declare prenome, sobrenome,
6                formato1, formato2: literal
7
8        { obtenção dos nomes }
9        leia(prenome, sobrenome)
10
11       { composição dos nomes }
12       formato1 ← prenome + " " + sobrenome
13       formato2 ← sobrenome + ", " + subLiteral(prenome, 1, 1) + "."
14
15       { resultados }
16       escreva(formato1)
17       escreva(formato2)
18 fim-algoritmo
```

3.3.4 Expressões relacionais

Comparações de valores permitem ao algoritmo executar ações diferentes de acordo com os resultados. **Operadores relacionais** são operadores de comparação, ou seja, que avaliam a relação entre dois valores. Todos os operadores relacionais são binários, isto é, com dois operandos, e sempre resultam em **verdadeiro** ou **falso**.

A Tabela 3-III apresenta os operadores relacionais.

Tabela 3-III. Operadores relacionais.

Operador	Função
=	Igual
≠	Diferente
<>	Diferente (símbolo alternativo)
>	Estritamente maior
<	Estritamente menor
≥	Maior ou igual
>=	Maior ou igual (símbolo alternativo)
≤	Menor ou igual
<=	Menor ou igual (símbolo alternativo)

As comparações podem ser realizadas desde que os tipos sejam equivalentes.

Expressões inteiras e reais podem ser comparadas entre si, considerando as mesmas regras de promoção descritas para as expressões aritméticas.

Literais também podem ser comparados com outros literais, usando todos os operadores. No caso do tipo literal, o resultado é realizado em termos textuais, seguindo a ordem alfabética. Por exemplo, a expressão *"abacate" > "abacaxi"* resulta **falso**, *"página" = "página"* resulta **verdadeiro** e *"forma" ≠ "forma "* também é **verdadeiro** (pois há um espaço adicional na segunda constante literal).

No caso dos tipos **ponteiro** e **lógico**, os únicos operadores válidos são os de igualdade e desigualdade.

Combinações de várias expressões lógicas são feitas por operadores lógicos, compondo as expressões lógicas, apresentadas na próxima seção. Um algoritmo exemplo é apresentado na próxima seção, ilustrando o uso de expressões relacionais e lógicas.

3.3.5 Expressões lógicas

Os valores lógicos são somente **verdadeiro** e **falso**. Para estes valores, são definidas as operações **não**, **e** e **ou**. A Figura 3-3 mostra estes operadores e sua ordem de precedência, a qual, naturalmente, pode ser quebrada com o uso de parênteses.

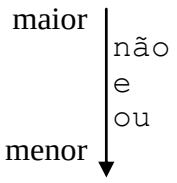


Figura 3-3. Operadores lógicos.

As operações lógicas seguem de forma bastante próxima o raciocínio lógico. Assim, o que não é verdadeiro é falso, e o que não é falso é verdadeiro, conforme a **tabela verdade** contida na Tabela 3-IV.

Tabela 3-IV. Tabela verdade para o operador *não*.

Operação	Resultado
não falso	verdadeiro
não verdadeiro	falso

O operador **e** exige que ambos os operandos tenham valor **verdadeiro** para que o resultado seja **verdadeiro**. Portanto, todas as outras combinações resultam **falso**. A Tabela 3-V mostra as combinações e os resultados.

Tabela 3-V. Tabela verdade para o operador *e*.

Operação	Resultado
`falso e falso`	`falso`
`falso e verdadeiro`	`falso`
`verdadeiro e falso`	`falso`
`verdadeiro e verdadeiro`	`verdadeiro`

A verificação do operador **ou** permite que o resultado seja **verdadeiro** bastando que um dos operandos também seja **verdadeiro**. O resultado é **falso** somente quando ambos os operadores tiverem valor **falso**, como mostrado na Tabela 3-VI.

Tabela 3-VI. Tabela verdade para o operador *ou*.

Operação	Resultado
`falso ou falso`	`falso`
`falso ou verdadeiro`	`verdadeiro`
`verdadeiro ou falso`	`verdadeiro`
`verdadeiro ou verdadeiro`	`verdadeiro`

As expressões lógicas são essenciais nas verificações de condições ou situações durante a execução de algoritmos. Por exemplo, um algoritmo pode tomar atitudes diferentes conforme o valor do saldo seja negativo **e** a conta não possua limite adicional (cheque especial). Condições mais complexas, que envolvem várias situações simultaneamente também podem ser modeladas nas expressões.

O Algoritmo 3-3 mostra um exemplo de verificações de algumas situações. O problema dado é o seguinte: "Verificar se um triângulo, descrito pelo comprimento de cada um de seus lados, é eqüilátero, isósceles ou escaleno, caso os lados ainda formem um triângulo.".

Algoritmo 3-3

```
1  { determinar, dados os comprimentos dos lados de um triângulo, se o
2    triângulo pode ser formado (se existe) e qual o tipo dele (eqüilátero,
3    isósceles ou escaleno) }
4
5  algoritmo
6        declare
7             lado1, lado2, lado3: real
8             existeTriângulo,
9             éEqüilátero, éIsósceles, éEscaleno: lógico
10
11       { leitura dos comprimentos dos lados }
12       leia(lado1, lado2, lado3)
13
14       { verificações necessárias }
15       existeTriângulo ← lado1 < lado2 + lado3 e
16                         lado2 < lado1 + lado3 e
```

```
17                          lado3 < lado1 + lado2
18       éEqüilátero ← existeTriângulo e lado1 = lado2 e lado2 = lado3
19       éIsósceles ← existeTriângulo e não éEqüilátero e
20                        (lado1 = lado2 ou lado2 = lado3 ou lado1 = lado3)
21       éEscaleno ← existeTriângulo e não éEqüilátero e não éIsósceles
22
23       { resultados }
24       escreva("Triângulo existe?", existeTriângulo)
25       escreva("É eqüilátero?", éEqüilátero)
26       escreva("É isósceles?", éIsósceles)
27       escreva("É escaleno?", éEscaleno)
28 fim-algoritmo
```

Os comandos **escreva**, nas linhas de 24 a 27, produzem como resultado as mensagens especificadas, seguido das palavras *verdadeiro* ou *falso*, conforme o valor lógico contido nas variáveis.

3.3.6 Ordem global de precedências de operadores

Quando as expressões misturam expressões aritméticas, relacionais e lógicas simultaneamente, a precedência sempre é feita avaliando-se primeiramente as partes aritméticas, seguidas pelas relacionais e, finalmente, as lógicas, sempre da esquerda para a direita.

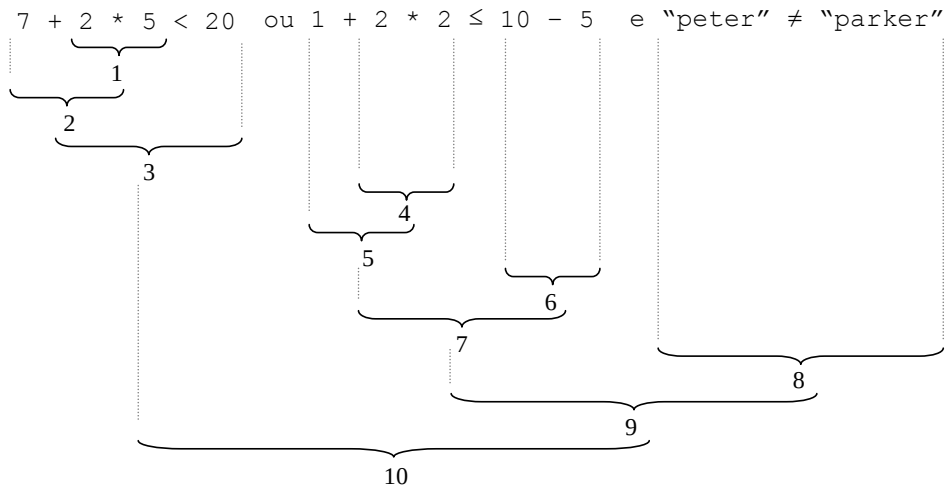


Figura 3-4. Ordem de avaliação de uma expressão composta.

A Figura 3-4 apresenta uma expressão e a ordem em que as diversas partes são avaliadas.

3.3.7 Uso de parênteses nas expressões

Quando se desejar que a ordem natural de precedência seja modificada, pares de parênteses são usados. Podem ser empregados tantos níveis de parênteses quanto forem necessários, aninhados ou não. Em

algoritmos exclusivamente os parênteses são usados, pois colchetes ([ ]) e chaves ({ }) possuem significados diferentes.

A Tabela 3-VII apresenta alguns exemplos nos quais o uso dos parênteses produz alterações nos resultados, pois alteram a precedência natural dos operadores. Especial atenção ainda pode ser dada às linhas 19 e 20 do Algoritmo 3-3, na qual a ausência dos parênteses alteraria o resultado, tornando-o incorreto.

Tabela 3-VII. Exemplos de expressões com parênteses.

Expressão	Resultado
`3 * (8 + 14) + 4`	70
`-100 * (200 – 50)/(9 – 1)`	-1875
`(200 – 50)/(9 – 1) * -100`	-1800
`(-5 – 14,0)/((2 + 6) * 10)`	-0,2375
`(8 + 7)/2 > 6 + 1`	**falso**
`45 > 10 e ("torre" < "pilar" ou "abc" = "abc")`	**verdadeiro**

3.3.8 Organização visual nas expressões

Em todas as expressões devem ser usados espaços entre os operadores e os operandos, para aumentar a clareza da expressão. Apenas o operador de divisão pode ser usado sem os espaços, pois não há comprometimento visual, principalmente quando próximo a parênteses.

É considerado uma prática muito ruim omitir os espaços, portanto este aspecto será cobrado na escrita dos algoritmos, juntamente como a organização visual geral (já visto na seção 1.4 da Unidade 1).

3.4 Considerações finais

Expressões algorítmicas apresentam uma característica bastante dúbia. Se por um lado parecem ser bastante naturais, a ponto de alguns pensarem "é fácil", por outro lado é repleta de detalhes, que levam muito facilmente a erros. Expressões algorítmicas não são expressões matemáticas e não estão sujeitas ao mesmo tipo de interpretação. Se, na matemática se escreve $a > b, a \neq b \neq 0$ querendo dizer que *a* é sempre maior que *b* e ambos são diferentes de zero, tal expressão em algoritmos não tem absolutamente nenhum sentido, pois devem ser usadas as regras de avaliação apresentadas.

Os principais pontos que devem ter atenção especial envolvem o uso correto dos operadores (que são sempre obrigatórios) e os tipos dos operandos. Especificar uma expressão de forma incorreta tem implicações graves na interpretação do algoritmo e, portanto, no resultado produzido. Em particular, as promoções e decessos entre reais e inteiros deve ser tratada com muito cuidado.

A prática é a melhor forma de se obter domínio sobre as expressões algorítmicas. Portanto, estudo e exercícios são fundamentais.

Unidade 4 - Estratégia de desenvolvimento de algoritmos

4.1 Primeiras palavras

O que se sabe de algoritmos até o momento é bastante limitado para que problemas um pouco mais complexos sejam resolvidos. Entretanto, os fundamentos apresentados nas Unidades de 1 a 3 são de extrema importância e devem ter sido compreendidos completamente.

Esta Unidade quebra um pouco o fluxo do texto, que vinha construindo o conhecimento sobre algoritmos, para mostrar um ponto importante, o qual será utilizado até o final deste material. Este ponto, que não é somente importante, mas essencial, é a **filosofia de desenvolvimento**.

Propor um algoritmo não depende somente do conhecimento das variáveis, dos tipos de dados, da estruturação das expressões e do uso dos comandos, mas também do modo que são utilizados. Em outras palavras, é possível saber como funcionam diodos, transistores, resistores e outros itens eletrônicos, mas daí a fazer um rádio, é outra história.

Esta Unidade trata de como se pensar para que um problema seja resolvido pela proposição de um algoritmo. Esta **metodologia de desenvolvimento** também é vista sob um enfoque um pouco mais formal.

4.2 Organização para resolver problemas

Cada um tem um modo de resolver seus problemas, sejam eles quais forem. Se for preciso arrumar o guarda-roupa, que está entulhado de roupas e outros objetos que despencam toda vez que uma porta é aberta, cada um tem seu próprio jeito de fazer a arrumação.

Uma pessoa, por exemplo, opta por retirar tudo e espalhar sobre a cama e outros móveis, agrupando as camisas, as calças e as blusas, já separando as coisas velhas e outros objetos. Terminada esta fase, para cada grupo de coisas, põe tudo de volta no guarda-roupa, organizando cada compartimento e gaveta (arrumando as camisetas, separando as camisas de mangas curtas e mangas compridas, vestidos, calças jeans e sociais etc.), um a um.

Outra pessoa opta por remover do guarda-roupa apenas o necessário e já partir para a organização. Separa, por exemplo, as roupas íntimas, meias e shorts, e já os arruma nas gavetas; separa as calças e faz sua arrumação nos cabides; junta as blusas nas gavetas de baixo; junta as roupas de cama na parte de cima. Cada item é praticamente terminado antes que outro seja iniciado.

Todas as alternativas (as duas acima e ainda outras) levam à arrumação pretendida, porém de forma diferente. Cada um pode justificar sua solução como melhor que outras, mas isso é outra questão.

Em suma, um algoritmo também pode ser abordado de forma diferente e várias soluções distintas podem levar à produção de respostas idênticas, todas resolvendo o problema.

4.3 Estratégia de abordagem para desenvolvimento de algoritmos

Para o caso de algoritmos, uma forma (que não é a única) de se abordar uma proposição de solução na forma de algoritmo é a chamada **abordagem *top-down***. O termo, que poderia ser traduzido "abordagem de cima para baixo", pode ser entendido como uma abordagem do **mais geral** para o **mais específico**.

Veja-se a primeira alternativa para a arrumação do guarda-roupa sugerida na seção anterior (a que sugere tirar tudo e, posteriormente, arrumar). Ela pode ser descrita pelos seguintes passos, de forma mais geral:

1) Remova todas as roupas, separando-as sobre a cama segundo grupos similares (camisetas, camisas, vestidos, roupas de inverno etc.)
2) Para cada grupo formado, organize-o (por cor, por uso ou outra forma) e o acondicione no local definitivo (gavetas, cabides).

Deve ser observado que a solução acima propõe uma solução coerente para o problema, mesmo já pensando na questão de algoritmos formais, que indica que o segundo passo somente é executado após o primeiro ter sido inteiramente completado.

Uma solução melhor, para ser dada a uma empregada ou ao caçula da família que foi "condenado" a fazer a organização, deveria ser mais específica, detalhando os procedimentos e especificando melhor as ações e critérios.

4.3.1 Abordagem *top-down*

Na forma de um algoritmo, que não tem pretensão nenhuma de ser executado por um computador, a solução para o problema de arrumação pode ser observada no Algoritmo 4-1.

Algoritmo 4-1

```
1  { solução geral e não computacional para a arrumação de um guarda-roupa
2    em situação calamitosa }
3
4  algoritmo
5        remova todas as roupas, separando-as sobre a cama segundo grupos
6                  similares
7        organize cada grupo formado e o acondicione nos locais apropriados
8  fim-algoritmo
```

Estabelecida a solução global, que deve ser completa e coerente, um detalhamento maior pode ser especificado, reescrevendo-se alguns comandos do algoritmo. Deve-se observar que, nesta nova versão (e nas subseqüentes), a notação de comentários fará parte da apresentação. O Algoritmo 4-2 mostra o resultado.

Algoritmo 4-2

```
1  { solução geral e não computacional para a arrumação de um guarda-roupa
2    em situação calamitosa – VERSÃO 2 }
3
4  algoritmo
5        { remoção de todas as roupas, separando-as sobre a cama
6          segundo grupos similares }
7        para cada peça de roupa,
8             pegue-a do guarda-roupa
9             coloque-a no grupo correto
10
11       { organização de cada grupo formado e acondicionamento
12         nos locais apropriados }
13       para cada grupo de roupas,
14            organize o grupo
15            para cada peça de roupa do grupo,
16                 transfira a peça para o local apropriado
17 fim-algoritmo
```

Mesmo assim, alguns pontos ainda podem precisar de detalhamento, o que leva a outra versão, expressa pelo Algoritmo 4-3.

Algoritmo 4-3

```
1  { solução geral e não computacional para a arrumação de um guarda-roupa
2    em situação calamitosa – VERSÃO 3 }
3
4  algoritmo
5        { remoção de todas as roupas, separando-as sobre a cama
6          segundo grupos similares }
7        para cada peça de roupa,
8             pegue-a do guarda-roupa
9             { colocação no grupo correto }
10            para cada grupo que já existe,
11                 se a peça pertencer ao grupo,  { for similar às demais }
12                      coloque-a no grupo,
13            se a peça não foi colocada em nenhum grupo,   { não existe }
14                 crie um novo grupo com a peça
15
16       { organização de cada grupo formado e acondicionamento
17         nos locais apropriados }
18       para cada grupo de roupas,
19            { organização do grupo }
20            caso o grupo seja:
21                 de camisetas, separe por cor
22                 de camisas, calças ou vestidos,
23                           separe por tipo { social/esporte }
24                 de saias ou blusas, separe por idade { novas/velhas }
25                 demais grupos, separe por cor
```

```
26            para cada peça de roupa do grupo,
27                 { transferência da peça para o local apropriado }
28                 procure uma gaveta ou cabide disponível, conforme
29                                 a peça
30                 coloque a peça no local, obedecendo a organização
31 fim-algoritmo
```

O Algoritmo 4-3 mostra uma versão bem mais detalhada do procedimento, aproveitando em cada passo o que foi detalhado nos passos anteriores. Naturalmente, o detalhamento pode ser **refinado** o quanto for necessário, até que esteja completamente compreensível por quem vai executá-lo (mesmo que este "alguém" seja um computador capaz de comandar um robô dotado de visão e bom-gosto e que seja hábil para dobrar as roupas sem sujá-las de graxa).

Esta estratégia ilustra o procedimento *top-down*. Inicialmente é feito um algoritmo mais geral, que descreve completamente a solução, mas sem todos os detalhes. Esta proposta inicial é refinada sucessivamente (e nem é preciso que os detalhamentos sejam feitos na ordem em que aparecem) até que, na versão final, esteja pronta para a execução.

4.3.2 Abordagem do problema

Quando se pensa em algoritmos, eles estão no contexto de um problema, como visto na Unidade 1. É comum que, mesmo usando a técnica de **refinamentos sucessivos** (vista em detalhes na próxima seção), a primeira coisa que o desenvolvedor faz é começar a escrever o algoritmo. Este é seu primeiro erro.

Há um conjunto de passos que devem ser seguidos para a elaboração de um algoritmo. São eles:

1) Entender a proposta do problema;
2) Buscar e entender uma solução não algorítmica para o problema;
3) Propor um algoritmo que traduza para uma seqüência de passos a solução já encontrada;
4) Testar os vários aspectos do algoritmo elaborado para verificar se não há problemas e se realmente produz a solução esperada;
5) Verificar o algoritmo pronto e considerar se algumas das etapas podem ser feitas de forma mais eficiente.

Nota-se que elaborar o algoritmo é apenas o terceiro passo, e não o primeiro. Os dois primeiros passos traduzem uma idéia de pensar antes no assunto, antes de resolvê-lo.

Deste modo, **entender o problema** significa verificar o que é solicitado e o que deve ser produzido como resposta. Devem ser levantados os dados disponíveis e, a partir deles, quais dados devem ser produzidos. Para exemplificar a abordagem como um todo, é proposto um problema relativamente simples: "Um professor de física trabalha com balística e, em um determinado momento, precisa que seja elaborado um algoritmo para determinar se um projétil colidirá ou não com um obstáculo (muro ou

prédio) presente em seu caminho. A Figura 4-1 mostra um esquema da situação, juntamente com alguns dados disponíveis. Sabe-se a velocidade inicial no lançamento e o ângulo, em graus, em relação ao solo. Tem-se também a distância do objeto do ponto de lançamento, além de sua altura e espessura. Deve-se considerar a superfície da Terra e sem influência de outros fatores, como vento."

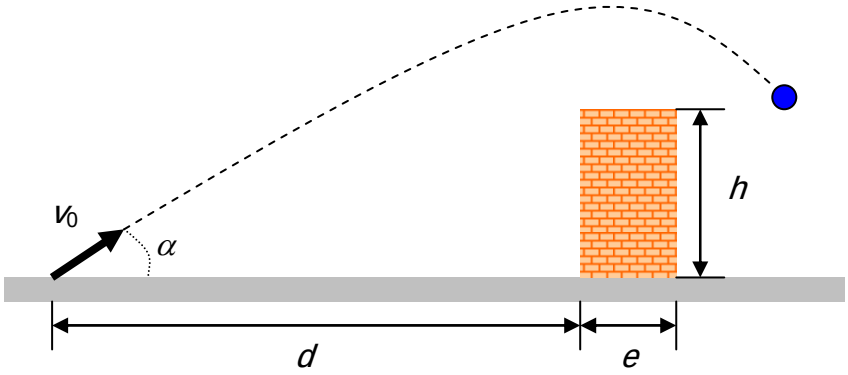


Figura 4-1. Problema de lançamento de objeto.

Para quem vai desenvolver o algoritmo, o primeiro passo é entender o que se espera como resultado. Neste caso, não interessa o ponto máximo da trajetória, a distância da origem que o projétil cai nem sua velocidade quando passa sobre o objeto. A resposta esperada pelo problema pode vir em duas sentenças: "sim, houve colisão" e "não, não houve colisão". Este momento é crucial, pois determina todo o desenvolvimento do algoritmo.

O segundo passo é se ter uma **solução não algorítmica**. Para isso, quem vai escrever o algoritmo não precisa entender de física ou saber qualquer uma das fórmulas envolvidas. Esta não é sua função. Por exemplo, uma apostila de cursinho pode ser consultada ou um especialista (ou amigo) pode ser ouvido. Neste exemplo, a solução que se obtém (independentemente da origem), é que somente haverá colisão se a altura do projétil nos limites do objeto for inferior a sua altura (ou seja, nas distâncias *d* e *d* + *e* do ponto de lançamento. Para isso, é preciso calcular o tempo que o projétil leva para atingir os limites do objeto na horizontal e calcular sua altura nestes momentos. Fisicamente, o tempo até o projétil estar a uma distância s_h do ponto de lançamento é $t = \frac{s_h}{v_0 \cos\alpha}$. Portanto, os tempos são $t_d = \frac{d}{v_0 \cos\alpha}$ e $t_{d+e} = \frac{d+e}{v_0 \cos\alpha}$. Nestes momentos, a altura é descrita pelo movimento vertical, dado por $s_v = v_0 \sin\alpha \cdot t - g\frac{t^2}{2}$. Novamente, observa-se que esta informação não é especialidade de quem escreve o algoritmo, mas este precisa entender o que as fórmulas indicam, mesmo que não haja qualquer conhecimento sobre o que elas signifiquem fisicamente. Assim, com estas equações, é possível calcular a altura do projétil nos dois momentos críticos e verificar se estão acima ou não do objeto que precisa ser transposto.

O terceiro passo é, finalmente, **propor o algoritmo**. Uma versão inicial do algoritmo é apresentada como Algoritmo 4-4.

Algoritmo 4-4

```
1  { verificação se um projétil, lançado de um ângulo de α graus em
2    relação ao solo colide ou não com um objeto situado em seu caminho,
3    sabendo-se a distância do lançamento e a altura e espessura do objeto;
4    o resultado produzido é verdadeiro se há colisão ou falso caso
5    contrário }
6  { A Figura 4-1 deve ser consultada para melhor referência (apostila
7    de Construção de Algoritmos, página 4-5) }
8
9  algoritmo
10       declarações
11
12       obtenha o ângulo, v0, a distância do objeto, sua espessura e altura
13       calcule os momentos em que o projétil passa pelos limites
14                           do objeto
15       calcule a altura do projétil nestes dois momentos
16       verifique se em algum destes dois momentos o projétil não
17                           está acima da altura do objeto
18       escreva o resultado
19 fim-algoritmo
```

Refinando o algoritmo para chegar no nível dos comandos vistos na Unidade 2, uma nova versão é obtida (Algoritmo 4-5).

Algoritmo 4-5

```
1  { verificação se um projétil, lançado de um ângulo de α graus em
2    relação ao solo colide ou não com um objeto situado em seu caminho,
3    sabendo-se a distância do lançamento e a espessura e altura do objeto;
4    o resultado produzido é verdadeiro se há colisão e falso caso
5    contrário }
6  { A Figura 4-1 deve ser consultada para melhor referência (apostila
7    de Construção de Algoritmos, página 4-5) }
8
9  algoritmo
10       { declarações }
11       declare
12            ângulo, velocidadeInicial, distância, espessura, altura,
13            momento1, momento2, alturaProjétil1, alturaProjétil2: real
14            colidiu: lógico
15
16       { obtenção do ângulo, v0, distância do objeto,
17         sua espessura e altura }
18       leia(ângulo, velocidadeInicial, distância, espessura, altura)
19
20       { cálculo dos momentos em que o projétil
21         passa pelos limites do objeto }
```

```
22      momento1 ← distância/(velocidadeInicial * cos(ângulo * 3,1415/180))
23      momento2 ← (distância + espessura)/
24                          (velocidadeInicial * cos(ângulo * 3,1415/180))
25                               { 3,1415 é aproximação para π
26                                 na conversão para radianos }
27
28      { cálculo da altura do projétil nestes dois momentos }
29      alturaProjétil1 ← velocidadeInicial * sen(ângulo * 3,1415/180) *
30                                      momento1 – 9,8 * pot(momento1, 2)/2
31      alturaProjétil2 ← velocidadeInicial * sen(ângulo * 3,1415/180) *
32                                      momento2 – 9,8 * pot(momento2, 2)/2
33
34      { verificação se em algum desses dois momentos o projétil não
35        está acima da altura do objeto }
36      colidiu ← alturaProjétil1 ≤ altura e alturaProjétil2 ≤ altura
37
38      { escrita do resultado }
39      escreva("Houve colisão?", colidiu)  { escreve "V" ou "F" }
40  fim-algoritmo
```

Escrito o algoritmo, deve-se proceder a sua **avaliação se está correto**. Para isso, é preciso verificar para algumas situações se o resultado produzido é o esperado. Esta avaliação é feita pelo acompanhamento do algoritmo passo a passo, realizando seus comandos e verificando se não há problemas. Para a "versão 2" (Algoritmo 4-5), nota-se que, caso uma das alturas do projétil esteja igual ou menor que a altura do objeto (linha 35), o operador **e** somente produz verdadeiro se ambas estiverem acima do objeto, proporcionando um resultado incorreto. Desta forma, a linha 35 é reescrita usando o operador **ou**.

```
35      colidiu ← alturaProjétil1 ≤ altura ou alturaProjétil2 ≤ altura
```

Um cuidado especial deve ser tomado nesta fase, pois o algoritmo "parecer" funcionar corretamente é muito diferente dele realmente realizar sua tarefa a contento. Muitas vezes, mesmo um algoritmo bem testado acaba por deixar passar algumas situações nas quais há falha em sua lógica de execução.

Finalmente, verificado que não há mais problemas com os cálculos e os resultados produzidos, o algoritmo pode ser **avaliado se pode ser melhorado**. Esta etapa pode ser mesclada em certos momentos com o desenvolvimento (passo anterior). No exemplo apresentado, o resultado pode ser considerado satisfatório. Uma possível melhoria poderia ser a redução do número de comandos, evitando o cálculo das variáveis **momento1** e **momento2** pela substituição de suas expressões diretamente no cálculo das duas alturas do projétil. Como pode haver certo comprometimento da clareza da solução, optou-se, neste caso, por manter a versão atual, embora a alteração proposta seja defensável.

4.3.3 Resolução de problemas por refinamentos sucessivos

Normalmente o desenvolvimento do algoritmo é feito com papel e lápis, e não em um computador ou editor de textos. Por isso, a abordagem *top-down* para a construção dos algoritmos vem associada a uma notação para indicar os detalhamentos, chamados **refinamentos**, de modo que não seja preciso reescrever, a todo momento, todas as linhas das versões anteriores do algoritmo.

Esta notação é apresentada por meio de um novo exemplo de desenvolvimento. Assim, deve ser considerado o seguinte problema: "A média final de um aluno da disciplina Cálculo Numérico é calculada com peso 6 para a média de provas (considerando 2 provas), peso 2 para a média de trabalhos práticos (4 trabalhos práticos) e peso 2 para o único seminário. As médias de provas e de trabalhos são médias aritméticas simples. É preciso escrever um algoritmo para, dadas as notas de provas, dos trabalhos e do seminário, calcular a média final e determinar se o aluno foi aprovado (média final maior ou igual a 6)."

Embora seja um exemplo bastante simples, o objetivo aqui é ilustrar a notação dos refinamentos, os quais passarão a ter maior importância da Unidade 5 em diante.

O Algoritmo 4-6 mostra uma solução para o problema proposto, delineando a solução de forma mais geral.

Algoritmo 4-6

```
1  { calcular a média final e determinar a aprovação (média final >= 6),
2    dadas as notas de 2 provas, 4 trabalhos e um seminário; as médias de
3    provas e de trabalhos são médias aritméticas simples e a composição
4    de pesos para a média final é 6 para média de provas, 2 para média de
5    trabalhos e 2 para seminário }
6
7  algoritmo
8        { ref declarar variáveis }
9
10       { ref obter todas as notas }
11       { ref calcular as médias de provas e de trabalhos }
12       { ref calcular a média final e verificar aprovação }
13
14       { apresentação dos resultados}
15       escreva(médiaFinal, aprovado)
16 fim-algoritmo
```

Pode-se observar que a estrutura de delineamento geral da solução é a mesma já vista neste texto. A diferença é que, ao invés do itálico usado nas outras versões, é usada a notação de comentários e um indicador **ref** para representar uma instrução ou passo que ainda precisa ser detalhada. Pode ser visto também, que nas linhas 14 e 15 optou-se por não usar refinamentos, pois se julgou desnecessário.

Os detalhamentos necessários podem ser feitos, agora, apenas para os refinamentos especificados. O Algoritmo 4-7 mostra como fica o refinamento para a linha 10 e o Algoritmo 4-8 para a linha 12.

Algoritmo 4-7

```
1  refinamento { obter todas as notas }
2        leia(prova1, prova2) { 2 provas }
3        leia(trabalho1, trabalho2, trabalho3, trabalho4) { 4 trabalhos }
4        leia(seminário) { 1 seminário }
5  fim-refinamento
```

Algoritmo 4-8

```
1  refinamento { calcular a média final e verificar aprovação }
2        médiaFinal ← 0,6 * médiaProvas + 0,2 * (médiaTrabalhos + seminário)
3        aprovado ← médiaFinal ≥ 6,0
4  fim-refinamento
```

Esta é uma notação formal que permite organizar tanto o raciocínio quanto a documentação. Todos os elementos que foram indicados para refinamento devem ser transformados em comentários quando o algoritmo final for "passado a limpo" em sua versão final. Em outras palavras, se a linha de raciocínio indicou um refinamento necessário, é porque, de alguma forma, ele é importante como um elemento de organização e merece, portanto, o comentário.

Naturalmente, um refinamento pode conter outros itens que serão refinados e assim por diante, criando uma hierarquia de **refinamentos sucessivos**. Um problema complexo se torna, deste modo, uma coleção de problemas mais simples.

Voltando ao exemplo, a incorporação dos dois refinamentos realizados (Algoritmo 4-7 e Algoritmo 4-8) deve produzir o Algoritmo 4-9.

Algoritmo 4-9

```
1  { calcular a média final e determinar a aprovação (média final >= 6),
2    dadas as notas de 2 provas, 4 trabalhos e um seminário; as médias de
3    provas e de trabalhos são médias aritméticas simples e a composição
4    de pesos para a média final é 6 para média de provas, 2 para média de
5    trabalhos e 2 para seminário }
6
7  algoritmo
8        { ref declarar variáveis }
9
10       { obter todas as notas }
11       leia(prova1, prova2) { 2 provas }
12       leia(trabalho1, trabalho2, trabalho3, trabalho4) { 4 trabalhos }
13       leia(seminário) { 1 seminário }
14
15       { ref calcular as médias de provas e de trabalhos }
16
17       { calcular a média final e verificar aprovação }
```

```
18       médiaFinal ← 0,6 * médiaProvas + 0,2 * (médiaTrabalhos + seminário)
19       aprovado ← médiaFinal ≥ 6,0
20
21       { apresentação dos resultados}
22       escreva(médiaFinal, aprovado)
23 fim-algoritmo
```

Podem ser observadas a inserção dos comandos refinados nos locais apropriados e também a remoção da indicação **ref**, com a manutenção do comentário.

Outro ponto importante é que os refinamentos não precisam, necessariamente, ser feitos em ordem e, dentro de cada um deles, as variáveis que forem necessárias vão sendo criadas. É claro que se deve manter a coerência entre as variáveis comuns aos refinamentos, como é o caso da variável lógica **aprovados**, que já aparece na versão inicial e é usada coerentemente nos refinamentos.

Nesta linha de raciocínio, o último refinamento que deve ser feito é o das declarações, pois somente após terminados todos os outros é que se pode saber todas as variáveis que serão necessárias.

4.4 Considerações finais

A esta altura, é comum que surja uma dúvida sobre a real necessidade dos refinamentos. A primeira razão é que o exemplo apresentado e também os próximos exemplos na Unidade 5 ainda são relativamente simples. Mas é essencial que esta metodologia, embora pareça enfadonha no início, seja incorporada ao processo de raciocínio no desenvolvimento. Infelizmente, embora alguns tenham o dom de resolver problemas com facilidade, depende de cada um não rejeitar um método que será de grande utilidade para a grande maioria.

Desta Unidade, alguns preceitos de grande importância foram abordados:

- Para fazer um algoritmo, a fase de escrita é somente a terceira etapa; é necessário antes disso compreender o problema e saber como ele pode ser resolvido (não algoritmicamente);
- Ao se escrever um algoritmo, é imprescindível que uma visão geral coerente de toda a solução esteja bem definida; esta é a chamada versão inicial ou esboço inicial, a qual pode conter os itens que deverão ser refinados;
- Utilizar os refinamentos auxilia no processo de raciocínio sobre a elaboração da solução, pois implica em quebrar um problema mais complexo em outros mais simples e, em conseqüência, mais facilmente tratados;
- A quantidade de refinamentos que serão necessários depende de quem está escrevendo o algoritmo; o grau de maturidade e a prática tornam diversos problemas mais simples de ser tratados com o tempo;

- A prática é o principal modo para que os conhecimentos apresentados nesta seção sejam corretamente aplicados e realmente se tornem tão úteis quanto seu potencial permite.

Unidade 5 - Comandos condicionais

5.1 Primeiras palavras

A grande maioria dos problemas que são enfrentados na prática não possui solução linear. Quase sempre é necessário fazer algo somente se uma dada condição ocorrer. Desta forma, esta Unidade trata o caso da execução condicional de comandos sob o aspecto formal dos algoritmos.

Esta ferramenta (a execução condicional) é de extrema importância e, portanto, deve ser observada com atenção. Juntamente com este novo tema, os conceitos sobre desenvolvimento estruturado e sobre a organização do desenvolvimento, abordados na Unidade 4, serão retomados.

5.2 Situações de decisão

Na Unidade 1, ao se fazer a introdução aos algoritmos, foi citado que um algoritmo (não tão formal quanto o que este texto apresenta) pode ser aquele especificado como instruções para cadastrar um novo contato na agenda do celular. Este algoritmo poderia ser ilustrado como: "pressione [Menu], selecione [Agenda] e opte por [Novo contato]; preencha o nome e o telefone; pressione [Ok] para encerrar ou [Off] para cancelar".

As ações condicionais podem ser observadas em vários momentos deste procedimento, como na decisão se o telefone cadastrado será colocado na parte reservada para o número fixo, para o comercial ou para o móvel; ou ainda, no final, na diferença do resultado caso a inclusão seja efetivada ou cancelada.

Estas decisões dependem de certas condições, como: o número do telefone que deve ser cadastrado é o fixo ou o celular? no fim das contas, vale realmente a pena cadastrar este contato? Ou ainda, de outras: o saldo está positivo? ainda há contas para pagar? a luz do quarto ficou acesa? há roupa no varal? será que vai chover?

Também nos algoritmos existem condições que alteram o resultado das ações em função das situações que forem ocorrendo.

5.3 Alteração condicional do fluxo de execução

Os comandos condicionais são as instruções que o algoritmo tem para verificar uma situação e alterar o fluxo de execução em função desta situação. Como observado nas Unidades anteriores, o fluxo de execução é sempre seqüencial, seguindo a ordem das instruções. Esta regra não se altera, pois o comando condicional é considerado uma única instrução e, em seu "interior" são especificados os conjuntos de instruções que dependem de condição.

Com vistas à versatilidade, os comandos condicionais se apresentam em dois formatos: o **condicional padrão** e o de **seleção múltipla**.

O primeiro é o comando condicional "comum" e é mais importante que o de seleção múltipla, pois tem uso mais geral. Também é possível

escrever um comando de seleção múltipla usando os condicionais padrão, de modo os de seleção múltipla podem ser considerados "supérfluos", embora sejam bastante práticos em muitas (porém restritas) situações.

5.3.1 Condicionais simples e completos

O **comando condicional completo** é escrito no formato indicado abaixo, sendo que é especificada uma condição lógica e dois grupos de comandos.

```
se expressão_lógica então
     conjunto_de_comandos_caso_verdadeiro
senão
     conjunto_de_comandos_caso_falso
fim-se
```

A lógica de execução do comando é simples: a avaliação da **expressão lógica** é feita e seu resultado calculado; se for **verdadeiro**, somente o primeiro conjunto de comandos é executado (especificados no **então**); se o resultado for **falso**, somente o segundo conjunto de comandos é executado (especificados pelo **senão**). A delimitação dos conjuntos de comandos é feita do **então** até o **senão** para a parte "se verdadeiro" e do **senão** até o **fim-se** para a parte "se falso".

Por exemplo, o seguinte problema pode ser considerado: "A média final de um aluno é calculada pela média aritmética simples de duas provas e sua aprovação depende desta média ter valor maior ou igual a 6. Escreva um algoritmo que, dadas as notas de provas, escreva a média final e uma mensagem que indique se houve ou não aprovação.".

A partir da abordagem da Unidade 4, o primeiro passo é entender o problema. Como o caso é simples, conclui-se que os dados de entrada são somente duas notas (assumidas no intervalo de 0 a 10, incluindo os limites) e o resultado esperado é um valor real (a média, no mesmo intervalo) e uma mensagem equivalente a "aprovado" ou "reprovado".

O segundo passo é conhecer a solução não algorítmica, que é trivial, pois basta somar as notas e dividir por 2 e, então, comparar este resultado com 6.

O terceiro passo é, finalmente, a proposição do algoritmo. Cada aluno deve se perguntar, neste momento, se cedeu à tentação de começar a escrever o algoritmo no primeiro passo!

Considerando a simplicidade do algoritmo, a proposta inicial não inclui refinamentos, sendo apresentada no Algoritmo 5-1.

Algoritmo 5-1

```
1  { cálculo da média final de um aluno, feita pela média aritmética de
2    duas provas, e escrever mensagem indicando aprovação ou reprovação }
3
4  algoritmo
5       declare notaProva1, notaProva2, média: real
6
7       { leitura das notas }
```

```
8        leia(notaProva1, notaProva2)
9
10       { cálculo da média }
11       média ← (notaProva1 + notaProva2)/2
12
13       { resultados }
14       escreva(média)
15       se média >= 6 então
16            escreva("Aprovado :)...")
17       senão
18            escreva("Reprovado :(...")
19       fim-se
20 fim-algorimo
```

O comando condicional **se** é um comando longo e ocupa as linhas de 15 a 19. A apresentação visual do comando é importante, principalmente à medida que os problemas se tornarem mais complexos. Sugere-se fortemente adotar o formato indicado no exemplo, com o alinhamento das cláusulas **se**, **senão** e **fim-se**.

No exemplo, a condição é dada pela avaliação da expressão relacional (se a média é superior ou igual a 6), que resulta necessariamente em **verdadeiro** ou **falso**. Se o resultado for **verdadeiro**, o único comando após o **senão** é executado e a mensagem "Aprovado" é escrita; caso contrário, a mensagem é "Reprovado". (Uma confrontação entre este algoritmo e o Algoritmo 4-9 pode ser feita, principalmente em relação ao modo como o problema de verificação da aprovação é feito.)

Muitas vezes o comando **se** não precisa da parte do **senão**, que pode ser omitida. O **formato simples do comando condicional** é apresentado abaixo.

```
        se expressão_condicional então
             conjunto_de_comandos_caso_verdadeiro
        fim-se
```

Um exemplo de uso do comando condicional simples é ilustrado no Algoritmo 5-2, que é apenas um trecho de um algoritmo do qual as partes restantes foram omitidas.

Algoritmo 5-2

```
1        (parte omitida)
2
3        leia(saldo)
4        se saldo < 0 então
5             leia(taxaJuros, númeroDias)  { só se o saldo for devedor }
6             valorDevedor ← abs(saldo) * taxaJuros * númeroDias
7             saldo ← saldo – valorDevedor
8        fim-se
9
10       (parte omitida)
```

No exemplo, se o valor do saldo não for devedor, nem a taxa de juros nem o número de dias que a conta ficou negativa são solicitados, além de não ser preciso descontar este valor do saldo.

5.3.2 Aninhamento de comandos

Os comandos condicionais são a primeira forma de comandos que possuem comandos "subordinados". Em outras palavras, o condicional é um comando que incorpora outros comandos.

Os mesmos princípios de execução que valem para o algoritmo são obedecidos para o conjunto de comandos subordinados, como execução seqüencial e execução de um comando somente após o anterior ter sido encerrado.

O conjunto de comandos "internos" a um **se** pode incluir qualquer tipo de comando, como atribuições, leituras ou outros comandos condicionais. Não há, portanto, restrições. Quando há comandos condicionais dentro de outros comandos condicionais, é dito que os comandos estão **aninhados**. Uma vantagem da notação utilizada é que não existe confusão entre o **senão** de um **se** com o dos outros.

No Algoritmo 3-3 (página 3-9) é apresentada uma solução para verificar a classificação de um triângulo em função dos comprimentos de seus lados. Abaixo, o Algoritmo 5-3 mostra uma forma alternativa de resolver o mesmo problema.

Algoritmo 5-3

```
1  { determinar, dados os comprimentos dos lados de um triângulo, se o
2    triângulo pode ser formado (se existe) e qual o tipo dele (eqüilátero,
3    isósceles ou escaleno) }
4
5  algoritmo
6        declare
7             lado1, lado2, lado3: real
8             classificação: literal
9
10       { leitura dos comprimentos dos lados }
11       leia(lado1, lado2, lado3)
12
13       se lado1 ≥ lado2 + lado3 ou lado2 ≥ lado1 + lado3 ou { existe? }
14                                    lado3 ≥ lado1 + lado2 então
15            escreva("Os lados fornecidos não formam um triângulo")
16       senão
17            se lado1 = lado2 e lado2 = lado3 então  { lados iguais? }
18                 classificação ← "Eqüilátero"
19            senão
20                 se lado1 = lado2 ou lado1 = lado3 ou    { isósceles? }
21                                    lado2 = lado3 então
22                      classificação ← "Isósceles"
23                 senão
```

```
24                      classificação ← "Escaleno"
25                  fim-se
26              fim-se
27
28              { escrita da classificação }
29              escreva(classificação)
30          fim-se
31  fim-algoritmo
```

Devem ser observados, inicialmente, os limites de cada comando **se**, finalizado cada um com seu respectivo **fim-se**. A primeira verificação (linha) checa se os lados podem não compor um triângulo; se não for possível, a única coisa feita é escrever a mensagem indicativa, rejeitando a execução de todos os demais comandos (que estão no **senão**). A expressão verificada na linha 16 resulta em **verdadeiro** somente se todos os lados forem iguais e, se este comando chegou a ser executado, então o triângulo tem que existir e a verificação é coerente. Se o comando **se** da linha 19 for executado, sabe-se que o triângulo não é eqüilátero e, portanto, se houver dois lados iguais (quaisquer que sejam), então o triângulo é isósceles. Caso esta última verificação resulte em **falso**, então o triângulo não é eqüilátero e nem isósceles; portanto não é mais necessária outra verificação para concluir que é escaleno, o que permite a atribuição direta indicada na linha 23. Pode-se notar que todos os comandos condicionais utilizados contém um único comando tanto na parte do **então** quanto do **senão**, exceto o iniciado na linha 12 (condição de inexistência), cujo **senão** tem dois comandos: um condicional e um comando de escrita. Este comando de saída da linha 28 apresenta na tela o valor atribuído à variável **classificação** e somente é executado, naturalmente, caso o triângulo exista e após ser verificado seu tipo.

5.3.3 Comandos de seleção múltipla

Uma forma alternativa de fazer verificações é usando do **comando de seleção múltipla**, também conhecido por **caso**.

```
caso expressão_inteira seja
     lista_de_seleção
senão
     conjunto_de_comandos_se_não_há_coincidências
fim-caso
```

O comando **caso** funciona exclusivamente com uma expressão inteira, cujo valor resultante é avaliado para ser confrontado com cada item da **lista de seleção**. Se não houver coincidências, os comandos especificados na parte do **senão** são executados.

A lista de seleção é composta por um ou mais itens, que obedecem ao formato abaixo.

```
lista_de_constantes: conjunto_de_comandos
```

Nesta especificação, a **lista de constantes** é uma lista formada por constantes inteiras, que serão confrontadas com o resultado da expressão inteira. Caso haja uma coincidência (igualdade), o **conjunto de comandos** especificado será executado; se não houver, os comandos serão ignorados.

A exemplificação do comando é a melhor forma de entender seu funcionamento. O Algoritmo 5-4 apresenta um algoritmo que, dada uma idade, classifica a pessoa segundo algumas faixas etárias.

Algoritmo 5-4

```
1  { classificação da faixa etária segundo um critério arbitrário }
2
3  algoritmo
4        declare idade: inteiro
5
6        { leitura da idade }
7        leia(idade)
8
9        { classificação }
10       caso idade seja
11         0:      escreva("bebê")
12         1..10:  escreva("criança")
13        11..14:  escreva("pré-adolescente")
14        15..18:  escreva("adolescente")
15        19..120: escreva("adulto")
16       senão
17            escreva("Idade inválida ou sem classificação definida")
18       fim-caso
19 fim-algoritmo
```

A expressão inteira usada no comando **caso** é formada apenas pela variável **idade**, que é inteira. Portanto, o valor da variável será confrontado com cada item da lista de seleção (cada linha é um item). A primeira alternativa indica que o valor vai ser comparado com 0; se for igual, é escrita a mensagem "bebê" e o comando se encerra (como é o último comando do algoritmo, este também termina). As demais opções usam indicação de intervalo: 1..10 significa de 1 até 10, ou seja, 1, 2, 3..., 9 e 10. Se houver coincidência da expressão (idade) com qualquer um destes valores, então o comando especificado é executado e o **caso** é terminado. As demais linhas também especificam intervalos (de 11 a 14, de 15 a 18 e de 18 a 120). Finalmente, o comando do **senão** somente será executado quando nenhuma das coincidências ocorrer (ou seja, se idade tiver valor menor que zero ou maior que 120).

É importante salientar que é considerado um erro se um mesmo valor constante for colocado em mais que um item da lista de seleção, da mesma forma que é um erro usar variáveis no lugar das constantes (não há como garantir que não haja valores iguais em opções diferentes). Também não é necessário que todo um intervalo de valores seja especificado, pois o **senão**

existe para este caso. Assim, as constantes especificadas podem ser "esparsas".

Como ocorre nos comandos condicionais, também na seleção múltipla a parte do **senão** é optativa, o que indica que, se não houver qualquer coincidência, nada será executado e o comando subseqüente ao **caso** ganha vez.

Na relação de constantes, além da constante isolada e do intervalo, é possível especificar outras combinações, usando-se a vírgula como separador. O Algoritmo 5-5 exemplifica algumas possibilidades válidas, dadas fora de qualquer contexto, servindo apenas como ilustração.

Algoritmo 5-5

```
1       { ... }
2       caso valor1 + valor2/3 - valor3/2 seja
3         -100..-1:         escreva("alguns valores negativos")
4         2, 3, 5, 7:       escreva("primo menor ou igual a 10")
5         1, 4, 6, 8..10:   escreva("não primo menor ou igual a 10")
6         15:               se valor1 > 0 então
7                             escreva("alguma coisa")
8                           senão
9                             escreva("outra coisa")
10                          fim-se
11        20..30:           leia(idade)
12                          se idade < 18 então
13                            escreva("Menor de idade")
14                          senão
15                            escreva("Já atingiu a maioridade")
16                          fim-se
17        16, 40, 80:       escreva("16, 40 ou 80...")
18      fim-caso
19      { ... }
```

Embora, à primeira vista, o comando pareça bastante versátil (e o é, em alguns casos), sua aplicação é bastante limitada. Inicialmente, é possível especificar apenas expressões inteiras. Outro ponto é que, nas constantes especificadas, somente é possível comparar por igualdade, o que torna inviável usar o **caso** para verificar se uma expressão é menor que zero, por exemplo. É incoerente, portanto, escrever um item da lista de seleção no formato "> 0: escreva("positivo")", pois não faz sentido que um valor inteiro seja **igual** a "> 0".

5.3.4 Testes de mesa

O acompanhamento de um algoritmo já foi mencionado como a quarta fase ao se abordar um problema (seção 4.3.2). Neste contexto, *acompanhar* significa executar os comandos passo a passo e anotar as modificações impostas por eles nas variáveis. O nome dado a este acompanhamento manual é **teste de mesa**.

Como acontece com os algoritmos, também para os testes de mesa há técnicas que facilitam o acompanhamento da execução.

A estratégia de teste de mesa indicada nesta seção é bastante simples, porém muito útil. Como um algoritmo especifica suas variáveis antecedendo os comandos, sempre é possível saber quais as variáveis que serão usadas. Deste modo, estas variáveis podem ser o cabeçalho de uma espécie de tabela de valores. Cada vez que um comando implica na alteração de um valor de uma ou mais variáveis, seu valor atual é riscado e o novo valor escrito logo abaixo. Com isso, torna-se bastante simples saber o valor mais recente (atual) de uma variável apenas consultando a tabela.

Considerando apenas alguns comandos apresentados no Algoritmo 5-6, a Figura 5-1 mostra histórico do acompanhamento das instruções. No exemplo, todas as variáveis são inteiras.

Algoritmo 5-6

```
1       { ... }
2       valor1 ← 10
3       valor2 ← valor1 * 5
4       valor3 ← valor1 + valor2/2
5       se valor3 < valor1 + valor2 então
6            escreva(valor3, valor1 + valor2)
7       fim-se
8       valor1 ← valor1 + valor2
9       valor3 ← valor3/2
10      escreva(valor1, valor2, valor3)
11      { ... }
```

valor1	valor2	valor3	saída
~~10~~	~~50~~	~~35~~	~~35~~ 60
60		17	60 50 17

Figura 5-1. Teste de mesa para o trecho especificado em Algoritmo 5-6.

A prática no uso do teste de mesa é importante para que pequenos erros conceituais ou falhas no entendimento de alguns detalhes sejam detectados e, em conseqüência, corrigidos. Enfatiza-se que todos adotem como hábito fazer os testes de mesa, mesmo no caso de algoritmos mais simples.

5.4 Considerações finais

Esta Unidade apresentou uma das ferramentas de controle do fluxo de execução dos algoritmos, os comandos condicionais, sem a qual é praticamente impossível escrever um algoritmo. O condicional **se** é um comando simples em sua forma, mas usá-lo requer treinamento e um excelente conhecimento das expressões lógicas.

Fazer exercícios e testá-los (com os testes de mesa), avaliando cada expressão lógica com cuidado auxilia no entendimento do comando e permite um bom domínio desta importante estrutura.

O comando de seleção múltipla, embora possua suas limitações, também é importante, principalmente por permitir uma solução simples, clara e, em geral, elegante.

Finalmente, ressalta-se que os testes de mesa são essenciais para que o funcionamento dos algoritmos seja plenamente compreendido. É comum que alguns alunos achem os algoritmos um tanto quanto nebulosos. Se este for o caso, significa que os conceitos não foram compreendidos e este é o momento de tentar recuperar esta deficiência. O exercício com os testes de mesa acaba auxiliando bastante neste processo.

Unidade 6 - Comandos de repetição

6.1 Primeiras palavras

Todo problema que pode ter uma solução algorítmica precisa de entrada, saída, atribuição, condicionais e **repetições**. Só isso. Claro que há mais algumas coisas para ajudar no processo, mas esta Unidade, com comandos que permitem repetir um conjunto de ações, encerra o conjunto de estruturas de controle de fluxo. Ou seja, estes são os últimos comandos.

Enquanto condicionais permitem que um conjunto de comandos seja opcionalmente executado, os comandos de repetição são construídos para habilitar a execução de um conjunto de comandos diversas vezes.

Assim como não há algoritmos (a não ser os muito simples) que não usem condicionais, praticamente não há algoritmos (a não ser os simples) que não usem comandos de repetição.

6.2 Por que fazer de novo?

Situações repetitivas são mais comuns no cotidiano do que se pode perceber. Se uma receita de bolo requer que quatro gemas sejam colocadas na massa, está subentendido que o processo "pegue um ovo; quebre sua casca; separe a clara; coloque a gema na tigela" tenha que ser repetido quatro vezes. Se o pneu do carro furou e é preciso colocar o estepe no lugar, o processo "coloque a chave no parafuso; remova o parafuso; coloque o parafuso em um lugar onde possa ser encontrado depois" deve ser repetido para cada parafuso (três, quatro ou cinco vezes, geralmente, dependendo do veículo). Na aula de desenho, os passos "pegue uma folha em branco; faça seu trabalho; avalie o que foi feito; se ficou ruim, jogue fora" devem ser repetidos até que se julgue o trabalho bom e este possa ser entregue ao professor.

Além disso, há tipos de repetições diferentes. Uma indica "faça isso tantas vezes" e outra instrui "faça isso até que tal situação seja alcançada". Cada uma delas tem lógica diferente e se aplica a uma situação diferente.

Esta Unidade descreve quatro modos distintos de repetir.

6.3 Fluxo de execução com repetição

Um comando de repetição corresponde a uma única instrução, mesmo que tenha várias instruções internas a ele. Estas instruções "internas" são consideradas "subordinadas" à repetição.

O primeiro formato de comando de repetição é bastante específico: é usado quando se sabe quantas vezes o conjunto de comandos vai ser repetido. É algo como "faça isso 10 vezes". Este comando é denominado **para** e tem seu formato geral indicado abaixo.

```
para identificador ← expressão_início até expressão_fim faça
     lista_de_comandos_subordinados
fim-para
```

O **identificador** tem que se referir necessariamente a uma variável inteira, assim como também devem ser considerados inteiros os resultados da **expressão do valor de início** e da **expressão de valor de fim**. Se a expressão não for inteira, o valor real é truncado (i.e., tem sua parte fracionária desprezada).

O comando se comporta primeiro fazendo a atribuição do valor inicial à variável, que dentro do **para** é chamada **variável de controle**. (Qualquer variável inteira pode ser usada, mas somente é uma variável de controle se estiver sendo usada no lugar do identificador do comando **para**.) A seguir, o conjunto de comandos especificado é executado uma vez, sendo que o valor da variável de controle é válido e pode ser usado. Terminada a execução, a variável de controle é incrementada de 1 e os comandos executados novamente, sendo este processo repetido até que, na última vez, a variável tenha o valor especificado na expressão final.

Como exemplo, o Algoritmo 6-1 mostra como pode ser produzida uma lista de conversões de graus Celsius para Fahrenheit.

Algoritmo 6-1

```
1  { algoritmo para gerar uma tabela de conversão Celsius -> Fahrenheit
2    para valores de 0 a 40, de 1 em 1 }
3
4  algoritmo
5        declare celsius: inteiro { sempre inteiro, neste caso }
6                Fahrenheit: real { real, pois é obtido pela conversão }
7
8        { repetição para os valores indicados }
9        para celsius ← 0 até 40 faça
10            fahrenheit ← celsius * 1,8 + 32
11            escreva(celsius, "  <--->  ", fahrenheit)
12       fim-para
13 fim-algoritmo
```

Este algoritmo exemplo possui um único comando: o **para** que se inicia na linha 9 e termina na linha 12, com o delimitador **fim-para**. Subordinados a esta repetição estão os comandos das linhas 10 e 11. O comando **para** se inicia atribuindo 0 à variável **celsius** e então executando os dois comandos internos (calculando o valor convertido e escrevendo ambos). Escrito o primeiro par da "tabela de conversão", o valor de **celsius** é incrementado de um e os comandos internos são repetidos para o valor 1. Depois há novo incremento e repetição para o valor 2, e depois para 3, para 4, e assim sucessivamente, até que a última repetição se dá para o valor 40, que é o valor final. Após o **para** realizar todas as repetições, o algoritmo termina, pois não há mais comandos. Porém, antes de terminar, deve-se observar que o valor de **celsius** após o **para** é 41, ou seja, o último incremento é sempre feito.

Para especificar melhor o comportamento da repetição com o **para**, seguem algumas características importantes:

1) A repetição do **para** ocorre sempre de forma a incrementar a variável de controle, sendo, portanto, sempre crescente;
2) Caso o valor inicial seja maior que o valor final, os comandos internos não serão executados nenhuma vez e o valor da variável de controle será simplesmente igual ao valor de início (não haverá incremento, pois não houve execução);
3) Se o valor de início for igual ao valor de fim, os comandos serão executados uma única vez e a variável de controle terminará com valor incrementado de 1;
4) Alterar o valor da variável de controle dentro do **para** é considerado **erro** de uso do comando;
5) Caso os valores de início e de fim sejam formado por expressões que usam variáveis, estas expressões serão avaliadas somente uma vez antes das repetições começarem; portanto, se houver modificação dos valores das variáveis, o número de repetições estimado inicialmente não será alterado.

```
para valor ← 1 até 5 faça
        escreva(valor)
fim-para
```

valor	saída
1	1
2	2
3	3
4	4
5	5
6	

```
inicial ← 100
para valor ← 1 até 3 faça
        escreva(inicial)
        inicial ← inicial - 10
fim-para
escreva(valor, inicial)
```

inicial	valor	saída
100	1	100
90	2	90
80	3	80
70	4	

```
valorBase ← 15
para valor ← valorBase/3 até valorBase/2 { 5 a 7 }
  valorBase ← valorBase - 1
fim-para
escreva(valorBase)
```

valorBase	valor	saída
15	5	12
14	6	
13	7	
12	8	

Figura 6-1. Exemplos de trechos de algoritmo com o comando para **e seus respectivos testes de mesa.**

Pode-se notar que em todas as instâncias do comando **para** a variável de controle (**valor**, em todos os exemplos) acaba com valor igual

ao valor da expressão que indica o valor final acrescido de 1. Outra observação pertinente é que, dados os valores inicial e final, o número de repetições é claramente determinado (sempre igual a *final – inicial + 1*). No último exemplo, ainda, o conteúdo da variável **valorBase** é alterado durante a execução da repetição, mas isso não modifica o número de vezes que a repetição é efetuada (lembrando que os valores de início e fim são avaliados uma única vez, antes do início do ciclo de repetição).

O comando **para** é um comando bastante poderoso e usado amplamente como ferramenta para realizar repetições e resolver problemas. Lembrando-se de suas restrições de uso, pode ser aplicado sempre que se puder determinar, de antemão, quantas vezes o **laço de repetição** será realizado e os valores inicial e final que terá.

Porém, o comando **para** não pode ser usado em uma série de situações, exatamente aquela em que não se sabe quantas vezes a repetição será feita. Um exemplo simples poderia ser dado para o caso de se registrarem movimentações (créditos e débitos) em um saldo e parar quando o saldo ficar negativo. Não é possível saber quantas movimentações serão processadas, pois isso depende do saldo e dos valores de cada movimentação; o saldo pode já estar negativo e nenhuma movimentação ser feita, apenas uma movimentação já pode deixar o saldo negativo, ou então algumas centenas de movimentações podem ocorrer, até que a quantidade disponível da conta se esgote.

Para estas situações, há outros comandos de repetição. Estes se apresentam com filosofias de operação levemente diferentes, mas sua estrutura básica é similar.

O primeiro deles é chamado de **enquanto** e tem sua estrutura mostrada abaixo.

```
enquanto expressão_lógica faça
     conjunto_de_comandos
fim-enquanto
```

Sua operação é bastante simples: primeiro a condição especificada é avaliada e, sendo o resultado **verdadeiro**, o conjunto de comando é executado. Ao final de cada repetição, a condição é novamente avaliada para verificar se o conjunto de comandos será executado novamente ou se o comando será encerrado.

Algoritmo 6-2

```
1  { algoritmo que apresenta uma saudação para alguns nomes }
2
3  algoritmo
4        declare nome: literal
5
6        { primeiro nome a ser lido }
7        leia(nome)
8
9        { repetição que termina quando o nome for "fim" }
10       enquanto nome ≠ "fim faça
```

```
11            { escreve a saudação }
12            escreva("Olá,", nome, ", como vai?")
13
14            { leitura do próximo nome }
15            leia(nome)
16      fim-enquanto
17
18      { despedida }
19      escreva("Não há mais ninguém? Então, tchau!")
20 fim-algoritmo
```

O Algoritmo 6-2 mostra um algoritmo simples que fica repetidamente lendo nomes (todos em uma mesma variável chamada **nome**, que é reaproveitada, "esquecendo" os nomes anteriores). Uma saudação simples para cada nome é escrita na tela, a não ser que o nome digitado seja "fim", o que causa o encerramento da repetição e o término do algoritmo com a mensagem final de despedida (linha 19).

A Figura 6-2(a) mostra a execução do algoritmo se os nomes escritos forem Otávio, Pedro e Adalberto, seguidos de "fim" para terminar. É apresentado um teste de mesa comentado, indicando a operação de processamento do algoritmo.

Outro exemplo é também apresentado na Figura 6-2(b), que mostra o comportamento do algoritmo quando "fim" é a primeira (e única) entrada para o algoritmo.

nome	**saída**	*Comentário*
Otávio	*Olá, Otávio, como vai?*	O nome é lido na linha 7 e comparado, na linha 10, com a constante "fim"; como são diferentes, os comandos do enquanto são executados: a mensagem é escrita e um nome é lido novamente (linha 15).
Pedro	*Olá, Pedro, como vai?*	"Pedro" foi transferido na linha 15 para a variável **nome** e a primeira repetição se encerra. O teste da linha 10 é feito novamente e novamente resulta em **verdadeiro**. Nova mensagem e nova leitura são feitas (segunda repetição).
Adalberto	*Olá, Adalberto, como vai?*	Como "Adalberto" é diferente de "fim", a terceira repetição é feita, com mensagem para Adalberto e leitura do valor "fim".
fim	*Não há mais ninguém? Então tchau!*	Ao fim da terceira repetição, a comparação da linha 10 finalmente resulta em **falso** ("fim" ≠ "fim") e o enquanto se encerra, passando então para o **escreva** da linha 19, que é o último comando do algoritmo.

(a)

nome	**saída**	*Comentário*
fim	*Não há mais ninguém? Então tchau!*	A leitura da linha 7 já coloca o valor "fim" na variável **nome**. Na linha 10, é feita a comparação do **enquanto**, que já resulta em **falso**. Com isso, os comandos internos não são executados e a repetição é encerrada. O próximo comando é executado (linha 19) e o algoritmo termina.

(b)

Figura 6-2. Teste de mesa para o Algoritmo 6-2. (O texto traz mais detalhes.)

Os exemplos de execução ilustram a situação "normal", na qual vários nomes são digitados e, depois, há o encerramento indicado pela entrada "fim", e uma situação "limite", na qual "fim" é indicado logo na primeira vez. O algoritmo foi escrito para funcionar de forma harmoniosa para ambas as situações, evitando que uma mensagem como "Olá, fim, tudo bem?" seja escrita.

O comando **enquanto** é, portanto, um comando de repetição que é usado quando não se sabe quantas vezes a repetição será feita, usando para isso um **teste de término superior** (isto é, no início do comando). Esta característica faz com que o comando possa não ter seus comandos internos executados nenhuma vez, como ilustrado na Figura 6-2(b). Outro ponto relevante é que, ao contrário do **para**, não existe uma variável de controle. Isto quer dizer que não há restrições ou incrementos automáticos envolvidos.

Nem sempre, entretanto, se deseja que a verificação seja feita antes do conjunto de comandos, pois pode ser necessária a primeira repetição e, depois, verificar se deve haver ou não outra execução. Para esta situação existe o terceiro tipo de comando, que se apresenta com duas alternativas, ambas mostradas na seqüência.

```
faça
     conjunto_de_comandos
enquanto expressão_lógica
```

```
faça
     conjunto_de_comandos
até condição_lógica
```

Em cada uma delas há, primeiro, a execução do conjunto de comandos. Após cada vez que houver a execução, o teste será feito para verificar se haverá nova repetição ou o encerramento do comando. Como o comando se apresenta em dois modos diferentes, a primeira versão, **faça/enquanto**, tem sua repetição continuada somente quando a expressão lógica for avaliada como verdadeira. Na versão **faça/até**, a repetição é continuada somente quando a expressão lógica for avaliada como falsa.

Como exemplos, o problema da tabela de conversão de graus Celsius para Fahrenheit é revisitado, com uma pequena alteração. "Escreva um algoritmo que gere uma tabela de conversão de graus Celsius para

Fahrenheit, iniciando em 0 e terminando em 10, com intervalo de 0,5.” O Algoritmo 6-3 e o Algoritmo 6-4 mostram uma possível solução, a primeira usando o **faça / enquanto** e outra o **faça / até**.

Algoritmo 6-3

```
1  { geração de uma tabela de conversão de graus Celsius para Fahrenheit
2    de 0 a 10 (inclusive), de 0,5 em 0,5. }
3
4  algoritmo
5        declare celsius, fahrenheit: real
6
7        { geração da tabela }
8        celsius ← 0    { valor inicial }
9        faça
10            { calcula o valor convertido e escreve ambos }
11            fahrenheit ← celsius * 1,8 + 32
12            escreva(celsius, " <---> ", fahrenheit)
13
14            { passa para o próximo valor em Celsius }
15            celsius ← celsius + 0,5
16       enquanto celsius ≤ 10
17 fim-algoritmo
```

Algoritmo 6-4

```
1  { geração de uma tabela de conversão de graus Celsius para Fahrenheit
2    de 0 a 10 (inclusive), de 0,5 em 0,5. }
3
4  algoritmo
5        declare celsius, fahrenheit: real
6
7        { geração da tabela }
8        celsius ← 0    { valor inicial }
9        faça
10            { calcula o valor convertido e escreve ambos }
11            fahrenheit ← celsius * 1,8 + 32
12            escreva(celsius, " <---> ", fahrenheit)
13
14            { passa para o próximo valor em Celsius }
15            celsius ← celsius + 0,5
16       até celsius > 10
17 fim-algoritmo
```

Em ambos os algoritmos, a condição de término gira em torno do valor da variável **celsius**, o qual é alterado a cada ciclo da repetição, sendo feito um acréscimo de 0,5 a cada **laço da repetição**. No Algoritmo 6-3 é feita a verificação usando o “menor ou igual”, pois deve haver a próxima execução durante todo o tempo em que a variável **celsius** permanecer abaixo de 10 ou for igual a este valor. A condição do **até** é a oposta do anterior, e no Algoritmo 6-4, é feita a verificação por “maior que 10” (ou

seja, "não menor ou igual"); neste caso, quando a condição for verdadeira, o ciclo de repetição é encerrado, o que ocorrerá quando **celsius** tiver valor 10,5.

O acompanhamento de ambos os algoritmos com o teste de mesa é muito importante para compreender como cada instrução é executada e como o controle de término da repetição é considerado.

Por ser importante, a diferença dos comandos **enquanto** e do comando **faça** (em qualquer versão) é novamente salientada: o **enquanto**, por fazer o teste no início, pode executar seus comandos internos **zero ou mais vezes**; os comandos **faça**, por sua vez, podem executar os comandos subordinados **uma ou mais vezes**, já que o teste é feito no final do comando.

6.3.1 Condições de execução de repetição: estado anterior, critério de término, estado posterior

Todos os comandos de repetição são, de certa forma, traiçoeiros. É muito freqüente escrever as repetições e, devido à falta de cuidado, haver uma repetição a mais ou uma a menos. Em outras palavras, uma repetição que deveria ocorrer 100 vezes pode ser feita 99 ou 101 vezes, o que certamente ocasionará um erro no resultado esperado para o algoritmo. Em especial este erro é menos comum no uso do **para**, quando comparado aos outros comandos de repetição.

Por este tipo de equívoco ser freqüente e até provável de acontecer, são recomendados alguns cuidados importantes quando repetições forem usadas.

No caso do **para**, a recomendação é verificar com atenção as expressões que são usadas para indicar os valores inicial e final da repetição. O ponto chave é lembrar que ambas as expressões são inteiras e que, se um valor real for usado, deve-se dar atenção especial se se deseja truncar ou arredondar o resultado. Por exemplo, se a repetição iniciar em 1 e for até o resultado da expressão *188,0/10*, a repetição é de 1 a 18; se for de 1 até *arred(188,0/10)*, o valor final será 19. Essa diferença pode produzir um resultado incorreto, dependendo da situação.

Assim, para cada repetição devem ser verificados seu **estado anterior**, ou seja, o valor da variáveis que participam das condições de término da repetição antes dela se iniciar, o **critério de término**, que é até quando a repetição continua, e o **estado posterior**, que são os valores após o encerramento da repetição.

O comando **para** tem essa verificação mais simples: a única variável que participa da condição de execução é a variável de controle e seus valores inicial e final são fornecidos através das expressões. O cuidado deve ser tomado segundo as recomendações de conversões entre reais e inteiros indicadas acima. Portanto, o estado anterior é a atribuição automática do primeiro valor à variável; o critério de término é haver repetição a ser feita, o que depende do valor final; e o estado posterior é o valor final acrescido de 1, caso tenha havido pelo menos uma execução dos comandos internos, ou é o próprio valor inicial, se nenhuma repetição tiver ocorrido.

Para os demais comandos, é preciso ter mais cuidado, visto que não há variável de controle. Para estes, é necessário verificar a expressão lógica usada no comando. Todas as variáveis usadas nesta expressão, além de algumas outras que contribuem para que seus conteúdos sejam alterados, são importantes. O critério de término depende não somente da expressão, mas também de como as variáveis são alteradas nos comandos internos. Por fim, o estado posterior deve ser avaliado para os valores envolvidos na expressão lógica e para os valores correlatos (i.e., que contribuem para seus valores).

Observar cada repetição com cuidado evita problemas e, em conseqüência, incorreções nos algoritmos. A prática ajuda a avaliar as situações que se apresentam e, portanto, o treino e a realização de exercícios com atenção são fundamentais para dominar estes conceitos.

6.3.2 Aplicações práticas de repetições

O princípio de funcionamento das repetições não é extremamente complexo. Porém, conseguir extrair proveito de suas características para resolver problemas pode ser um desafio. Nesta seção serão cobertos alguns dos usos comuns de repetições, através de exemplos.

A primeira aplicação é, naturalmente, a de repetição de tarefas que ocorrem várias vezes, como já foi exemplificado nos algoritmos anteriores. Outro exemplo: "A média final de uma disciplina é calculada pela média aritmética simples de três provas. Escreva um algoritmo que, dado o número de alunos de uma turma e suas notas de provas, calcule para cada um deles sua média final.". Um algoritmo para este problema é o Algoritmo 6-5, no qual a variável **i** é usada somente para contar os alunos.

Algoritmo 6-5

```
1  { calcular a média de três provas para cada aluno de uma turma, dado o
2    número de alunos e as notas das provas }
3
4  algoritmo
5        declare
6             i, númeroAlunos: inteiro
7             prova1, prova2, prova3, média: real
8
9        { obtenção do número de alunos }
10       leia(númeroAlunos)
11
12       { cálculo da média para cada aluno }
13       para i ← 1 até númeroAlunos faça
14            { obtenção das notas }
15            leia(prova1, prova2, prova3)
16
17            { cálculo e escrita dos resultados }
18            média ← (prova1 + prova2 + prova3)/3
19            escreva(média)
20       fim-para
21 fim-algoritmo
```

Outras aplicações envolvem tarefas comuns em algoritmos, como por exemplo, determinar o maior (ou o menor) valor de um conjunto de dados, realizar somas de vários valores, calcular a média, contar o número de ocorrências de uma determinada situação, ou então combinações entre elas.

Os primeiros exemplos são de determinação do **maior valor** em um conjunto de dados. Naturalmente a determinação do menor valor é bastante similar e será apenas comentada. Para tanto, o problema é o seguinte: "Uma adega possui uma relação de seus vinhos, contendo nome do produto, seu preço e seu tipo (tinto, branco ou rosê). O número total de garrafas não é conhecido inicialmente, mas convenciona-se que se o nome do vinho for "fim" significa que a relação se encerrou. Escreva um algoritmo para ler a lista de vinhos e indicar o produto mais caro, dando seu nome, seu preço e seu tipo. Assuma que não existam dois vinhos com o mesmo preço[*].".

O Algoritmo 6-6 resolve o problema usando uma estratégia simples. Uma variável (chamada **preçoMaior**) é iniciada com um valor abaixo do menor preço possível (que pode até ser R$0,00, se o vinho for de graça). Para cada vinho, seu preço é comparado com o valor armazenado em **preçoMaior** e, se for maior que este, o novo vinho mais caro é registrado. Além do preço, também são guardados seu nome e tipo, para que sejam escritos posteriormente.

Algoritmo 6-6

```
1  { dados o nome, o preço e o tipo dos vinhos (indicados aqui por "T" para
2    tinto, "B" para branco ou "R" para rosê, descrever o vinho mais caro;
3    não são considerados vinhos de preços iguais;
4    fim dos dados indicado por nome = "fim" }
5
6  algoritmo
7       declare
8            nome, nomeMaior,
9            tipo, tipoMaior: literal
10           preço, preçoMaior: real
11
12      { repetição para leitura dos dados }
13      preçoMaior ← -1  { para forçar a troca na primeira verificação }
14      faça
15           { dados }
16           leia(nome, preço, tipo)
17
18           { verificação do maior preço }
19           se preço > preçoMaior e nome ≠ "fim" então
20                nomeMaior ← nome
21                preçoMaior ← preço
22                tipoMaior ← tipo
```

[*] Esta restrição é para evitar o problema de dois ou mais vinhos "empatados" como mais caros.

```
23            fim-se
24       até nome = "fim"
25
26       { apresentação do resultado }
27       se preçoMaior = -1 então
28            escreva("Nenhum vinho foi apresentado.")
29       senão
30            escreva(nomeMaior, preçoMaior, tipoMaior)
31       fim-se
32 fim-algoritmo
```

Há ainda duas observações pertinentes quanto à solução. A primeira é que, caso na primeira informação seja digitado "fim" para o nome (e quaisquer valores para preço e tipo), o valor de **preçoMaior** chega ao final inalterado, o que permite saber que não houve entradas e, portanto, não há resultados. A outra é que, na linha 19, a comparação se o nome do vinho não é "fim" é necessária para que os dados de um vinho chamado "fim", que não existe, não sejam considerados como válidos ao se achar o mais caro. Fica como exercício pensar como alterar o algoritmo para que o vinho mais barato seja encontrado. Neste caso, qual seria o valor inicial para uma eventual variável **preçoMenor**? Para se pensar no assunto, pode-se assumir que não há nenhum vinho com valor acima de R$200,00.

Em muitas situações, quando não há limites para o intervalo de variação da variável cujo maior valor (ou o menor) se deseja achar, a tática de colocar um valor inicial pequeno não funciona. Por exemplo, suponha que se precise achar a menor temperatura de um conjunto. Caso se opte por colocar a maior temperatura com valor inicial -1 e todas as temperaturas que forem entradas estiverem abaixo deste valor, nunca haverá troca e, portanto, o algoritmo falha. Se se optar por usar -100 no lugar do -1, pode ser que todas as temperaturas sejam abaixo de -100 e o algoritmo falha de novo. Esta técnica funciona apenas quando se sabe que há um limite para os valores. O Algoritmo 6-7 apresenta uma solução para o seguinte problema: "Dado um conjunto de temperaturas e conhecendo-se a quantidade de itens de entrada, determinar a menor temperatura, escrevendo um algoritmo que indique este resultado.".

Algoritmo 6-7

```
1  { determinar a menor temperatura de um conjunto, sabendo-se a quantidade
2    de valores disponíveis }
3
4  algoritmo
5       declare
6            i, númeroEntradas: inteiro
7            temperatura, menorTemperatura: real
8
9       { obtenção da quantidade de itens }
10      leia(númeroEntradas)
11
12      se númeroEntradas ≤ 0 então
13           escreva("Não há dados; portanto não há menor temperatura")
```

```
14       senão
15            { leitura da primeira temperatura, em separado }
16            leia(temperatura)
17            menorTemperatura ← temperatura { a primeira é a menor }
18
19            { repetição para leitura das outras temperaturas}
20            para i ← 2 até númeroEntradas faça
21                 { leitura}
22                 leia(temperatura)
23
24                 { verificação do maior preço }
25                 se temperatura < menorTemperatura então
26                      menorTemperatura ← temperatura
27                 fim-se
28            fim-para
29
30            { resultado }
31            escreva(menorTemperatura)
32       fim-se
33 fim-algoritmo
```

Nesta última solução, a primeira temperatura é lida em separado e, como é a única até o momento, ela é considerada a menor. O comando **para** da linha 20 faz a leitura para os itens restantes (notando-se que começa em 2, e não em 1), comparando-os com o primeiro ou com o menor até o momento. Neste caso, independentemente das temperaturas que forem informadas, o resultado correto será fornecido, não sendo necessário conhecer a faixa de variação das temperaturas.

Outra aplicação prática bastante usual para as repetições é contar. Isso pode ser ilustrado pelo Algoritmo 6-8, que resolve o seguinte problema: "Conhecendo-se a quantidade de pessoas de um grupo e suas idades, escrever um algoritmo que indique quantas são maiores de idade (referência a 18 anos) e qual a porcentagem destes que têm idade acima de 60 anos.".

Algoritmo 6-8

```
1  { determinar, dadas uma quantidade conhecida de pessoas e suas idades,
2    quantas são maiores de 18 (maior ou igual) e, dentre estas, a
3    porcentagem de pessoas acima de 60 anos }
4
5  algoritmo
6       declare
7            i, númeroPessoas,
8            conta18, conta60: inteiro
9
10      { obtenção da quantidade de pessoas }
11      leia(númeroPessoas)
12
13      { contagem de maiores de idade e de acima de 60 anos }
```

```
14       conta18 ← 0
15       conta60 ← 0
16       para i ← 1 até númeroPessoas faça
17            { leitura da idade }
18            leia(idade)
19
20            { analisa a idade }
21            se idade ≥ 18 então
22                 conta18 ← conta18 + 1 { conta mais 1 }
23                 se idade > 60 então
24                      conta60 ← conta60 + 1
25                 fim-se
26            fim-se
27       fim-para
28
29       { resultados }
30       se conta18 = 0 então
31            escreva("Não há maiores de idade nem sexagenários")
32       senão
33            escreva("Há", conta18, "pessoas maiores de idade")
34            escreva("Destas,", 100,0 * conta60/conta18,
35                                        "% têm mais de 60 anos")
36       fim-se
37  fim-algoritmo
```

O uso de contadores, normalmente dentro de condicionais, mas não necessariamente, é feito começando com zero e, quando preciso, incrementando o valor de 1. Também é muito freqüente ser necessário, no final, verificar se houve contagem, pois sempre há o risco de divisão por zero (como na linha 34, se não houvesse sido feita a verificação).

De forma similar aos contadores, é também possível fazer somas e, em decorrência delas, calcular médias. Problema: "Uma loja de informática possui uma lista de preços de DVDs virgens de várias marcas, juntamente com as quantidades em estoque de cada um. Deseja-se saber, a partir da relação de preços e quantidades, o valor total dos DVDs e o preço médio da unidade. Escreva um algoritmo para processar os dados e escrever estes resultados. Como o número de itens não está disponível, deve-se considerar que a lista termina quando for digitado o valor zero tanto para preço quanto para quantidade.". Uma solução é apresentada no Algoritmo 6-9.

Algoritmo 6-9

```
1  { dada uma relação de preços e quantidade relativas a mídias de DVD
2    virgens, calcular o valor total em estoque (em reais) e o preço médio
3    da unidade de DVD; o fim dos dados é indicado por preço e quantidade
4    iguais a zero }
5
6  algoritmo
7       declare
8
```

```
9
10       { valores iniciais dos contadores }
11       númeroItens ← 0
12       valorTotal ← 0
13       somaUnitários ← 0
14
15       { leitura dos dados }
16       leia(preço, quantidade)
17       enquanto preço ≠ 0 e quantidade ≠ 0 faça
18            { conta e acumula }
19            númeroItens ← númeroItens + 1
20            valorTotal ← valorTotal + preço * quantidade
21            somaUnitários ← somaUnitários + preço
22
23            { obtém o próximo item }
24            leia(preço, quantidade)
25       fim-enquanto
26
27       { resultados }
28       escreva("Valor total: R$", valorTotal)
29       se númeroItens = 0 então
30            escreva("Não é possível calcular o preço médio unitário")
31       senão
32            escreva("Valor unitário médio: R$",
33                                    1,0 * somaUnitários/númeroItens)
34       fim-se
35  fim-algoritmo
```

Somas também usam, geralmente, a tática de iniciar o acumulador com zero e, sucessivamente, ir somando a ele os valores que forem necessários.

Um problema alternativo que utiliza somas é a classe que calcula valores por aproximação. Um exemplo é o fato de o valor do cosseno de um arco poder ser calculado por uma soma. É claro que a soma precisaria ser infinita para que os resultados fossem iguais, mas o caso mais comum é se contentar com uma aproximação. Assim, pode-se escrever $\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots$ e, deste modo, elaborar um algoritmo para, dado um valor de arco, aproximar seu cosseno. Uma proposta é o Algoritmo 6-10.

Algoritmo 6-10

```
1  { dado o comprimento de um arco, calcular seu cosseno pela soma
2    cos(x) = Σ x^i/i! para um dado número de termos }
3
4  algoritmo
5       declare i, termo, baseFatorial, fatorial: inteiro
6               cosseno, ângulo: real
7
```

```
8       { leitura do arco e do número de termos }
9       leia(ângulo, númeroTermos)  { ângulo em radianos }
10
11      { cálculo da aproximação do cosseno }
12      cosseno ← 0   { acumulador do resultado }
13      baseFatorial ← 1
14      fatorial ← 1
15      termo ← 1
16      para i ← 1 até númeroTermos faça
17           { faz o somatório }
18           se i % 2 = 1 então
19                cosseno ← cosseno + termo   { soma termos ímpares }
20           senão
21                cosseno ← cosseno – termo   { subtrai termos pares }
22           fim-se
23
24           { calcula o próximo termo }
25           fatorial ← fatorial * baseFatorial * (baseFatorial + 1)
26           baseFatorial ← baseFatorial + 2
27           termo ← pot(x, i + 1)/fatorial
28      fim-para
29
30      { resultado calculado }
31      escreva("cos(", ângulo, ") = ", cosseno)
32 fim-algoritmo
```

Neste algoritmo alguns pontos podem ser destacados. Inicialmente, há o cuidado para que o somatório considere os termos positivos e negativos, o que é feito somando os termos de posições pares e subtraindo o das ímpares. No caso, o resto de uma divisão inteira por 2 é sempre zero para pares e 1 para ímpares. Outro detalhe é o controle do fatorial, que se inicia com 1, pois haverá sucessivas multiplicações (e não ajuda multiplicar por zero), e o controle também da base do fatorial. Como a cada passo são multiplicados dois fatores (2!, 4!, 6!...), as duas multiplicações são feitas de uma vez (linha 25).

O último exemplo: "Um número inteiro maior que zero é chamado de número perfeito quando a soma de todos seus divisores (exceto o próprio número) tiver valor igual a ele. Por exemplo, 28 é perfeito, pois 1+2+4+7+14 = 28. Escreva um algoritmo que leia um número inteiro qualquer e determine se ele é perfeito.". O Algoritmo 6-11 ilustra uma solução.

Algoritmo 6-11

```
1   { dado um número inteiro, determinar se é um número perfeito, ou seja,
2     se é igual à soma de seus divisores, exceto ele mesmo }
3
4  algoritmo
5       declare
6
```

```
7        { leitura do número }
8        leia(valor)
9
10       { soma dos divisores de valor }
11       soma ← 0
12       para i ← 1 até valor/2
13            se valor % i = 0 então    { soma somente se for divisor }
14                 soma ← soma + i
15            fim-se
16       fim-para
17
18       { resultado }
19       se soma = valor e valor ≠ 0 então
20            escreva(valor, "é um número perfeito")
21       senão
22            escreva(valor, "não é um número perfeito")
23       fim-se
24 fim-algoritmo
```

O "truque" deste algoritmo é somente fazer a soma dos termos que interessam. No caso, somar os divisores (ou seja, aqueles que dividem **valor** e deixam resto zero). Um outro detalhe é que o comando **para** vai somente até a "metade do caminho"; isso é porque não existem divisores que sejam maiores que a metade de um dado número. Ficam como exercício para o leitor determinar a razão da comparação com zero na linha 19 e porque os números negativos são tratados de forma adequada pelo algoritmo.

6.4 Considerações finais

As repetições são a alma do processamento de dados. Por meio delas são indicadas tarefas executadas diversas vezes ou para um grande número de valores.

As diferenças entre os três tipos de comandos de repetição é um ponto crucial: o **para** é usado quando se sabe o primeiro e o último valor da repetição, que usa somente inteiros e vai sempre de 1 em 1; o **enquanto** é um comando mais livre, sem variável de controle, que testa condições simples ou complexas, sendo a verificação feita **antes** da execução dos comandos internos; e o par **faça / enquanto** e **faça / até** definem repetições livres como as do **enquanto**, mas o teste é feito sempre **depois** da repetição. Entender estas diferenças ajuda a usar o comando mais adequado a cada situação.

Finalmente, é preciso reforçar que repetições são utilíssimas. Dentre os exemplos vistos, podem ajudar a localizar o maior ou o menor do conjunto, somar valores, contá-los e calcular médias. Outras aplicações de repetições incluem busca por valores, determinação da ocorrência ou não de uma condição para um conjunto de dados ou fazer cálculos específicos, como determinar o máximo divisor comum entre dois valores inteiros.

Unidade 7 - Ponteiros

7.1 Primeiras palavras

Esta é uma Unidade curta. Seu objetivo é dar uma noção geral e precisa do que são **ponteiros**. Os ponteiros são um tipo de variável de ampla utilização em algoritmos, mas são considerados como estruturas avançadas.

Ponteiros são úteis para disciplinas como **Estruturas de dados** e **Organização e recuperação da informação**. Com eles é possível usar partes da memória sem precisar declarar uma variável para cada parte usada e estruturar dados de forma mais complexa (naturalmente mais difícil, mas com vantagens compensadoras).

Esta Unidade, porém, visa apenas fornecer condições para que os alunos entendam **o que** é um ponteiro e **como** podem ser usados para manipular a memória. É o básico do básico; por isso o assunto é breve.

7.2 Uma questão de organização

Um aluno aplicado cumpre suas tarefas com afinco. Faz exercícios, estuda, faz anotações, escreve textos e ensaios. Para isso, utiliza um caderno, por exemplo. Este caderno é a sua "memória" de estudos. Se o volume de coisas feitas no caderno for muito grande e for necessário localizar as coisas, vários recursos podem ser usados para ajudar na localização de coisas importantes. Uma alternativa seria comprar uma coleção de etiquetas que colam e descolam, multicoloridas, e encher o caderno de "orelhas" com pequenas anotações. Outra, seria numerar as páginas e manter, normalmente a um grande custo, um índice remissivo para o conteúdo. Em ambas, contar com a memória natural é sempre útil, também.

Organizar os dados é importante sempre, desde o uso da organização mais simples até a mais elaborada. O "esquema" de organização é foco para que sejam localizadas as informações e para relacioná-las umas às outras.

7.3 Ponteiros e endereços na memória

No modelo de computador, o algoritmo e os dados são armazenados na memória. A unidade de processamento executa o algoritmo, o qual dá instruções para que os dados sejam transformados de forma a produzir a resposta pretendida. Esta foi a Unidade 2.

Uma questão importante que não foi detalhada é a relação entre uma variável e a memória. Já se sabe que uma variável usa uma parte da memória para guardar um valor; sabe-se também que seu tipo determina como este valor é armazenado e interpretado. Outra coisa conhecida é que a posição de memória é localizada corretamente, pois é dado um nome para ela, que é o identificador.

7.3.1 Memória e endereçamento

Para especificar melhor a relação entre a memória e as variáveis, é preciso colocar que, para organizar a memória, existe um esquema de **endereçamento**. Desta maneira, toda parte da memória tem um endereço, que é sua posição. Portanto, toda variável tem seu endereço e quando um algoritmo usa o identificador da variável, a unidade de processamento "sabe" que aquele identificador está associado a um endereço e, assim, consegue localizar o dado na memória.

Embora existam diversos esquemas de endereçamento, usualmente a memória pode ser pensada como uma coleção finita de unidades (chamadas de *bytes* em um computador real) e cada uma delas tem um endereço. Este endereço usualmente começa em zero e vai até o número total de unidades menos 1. Em uma memória real de 1Gbyte, existem 1.073.741.824 de unidades, que podem ser identificadas pelos números de 0 até 1.073.741.823, por exemplo. Nem sempre o endereçamento é feito desta forma, mas é importante que se saiba que sempre há um endereçamento.

7.3.2 Acesso à memória com ponteiros

Voltando à Unidade 2, relembra-se que tanto os dados quanto os algoritmos estão na memória. Portanto, tudo o que o computador armazena, que é na memória, tem endereço.

Nos algoritmos existe um tipo abstrato de dados que foi propositalmente ignorado até agora: o tipo **ponteiro**. Uma variável do tipo **inteiro** guarda valores inteiros, variáveis reais guardam reais, as literais guardam textos e as do tipo **lógico** guardam **verdadeiro** ou **falso**. As variáveis do tipo **ponteiro** guardam endereços.

O Algoritmo 7-1 mostra um exemplo simples de variáveis do tipo **ponteiro**.

Algoritmo 7-1

```
1  { dadas duas variáveis, uma inteira e outra literal, escrever os
2    endereços de memória na qual estão armazenadas }
3
4  algoritmo
5        declare
6             valorReal: real
7             valorLiteral: literal
8             endereçoReal: ↑real        { endereços de variáveis reais }
9             endereçoLiteral: ↑literal  { endereços de variáveis literais }
10
11       { armazenamento dos endereços }
12       endereçoReal ← &valorReal
13       endereçoLiteral ← &valorLiteral
14
15       { escrita do resultado }
16       escreva("A variável real está no endereço:", endereçoReal)
17       escreva("A variável literal está no endereço:", endereçoLiteral)
18 fim-algoritmo
```

Existe um operador unário & que resulta em um endereço de seu operando. Este operador permite obter o endereço de praticamente qualquer coisa que exista em um algoritmo, mas é usado normalmente com variáveis. Deste modo, a expressão ***&valorReal*** resulta no endereço da memória na qual a variável está armazenada. A situação é a mesma para a expressão que retorna o endereço de **valorLiteral**. É bom saber, também, que se uma variável precisar de muitas unidades de memória (que é o caso das variáveis literais, evidentemente), o operador & retorna o endereço da primeira unidade, ou seja, do "começo" da área de memória.

As variáveis do tipo **ponteiro** são variáveis como quaisquer outras, possuindo um identificador, ocupando sua porção de memória e tendo, também, seu próprio endereço. A expressão ***&endereçoReal*** resulta na posição de memória na qual a variável **endereçoReal** está armazenada.

Uma terminologia comum quando se usam ponteiros é o termo **apontar**. Como uma variável ponteiro contém o endereço de outra variável, diz-se que ela "aponta para a variável". No Algoritmo 7-1, **endereçoLiteral**, após a atribuição da linha , aponta para **valorReal**.

Uma forma de se pensar nesta terminologia um tanto confusa é lembrando que os termos "apontar" e "endereço" têm significado similar neste contexto. Uma variável aponta para outra se contiver seu endereço. Isto também significa que a segunda é apontada pela primeira.

A grande pergunta, ainda, é: "para que servem os ponteiros?".

A resposta mais simples vem do fato de se poder mexer no dado guardado na memória se for conhecido apenas seu endereço. O Algoritmo 7-2 mostra como isso pode ser feito, introduzindo também a notação pertinente.

Algoritmo 7-2

```
1  { dada uma variável, modificar seu conteúdo usando apenas seu endereço }
2
3  algoritmo
4        declare valor: inteiro         { um inteiro comum }
5                ponteiro: ↑inteiro     { um ponteiro para inteiro }
6
7        { atribuição de um dado inicial }
8        valor ← 100
9
10       { determinação do endereço da variável }
11       ponteiro ← &valor
12
13       { uso do ponteiro para mexer na memória }
14       ↑ponteiro ← 200
15
16       { verificação do valor armazenado em valor }
17       escreva(valor)    { 200!!! }
18 fim-algoritmo
```

O Algoritmo 7-2 usa a variável **ponteiro**, que guarda o endereço de variáveis inteiras, para saber onde a variável **valor** está na memória. Isso é feito na linha 11.

A instrução na linha 14, por sua vez, pode ser lida da seguinte forma: "atribua à posição de memória apontada por **ponteiro** o valor 200". Mais simples seria dizer "o **conteúdo** de **ponteiro** recebe 200". Por "conteúdo de" entende-se o valor apontado pela variável. O efeito do comando é, portanto, armazenar o valor da expressão (200) na posição de memória, que coincide com a posição de **valor**. Na prática, houve a alteração do conteúdo armazenado na variável **valor** de forma indireta.

Pode-se observar que os ponteiros possuem um tipo relacionado, isto é, apontam para um tipo específico (ponteiro para **inteiro**, ponteiro para **real** etc.). Assim como uma variável tem seu tipo para saber o que é armazenado na área de memória reservada para ela, também o ponteiro deve conhecer este tipo para também armazenar dados de maneira coerente.

Uma das grandes utilidades de ponteiros é dar a possibilidade de modificar os dados armazenados na memória, sabendo-se apenas onde estão. Com isso é viável alterar indiretamente o valor das variáveis e, como caso prático, é possível escrever uma única seqüência de comandos que cada hora manipula uma variável diferente, pois em momentos diferentes pode apontar para localizações diferentes.

O Algoritmo 7-3 ilustra como um mesmo trecho de código pode, em momentos diferentes, modificar variáveis diferentes.

Algoritmo 7-3

```
1  { exemplo de modificação de variáveis diferentes usando o mesmo conjunto
2    de comandos }
3
4  algoritmo
5        declare
6             númeroPassos,
7             valor1, valor2: inteiro
8             Ponteiro: ↑inteiro
9
10       { atribuições iniciais }
11       valor1 ← 0
12       valor2 ← 0
13
14       { repetição }
15       leia(númeroPassos)
16       para i ← 1 até númeroPassos faça
17            { escolha da variável }
18            se i % 5 = 0 então
19                 ponteiro ← &valor1
20            senão
21                 ponteiro ← &valor2
22            fim-se
23
```

```
24            { comandos de manipulação da variável escolhida }
25            ↑ponteiro ← ↑ponteiro + i
26            se ↑ponteiro % 2 = 1 então
27                 ↑ponteiro ← ↑ponteiro – 1
28            fim-se
29       fim-para
30
31       { escritas }
32       escreva(valor1, valor2)
33 fim-algoritmo
```

Neste exemplo, o condicional da linha 15 determina que sempre que **i** for divisível por 5, **ponteiro** apontará para **valor1**; nos casos restantes, apontará para **valor2**. Os comandos das linhas de 22 a 25 usam somente a variável **ponteiro** e manipulam a memória que estiver sendo apontada naquele momento da execução, ou seja, algumas vezes manipula **valor1**, outras vezes **valor2**.

7.4 Considerações finais

Ponteiros são, para o iniciante no desenvolvimento de algoritmos (e de programas), um terreno obscuro e com nevoeiro. Iluminar o caminho requer atenção e exercícios práticos. Os testes de mesa são também aqui um aliado poderoso para entender o que está envolvido e quais as conseqüências produzidas pelos comandos.

Os principais pontos desta Unidade estão em saber que ponteiros armazenam endereços de outras variáveis e, por meio destes, permitem modificar os dados diretamente onde estão.

A utilidade dos ponteiros nos algoritmos, para este curso, é bastante limitada. Em programação, porém, poderá ser visto um uso essencial, principalmente na linguagem C.

Unidade 8 - Estruturas compostas heterogêneas: registros

8.1 Primeiras palavras

Umas das maiores ênfases que se dá aos algoritmos é a organização. Desde os comentários iniciais, com a documentação interna, passando pela organização visual adequada e pela escolha de nomes de variáveis que sejam significativos, em todos os lugares há organização.

Os dados, porém, continuam apenas amontoados de variáveis. É claro que pode haver organização na declaração, mantendo variáveis relacionadas mais próximas, seja por sua função no algoritmo, seja por tipo de dados. Isto, porém, é muito pouco significativo, embora tenha seu mérito.

Existem formas mais sofisticadas de organizar os dados e, como será visto nas próximas Unidades, são formas muito úteis.

Vários dados correlacionados podem ser mantidos juntos, criando-se uma única variável que junte todos eles. Esta nova variável pertence a um grupo chamado **variáveis compostas heterogêneas**, ou, simplesmente, **registros**.

8.2 Preenchendo fichas

Desejando obter um crediário em uma loja, um cliente gasta um pouco do seu tempo respondendo várias coisas ao vendedor, que as vai anotando em uma longa folha de papel. Entre as coisas anotadas estão o nome, endereço, bancos em que possui conta e número de documentos pessoais. A **ficha** será posteriormente guardada (talvez até em forma digital) e, se tudo der certo, o cliente levará, além do produto dos seu sonhos, um grande carnê.

Uma situação semelhante se passa na locadora, que guarda também dados pessoais, como os da loja, além de uma relação de outras pessoas que também poderão locar DVDs em nome de um dado cliente.

Outra situação ocorre naquela biblioteca da cidadezinha do interior, que, desprovida de computador, tem a ficha catalográfica de cada livro preenchida em um cartão. Neste, pode-se ler o nome dos autores, do livro, o número do tombo, ano de publicação, o nome do tradutor (se houver), entre outros dados.

Nas três situações acima, os mesmos dados poderiam ser anotados em papéis diferentes. Por exemplo, o vendedor poderia anotar o nome do cliente em uma folha de papel, seu endereço em outra, CPF e RG em uma terceira. Na biblioteca, os nomes dos autores estariam em uma gaveta, os títulos dos livros e ano de publicação em outra, mas em cartões separados.

Nesta situação, juntar os dados é que seria um grande problema. Cada conjunto de dados só faz sentido se estiver junto. Cada conjunto forma uma unidade. A ficha é uma unidade e cada uma é um **registro** de uma entidade única, como um cliente ou um livro.

8.3 Conceito de estruturas compostas de dados

As variáveis compostas são uma classe de variáveis que permitem agrupar dados que possuem uma relação natural de proximidade entre si, caracterizando uma unidade. Ao se fazer este agrupamento de dados distintos em um único lugar, cria-se um único "objeto", o qual faz pouco sentido se for quebrado em partes.

Os **registros**, ou **variáveis compostas heterogêneas** (que é um nome muito longo), são uma única variável formada por partes, cada uma destas partes sendo composta por outros dados. Além dos registros, há também as **variáveis compostas homogêneas**, que serão vistas na Unidade 10.

A denominação **heterogênea** quer simplesmente dizer que cada parte da variável não precisa ter o mesmo tipo. Assim, a variável composta pode ser formada por uma parte inteira, outra real e outras 2 ou 3 que sejam literais. Ou qualquer outra combinação.

8.3.1 Declaração e uso de registros

Um registro pode ser declarado em um algoritmo utilizando o formato abaixo, no qual a organização visual ainda mantém sua importância. Uma declaração de variável composta determina a criação de um **novo tipo de dados**, personalizado segundo o desenvolvedor do algoritmo.

```
registro
     lista_de_campos
fim-registro
```

A **lista de campos** é a que define as partes constituintes de um registro. O termo **campo** é usado para especificar cada uma destas partes e o formato é idêntico ao de declarações de variáveis, o que facilita o entendimento.

Um primeiro exemplo pode ser dado pelo seguinte problema: "Um aluno de geometria analítica precisa determinar, entre dois pontos em R^3, qual é o mais próximo à origem. Escreva um algoritmo que calcule as distâncias e determine qual o mais próximo.".

A solução deste problema passa por entender o que se pretende e que a distância do ponto $\mathbf{p} = [p_1, p_2, p_3]$ à origem é dada pela fórmula $d(\mathbf{p}) = \sqrt{p_1^2 + p_2^2 + p_3^2}$. Assim, o Algoritmo 8-1 propõe uma solução.

Algoritmo 8-1

```
1  { determinar, em R³, qual de dois pontos é o mais próximo da origem }
2
3  algoritmo
4        declare
5             ponto1, ponto2: registro
6                                  x, y, z: real
7                             fim-registro
```

```
8             distância1, distância2: real
9
10       { leitura das coordenadas dos pontos }
11       leia(ponto1.x, ponto1.y, ponto1.z)
12       leia(ponto2.x, ponto2.y, ponto2.z)
13
14       { cálculo das distâncias }
15       distância1 ← raiz(pot(ponto1.x, 2) + pot(ponto1.y, 2) +
16                                                  pot(ponto1.z, 2))
17       distância2 ← raiz(pot(ponto2.x, 2) + pot(ponto2.y, 2) +
18                                                  pot(ponto2.z, 2))
19
20       { comparação e resultado }
21       se distância1 < distância2 então
22            escreva("O primeiro ponto é o mais próximo")
23       senão
24            se distância2 < distância1 então
25                 escreva("O segundo ponto é o mais próximo")
26            senão
27                 escreva("Os pontos eqüidistam da origem")
28            fim-se
29       fim-se
30  fim-algoritmo
```

As variáveis **ponto1** e **ponto2** são definidas como registros. Cada uma delas contém três campos internos, sendo, no caso, todos do tipo real. Os identificadores **x**, **y** e **z** são usados para indicar as três coordenadas cartesianas que cada ponto possui.

A especificação de cada campo é feita com o **operador de acesso a campo**, que é um ponto (**.**). Por exemplo, enquanto a variável **ponto1** especifica o registro completo, **ponto1.x** especifica somente o campo **x** da variável **ponto1**, que tem o tipo **real**. A Figura 8-1 mostra um esquema que ilustra a variável **ponto1** e seus campos; para a variável **ponto2** o esquema, naturalmente, é similar.

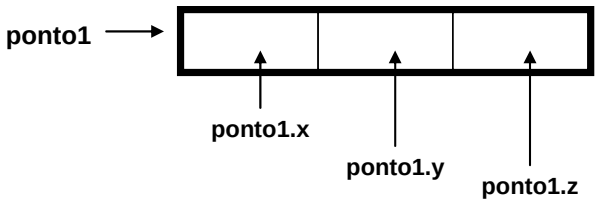


Figura 8-1. Esquema da variável ponto1**, que é um registro com três campos reais,** x**,** y **e** z**, declarada no Algoritmo 8-1.**

Neste caso, o uso do registro não é obrigatório, pois poderiam ter sido usadas variáveis separadas para cada coordenada de cada ponto, fazendo uso de seis valores reais (**x1**, **y1**, **z1**, **x2**, **y2** e **z2**, por exemplo). Porém, a questão da organização é um ponto importante e a legibilidade da solução só tem a ganhar com uma boa estruturação dos dados.

O uso de registros traz uma facilidade adicional. Para exemplificá-la, um problema similar ao anterior é proposto: "Sabendo-se que existe um conjunto de pontos e que a quantidade de elementos é também conhecida, determinar qual deles é o mais próximo da origem.". Sua solução é mostrada pelo Algoritmo 8-2.

Algoritmo 8-2

```
1  { determinar, em R³, qual de um conjunto de pontos é o mais próximo
2    da origem, sabendo-se o número de elementos do conjunto }
3
4  algoritmo
5        declare
6             ponto,
7             pontoMaisPróximo: registro
8                                  x, y, z: real
9                               fim-registro
10            distância: real
11
12       { obtenção do número de pontos }
13       leia(númeroPontos)
14
15       se númeroPontos ≤ 0 então
16            escreva("Não há pontos a serem processados ou valor inválido")
17       senão
18            { leitura das coordenadas do primeiro ponto}
19            leia(ponto.x, ponto.y, ponto.z)
20            pontoMaisPróximo ← ponto  { o primeiro é o mais próximo }
21            menorDistância ← raiz(pot(ponto.x, 2) + pot(ponto.y, 2) +
22                                                    pot(ponto.z, 2))
23
24            { processa os demais pontos }
25            para i ← 2 até númeroPontos faça
26                 leia(ponto.x, ponto.y, ponto.z)
27                 distância ← raiz(pot(ponto.x, 2) + pot(ponto.y, 2) +
28                                                    pot(ponto.z, 2))
29
30                 { comparação }
31                 se distância < menorDistância então
32                      menorDistância ← distância
33                      pontoMaisPróximo ← ponto
34                 fim-se
35            fim-para
36
37            { resultado }
38            escreva("Ponto mais próximo:", pontoMaisPróximo.x,
39                                 pontoMaisPróximo.y, pontoMaisPróximo.z)
40       fim-se
41 fim-algoritmo
```

O ponto de maior destaque são as atribuições existentes nas linhas 20 e 33. Como tanto a variável, que fica à esquerda do símbolo de atribuição (←), quanto a expressão, do lado direito, possuem o mesmo tipo, a atribuição é válida (ou seja, é feita uma cópia do conteúdo). Portanto, uma vantagem que existe com os registros é que se pode copiar um registro inteiro para outro, usando um único comando. Certamente é preciso entender que a cópia é completa, isto é, uma reprodução fiel do registro copiado é feita, incluindo campos ainda não preenchidos (que ainda não tiveram atribuição).

A parte ruim é que, para a leitura e a escrita, somente são válidas as especificações campo a campo da variável composta. Por exemplo, um comando **escreva(ponto1)** é considerado errado. Cada campo deve ser especificado individualmente, como foi feito nas linhas 19, 26 e 38 do Algoritmo 8-2.

Da mesma forma, expressões relacionais entre registros devem ser feitas identificando os campos, não sendo considerado correto comparar registros inteiros de uma vez. Afinal, qual seria o resultado de **ponto1 < ponto2**? Mesmo as comparações de igualdade ou desigualdade não são consideradas válidas (e.g., **ponto1 = ponto2**).

Outro exemplo pode ser dado pela reescrita do Algoritmo 6-6 (página 6-10), sobre o problema dos vinhos, para uma versão com registros, dada pelo Algoritmo 8-3.

Algoritmo 8-3

```
1  { dados o nome, o preço e o tipo dos vinhos (indicados aqui por "T" para
2    tinto, "B" para branco ou "R" para rosê, descrever o vinho mais caro;
3    não são considerados vinhos de preços iguais;
4    fim dos dados indicado por nome = "fim" }
5
6  algoritmo
7        tipo tVinho: registro
8                          nome,
9                          tipo: literal
10                         preço: real
11                   fim-registro
12       declare
13            vinho, vinhoCaro: tVinho
14
15       { repetição para leitura dos dados }
16       vinhoCaro.preço ← -1  { para forçar a troca na primeira vez }
17       faça
18            { dados }
19            leia(vinho.nome, vinho.preço, vinho.tipo)
20
21            { verificação do maior preço }
22            se vinho.preço > vinhoCaro.preço e vinho.nome ≠ "fim" então
23                 vinhoCaro ← vinho   { copia tudo }
24            fim-se
25       até vinho.nome = "fim"
```

```
26
27       { apresentação do resultado }
28       se vinhoCaro.preço = -1 então
29            escreva("Nenhum vinho foi apresentado.")
30       senão
31            escreva(vinhoCaro.nome, vinhoCaro.preço, vinhoCaro.tipo)
32       fim-se
33 fim-algoritmo
```

A solução é similar à apresentada anteriormente, modificada para que os dados sobre uma dada garrafa de vinho formem uma única entidade, no caso o registro.

A novidade apresentada no Algoritmo 8-3 é a possibilidade da **definição de novos tipos**, o que é feito de forma similar a uma declaração de variável. As linhas de 9 a 11 mostram como a declaração **tipo** define um novo tipo abstrato de dados, personalizado pelo desenvolvedor. No exemplo, o novo tipo recebe o nome de **tVinho** (que indica, pelo *t*, que é um tipo[*]).

Na Figura 8-2 podem ser observados alguns exemplos de declarações de tipos usando registros. Em especial, pode-se indicar no terceiro exemplo o uso de registros como campos de outro registro, como o uso de **tTransação** e **tData**. O acesso a campos internos também é feito com o operador **.**, como por exemplo **operação.transaçãoPrincipal.valor**, com **operação** sendo uma variável declarada do tipo **tOperaçãoBancária**. Não custa lembrar que **operação.transaçãoPrincipal** é um registro do tipo **tTransação** e, portanto, pode ser atribuído diretamente a outra variável ou campo com o mesmo tipo.

```
tipo tDada: registro
               dia, mês, ano: inteiro
            fim-registro
```

```
tipo tAluno: registro
                nome: literal
                código: inteiro
                cidadeNascimento,
                cidadeResidência: literal
                rg: literal
             fim-registro
```

```
tipo tTransação: registro
                    operação: literal { entrada/saída }
                    valor: real
                 fim-registro

    tOperaçãoBancária: registro
                            saldo: real
                            limiteEspecial: real
```

[*] A notação que usa o *t* para indicar um tipo é uma opção de estilo, e não uma regra. A adoção desta estratégia é, portanto, opcional. Ressalta-se, porém, que a diferenciação entre variáveis e tipos ajuda na clareza e permite uma melhor organização do algoritmo, servindo como auto-documentação.

```
                                transacaoPrincipal: tTransação
                                data: tData
                            fim-registro

declare operação: tOperaçãoBancária    { variável }
```

Figura 8-2. Exemplos de declarações de tipos, incluindo o uso de outros registros como campos.

8.4 Considerações finais

Os registros são uma forma imprescindível de organização dos algoritmos. Ajudam a organizar e estruturar os dados de modo que a leitura do algoritmo fica facilitada, pois o código se torna praticamente auto-documentado.

Ainda mais que organizar, os registros também permitem facilidades importantes, como será observado nas Unidades seguintes.

Um último comentário: nem sempre é preciso que os dados sejam armazenados em registros; agrupá-los é uma decisão organizacional, de forma que extremos para menos (nenhum agrupamento quando seria bom fazê-los) ou para mais (agrupar tanto os dados, que se torna complicado usá-los) podem ser prejudiciais. A prática e o bom senso ainda são a melhor regra.

Unidade 9 - Sub-rotinas

9.1 Primeiras palavras

Neste ponto, entre aquilo que será visto na disciplina, não há mais novidades em termos de comandos. Atribuição, leitura e escrita, condicionais e repetições compõem, praticamente, todo o quadro de comandos que estão disponíveis para o desenvolvimento dos algoritmos.

Esta Unidade não traz novidades, portanto, em termos de comando, mas sim de organização.

O assunto, agora, é separar conjuntos de comandos que constituem uma **unidade funcional**. A organização se baseia em resolver sub-problemas, ou seja, partes de um problema maior, e separar estes comandos em uma **sub-rotina**.

A organização em sub-rotinas, porém, não é um ponto simples, pois envolve uma grande quantidade de detalhes. Cada um destes detalhes será tratado no momento devido.

9.2 Separando para organizar

Um jantar mais formal deve ser feito com cuidados e envolve uma série de considerações. Não se pode esquecer de alguns aperitivos na recepção, como azeitonas, queijo em cubinhos, patê com torradas. Também algumas bebidas. Uma salada tem que estar disponível e bem apresentada. Considera-se sobre um prato de entrada e sobre o prato principal, além da bebida que os acompanha. Finalmente, e não menos importante, é a sobremesa.

Caso se pretenda pedir a uma pessoa que execute a tarefa de preparar cada uma destas coisas, que não são poucas, é preciso orientá-la bem. É necessário, também, passar-lhe as receitas.

Com as orientações gerais sobre o que fazer, sabe-se o rumo do jantar. Com as receitas individuais, tem-se o conhecimento para preparar cada prato.

Na prática, as orientações gerais são passadas oralmente ou, então, por escrito. Separadamente, ficam as receitas. Cada receita, também, costuma vir em um papel separado ou ter páginas específicas em um caderno ou livro. É complicado pensar que, nas receitas, todos os ingredientes sejam listados e, a seguir, tenha um conjunto de instruções que permitam fazer, ao mesmo tempo, a salada, a sopa de entrada, o prato principal e a sobremesa. Tudo vem separado, organizado.

Separar também é organizar. E ajuda bastante.

9.3 Conceituação de sub-rotinas e fluxo de execução

Nos algoritmos, separar para organizar significa redefinir o velho conhecido fluxo de execução. Até este ponto, o fluxo é contínuo, iniciando pelo primeiro comando e indo seqüencialmente até o último.

Porém, de forma parecida à organização de um jantar (e instruções necessárias para prepará-lo), algumas coisas podem ser indicadas separadamente. Por exemplo, o prato principal pode ser um mussaká, e, nas instruções gerais, haver somente uma indicação: "faça o mussaká". Será na receita do mussaká que estarão listados todos os ingredientes e o modo de fazer.

O **fluxo de execução** segue, assim, as instruções principais (que equivalem ao algoritmo) e, em um dado momento, suspende esta execução para executar uma tarefa específica, que neste caso é a preparação do prato principal (ou seja, uma sub-rotina). Terminada a tarefa, as instruções gerais são retomadas no ponto seguinte àquele em que foram temporariamente suspensas.

Nos algoritmos, uma **sub-rotina** é um conjunto completo de instruções, com uma função bem definida no contexto do algoritmo. O algoritmo principal, quando preciso, apenas indica que aquele conjunto de comandos deve ser executado. Após esta execução, o fluxo dos comandos é retomado.

Em geral, as sub-rotinas servem aos seguintes propósitos:

1) Separar conjuntos de comandos de forma a obter uma melhor organização da solução, ajudando na clareza e no entendimento do algoritmo;
2) Separar conjuntos de comandos que se repitam em várias partes do algoritmo, permitindo indicar uma única vez a solução e indicar vários lugares diferentes em que ela deva ser aplicada;
3) Separar conjuntos de comandos que realizem uma tarefa, simples ou complexa, de forma completa; deste modo, uma solução feita para um problema pode ser usada em outro, minimizando esforços.

Mas para entender como isso acontece, o melhor é tomar um exemplo. Para tanto, pode-se considerar o seguinte problema: "Escreva um algoritmo para ler dois números racionais e escrever o valor de sua soma e sua multiplicação. Assuma que os valores lidos são válidos (ou seja, o denominador nunca é nulo) e que ambos os valores sejam diferentes de zero.". O Algoritmo 9-1 apresenta uma solução para o problema.

Algoritmo 9-1

```
1  { a partir de dois números racionais, calcular e apresentar a soma e a
2    multiplicação de ambos }
3
4  algoritmo
5        tipo tRacional: registro
6                              numerador, denominador: inteiro
7                        fim-registro
8
9        declare
10            número1, número2, resultado: tRacional
11            valor1, valor2, resto: inteiro
12
13       { obtenção dos números }
```

```
14       leia(número1.numerador, número1.denominador)
15       leia(número2.numerador, número2.denominador)
16
17       { cálculo da soma }
18       resultado.numerador ← número1.numerador * número2.denominador +
19                                 número1.denominador * número2.numerador
20       resultado.denominador ← número1.denominador * número2.denominador
21
22       { cálculo do MDC }
23       valor1 ← resultado.numerador
24       valor2 ← resultado.denominador
25       faça
26            resto ← valor1 % valor2
27            valor1 ← valor2
28            valor2 ← resto
29       até resto = 0            { resultado do MDC fica em valor1 }
30
31       { simplificação da razão }
32       resultado.numerador ← resultado.numerador/valor1
33       resultado.denominador ← resultado.denominador/valor1
34
35       { escrita da soma }
36       escreva(resultado.numerador, "/", resultado.denominador)
37
38       { cálculo do produto }
39       resultado.numerador ← número1.numerador * número2.numerador
40       resultado.denominador ← número1.denominador * número2.denominador
41
42       { cálculo do MDC }
43       valor1 ← resultado.numerador
44       valor2 ← resultado.denominador
45       faça
46            resto ← valor1 % valor2
47            valor1 ← valor2
48            valor2 ← resto
49       até resto = 0            { resultado do MDC fica em valor1 }
50
51       { simplificação da razão }
52       resultado.numerador ← resultado.numerador/valor1
53       resultado.denominador ← resultado.denominador/valor1
54
55       { escrita do produto }
56       escreva(resultado.numerador, "/", resultado.denominador)
57  fim-algoritmo
```

Este algoritmo calcula a soma e o produto e, para melhor apresentar os resultados, racionaliza as razões resultantes, de forma que, se o resultado for 2/6, a escrita será de 1/3.

O que se pode notar de imediato é que, como são necessárias duas simplificações, todo um conjunto de comandos deve ser repetido. Uma forma de simplificar um pouco este código foi usar a mesma variável para guardar tanto o resultado da soma quanto o da multiplicação. Deve-se ainda pensar que, caso fossem solicitadas outras operações (subtração e divisão, por exemplo), novas racionalizações de frações teriam que ser incorporadas.

O ponto principal, aqui, é que a racionalização de uma razão é sempre feita da mesma forma e, assim, poderia ser escrita uma única vez e, quando necessário, teria indicada uma solicitação de sua execução.

O Algoritmo 9-2 mostra uma versão modificada do Algoritmo 9-1, incluindo sub-rotinas para a racionalização de frações. Esta solução está, porém, repleta de particularidades, as quais serão detalhadas no restante da Unidade.

Algoritmo 9-2

```
1  { a partir de dois números racionais, calcular e apresentar a soma e a
2    multiplicação de ambos }
3
4  { definição do tipo }
5  tipo tRacional: registro
6                      numerador, denominador: inteiro
7                  fim-registro
8
9  procedimento simplifiqueRacional(var racional: tRacional)
10 { modifica um racional para que fique na forma mais simples }
11       declare valor1, valor2, resto: inteiro
12
13       { cálculo do MDC }
14       valor1 ← racional.numerador
15       valor2 ← racional.denominador
16       faça
17            resto ← valor1 % valor2
18            valor1 ← valor2
19            valor2 ← resto
20       até resto = 0            { resultado do MDC fica em valor1 }
21
22       { simplificação da razão }
23       racional.numerador ← racional.numerador/valor1
24       racional.denominador ← racional.denominador/valor1
25 fim-procedimento
26
27 algoritmo
28       declare número1, número2, resultado: tRacional
29
30       { obtenção dos números }
31       leia(número1.numerador, número1.denominador)
32       leia(número2.numerador, número2.denominador)
33
```

```
34      { cálculo da soma }
35      resultado.numerador ← número1.numerador * número2.denominador +
36                            número1.denominador * número2.numerador
37      resultado.denominador ← número1.denominador * número2.denominador
38      simplifiqueRacional(resultado)
39
40      { escrita da soma }
41      escreva(resultado.numerador, "/", resultado.denominador)
42
43      { cálculo do produto }
44      resultado.numerador ← número1.numerador * número2.numerador
45      resultado.denominador ← número1.denominador * número2.denominador
46      simplifiqueRacional(resultado)
47
48      { escrita do produto }
49      escreva(resultado.numerador, "/", resultado.denominador)
50 fim-algoritmo
```

As linhas de 9 a 25 indicam uma sub-rotina na forma de um **procedimento**. Cada procedimento tem seu próprio nome, que é um identificador; neste caso, o nome é **simplifiqueRacional**, para deixar clara sua função. A sub-rotina define sobre o que trabalhará: uma variável do tipo **tRacional** chamada simplesmente de **racional** (ainda na linha 9). No bloco definido pelo par **procedimento**/**fim-procedimento**, está tudo o que é necessário para completar o propósito da sub-rotina: um comentário de documentação, todas as variáveis necessárias e os comandos que devem ser executados.

É importante notar que o algoritmo continua se iniciando com o primeiro comando após a palavra **algoritmo**, ou seja, começa na linha 31, após a declaração das variáveis. O procedimento é apenas uma **declaração de sub-rotina**. Em outras palavras, é apenas uma indicação do que deve ser feito e não que os comandos devam ser executados antes do algoritmo. Correspondem, portanto, apenas à especificação das instruções. Cada sub-rotina fica, assim, conhecida com antecedência, mas somente é executada quando houver uma chamada explícita indicando esta ação.

Nas linhas 38 e 46 é que se encontram as solicitações de execução, conhecidas como **chamadas de procedimento**, pontos nos quais é feita a solicitação de execução do código anteriormente declarado Nelas estão as indicações que o **simplifiqueRacional** tem que ser executado, além de informar que as ações devem ser feitas sobre a variável **resultado**. Para esclarecer: a declaração do procedimento feita antes do algoritmo é uma solução genérica; é na chamada do procedimento que se especifica com clareza sobre qual variável (ou variáveis) efetivamente se dará o processamento. Mais detalhes ainda serão discutidos; portanto, é preciso um pouco de paciência do leitor.

Por agora, é preciso atentar que as chamadas feitas a **simplifiqueRacional** alteraram o fluxo de execução para o código da sub-rotina (declarando suas variáveis e executando seus comandos) e retornaram, depois, à execução normal, retomando a execução seqüencial

dos comandos. A chamada funciona como se houvesse sido definido um novo comando.

E voltando, ainda, à hipótese de se precisar incluir também uma subtração e uma divisão de racionais, as simplificações seriam resolvidas apenas por chamar o procedimento novamente após cada cálculo.

Como exercício, fica para o leitor considerar que outras coisas existem no Algoritmo 9-2 que sejam também repetitivas e, assim, possam ser transformadas em sub-rotinas.

É conveniente, ainda, considerar uma sub-rotina como uma solução genérica para um problema específico (como a simplificação do número racional). Sob este ângulo, as sub-rotinas devem ser completas, isto é, ter explícitas todas as suas necessidades internamente. No Algoritmo 9-2, a sub-rotina **simplifiqueRacional** é completa, pois tem, em sua definição, todos os elementos de que precisa: identifica o parâmetro sobre o qual a operação será realizada, as declarações de todas as variáveis necessárias para a solução do problema e todo o código que deve ser executado para cumprir sua tarefa. Deve-se notar que a sub-rotina quase que independe do algoritmo principal; ou seja, pode ser usada em qualquer algoritmo que tenha um tipo **tRacional** definido como na linha 5. Esta é a única dependência externa da sub-rotina: o tipo de seu parâmetro.

9.3.1 Procedimentos

Um procedimento é uma forma de sub-rotina. Sua forma genérica, apresentada abaixo, delimita a especificação da sub-rotina.

```
procedimento identificador(lista_de_parâmetros)
{ comentário de descrição do procedimento }
     declarações_locais

     conjunto_de_comandos
fim-procedimento
```

O **identificador** é o nome dado à sub-rotina, que deve ser não conflitante com as demais declarações (como tipos ou variáveis). A **lista de parâmetros**, que pode ser vazia, lista os **parâmetros formais** do procedimento (detalhados na próxima seção). As declarações locais correspondem a quaisquer declarações (tipos, variáveis, sub-rotinas etc.) que sejam exclusivas à sub-rotina.

Para especificar a lista de parâmetros, deve-se usar uma lista de identificadores com seus respectivos tipos, separados por vírgulas. Cada parâmetro deve, portanto, ter um tipo associado a ele.

O **conjunto de comandos**, finalmente, corresponde à lista de comandos que deve ser executada para completar o propósito da sub-rotina. Estes comandos manipulam, assim, os parâmetros e as declarações locais para obter a solução desejada.

9.3.2 Parâmetros formais

Para compreender com precisão o que acontece na chamada de uma sub-rotina é preciso compreender o papel dos **parâmetros** que elas podem ter. No exemplo do Algoritmo 9-2, o procedimento **simplifiqueRacional** tem apenas o parâmetro **racional** do tipo **tRacional**.

O parâmetro tem a função de permitir uma solução genérica para o código da sub-rotina e, no momento da chamada, é indicado o real **argumento** sobre o qual as ações serão realizadas. Voltando ao exemplo da racionalização, **racional** é o parâmetro genérico e as linhas 38 e 46 especificam que o que é feito com **racional**, na realidade, é feito com **resultado**.

Os parâmetros podem obedecer duas formas distintas de comportamento, caracterizando parâmetros com **passagem por valor** ou **passagem por referência**. O exemplo visto utiliza o parâmetro passado por referência.

Passagem de parâmetros por referência indica que, na sub-rotina, o parâmetro formal é uma **referência** à variável utilizada no momento da chamada. O termo *referência* quer dizer que o parâmetro efetivamente corresponde à variável real, sendo ambas a mesma coisa. Em outras palavras, tanto o identificador da variável (**racional**, no exemplo) quanto o do parâmetro (**resultado**) fazem *referência* à mesma posição de memória. O efeito disso é que modificações feitas no parâmetro formal são feitas, na verdade, na variável externa. A principal utilização de passagem por referência é poder modificar o valor das variáveis dentro de uma sub-rotina, de modo que as alterações feitas internamente fiquem refletidas externamente à sub-rotina.

Por outro lado, a **passagem de parâmetros por valor** não faz a ligação entre o parâmetro e a variável usada na chamada. É feita a **cópia do valor** da variável externa para o parâmetro, o qual possui sua própria área de memória para armazenamento. A conseqüência disso é que, feitas alterações pela sub-rotina no valor do parâmetro formal, somente a cópia (o parâmetro) é afetada, sendo que o valor original do argumento usado na chamada é preservado. Assim, sempre que não houver necessidade de que um valor alterado em uma sub-rotina seja conhecido externamente a ela, a passagem por valor deve ser empregada.

Para diferenciar as duas formas de passagem de parâmetros, a palavra reservada **var** é usada para indicar, para cada parâmetro individualmente, que se trata de uma passagem por referência (com alteração da **var**iável externa). Se não houver indicação, então a passagem será por valor.

O Algoritmo 9-3 ilustra as duas formas de passagem de parâmetros.

Algoritmo 9-3

```
1  { algoritmo simples (e sem função específica) que ilustra o formato
2    e as conseqüências de sub-rotinas com passagem de parâmetros por
3    valor e por referência }
4
5  procedimento escrevaInteiroEFracionário(valor: real)
```

```
6  { escreve as partes inteira e fracionária de um valor }
7
8        escreva("Parte inteira:", int(valor))
9        valor ← valor – int(valor)
10       escreva("Parte fracionária:", valor)
11 fim-procedimento
12
13 procedimento transformeEmPositivo(var valor: real)
14 { transforma o valor de um número real em positivo }
15
16       se valor < 0 então
17            valor ← -valor   { transforma em positivo }
18       fim-se
19 fim-procedimento
20
21 algoritmo
22       declare número: real
23
24       { o valor "especial" 999 indica o término }
25       faça
26            leia(número)
27
28            escrevaInteiroEFracionário(número)
29
30            escreva("O módulo é:")
31            transformeEmPositivo(número)
32            escreva(número)
33       enquanto número ≠ 999
34 fim-algoritmo
```

O procedimento **escrevaInteiroEFracionário** possui um parâmetro formal, com passagem por valor. No código desta sub-rotina é usado um artifício simples para se obter a parte fracionária do número, sendo este valor armazenado na própria variável que corresponde ao parâmetro. Como a alteração é feita em uma cópia do conteúdo, não há qualquer efeito da atribuição da linha 9 sobre a variável **número**, usada na chamada de procedimento.

O modificador **var**, na declaração do procedimento **transformeEmPositivo** (linha 13), por sua vez, proporciona um comportamento diverso para a atribuição que é feita internamente ao procedimento. Como o parâmetro é uma referência direta à variável usada na chamada do procedimento (neste caso, a variável **número**), a modificação no valor do parâmetro **valor** implica a modificação da variável **número**, já que ambas se referem diretamente à mesma posição de memória. Não há como modificar uma sem modificar a outra.

Em termos gerais, usa-se o termo **parâmetros de entrada** para aqueles que "carregam" informações para o interior da sub-rotina. **Parâmetros de saída**, por seu turno, referem-se aos parâmetros que transferem dados para o exterior da sub-rotina. A passagem por valor é

usada para parâmetros de entrada. Parâmetros de saída exigem a passagem por referência. Deve-se notar, ainda, que no exemplo do Algoritmo 9-3, o parâmetro formal **valor** de transformeEmPositivo é usado tanto quanto parâmetro de entrada (quando leva por referência um valor externo que é usado na sub-rotina) quanto de saída (ao fazer que a alteração feita internamente tenha abrangência fora do escopo do procedimento).

Algumas vezes, parâmetros passados por referência são exclusivamente de saída. Isto pode ser identificado quando o valor armazenado na variável antes da chamada da sub-rotina não interfere com o resultado produzido por ela. Portanto, somente há a *saída* de dados, e não a *entrada*.

A declaração da lista de parâmetros é feita, como visto, nos parênteses que seguem o identificador da sub-rotina. Todos os parâmetros devem ter seu tipo especificado, assim como a forma de passagem (por valor ou por referência). É possível, caso alguns parâmetros consecutivos compartilhem tanto o tipo quanto a forma de passagem, especificar uma única vez o tipo e a forma. A Tabela 9-I mostra alguns exemplos genéricos de declarações de parâmetros, algumas separando os tipos para cada argumento, outras os agrupando (quando compartilharem modo de passagem e tipo).

Tabela 9-I. Exemplos de declarações de parâmetros.

Exemplo de lista de parâmetros	Comentário
`p1, p2: inteiro`	Dois parâmetros inteiros, por valor.
`p1: inteiro, p2: inteiro`	Dois parâmetros inteiros por valor (idêntica ao exemplo anterior).
`p1: real, p2: inteiro, p2: real`	Três parâmetros, sendo o primeiro e o último reais e o segundo inteiro, todos por valor
`p1, p2, p3: tTipo`	Três parâmetros do tipo tTipo, por valor.
`var p1, p2: literal`	Dois parâmetros literais, ambos por referência.
`var p1: literal, p2: literal`	Um parâmetro literal por referência (o primeiro) e outro, também literal, por valor.
`p1: literal, var p2: literal`	Dois parâmetros literais, o primeiro por valor e o segundo por referência.
`p1: real, var p2: real, p3: real`	Três parâmetros, todos reais, sendo o primeiro e terceiro por valor e somente o segundo por referência.

9.3.3 Regras de escopo de declarações

Ao longo dos exemplos de algoritmos apresentados nesta Unidade, é possível observar que as declarações são feitas em vários **locais** diferentes dentro do algoritmo. E o termo *declarações*, aqui, não se refere somente a declarações de variáveis, mas também a **declarações de tipo e de sub-rotinas**.

O Algoritmo 9-2 será usado para exemplificar estes locais de declaração. Nele existem variáveis que pertencem somente à parte principal

do algoritmo e não fazem sentido nem são usadas fora dele, que são declaradas internamente ao par **algoritmo/fim-algoritmo**. Estas declarações ocorrem na linha 28. Outras variáveis pertencem apenas ao procedimento **simplifiqueRacional**, correspondendo ao parâmetro da linha 9 e às variáveis declaradas na linha 11. Além das variáveis, existem a declaração do tipo **tRacional** (linhas 5 a 7) e a declaração da sub-rotina que correspondem ao procedimento como um todo (com início na linha 9 e conclusão na linha 25).

As declarações feitas *internamente*, como os parâmetros e as variáveis internas a uma sub-rotina, além daquelas que pertencem exclusivamente à parte principal do algoritmo, são conhecidas como **declarações locais** (linhas 9, 11 e 28 do Algoritmo 9-2, como exemplo). As demais declarações, feitas fora deste **escopo**[*] local são as **declarações globais** (que são as declarações do tipo e do procedimento).

No caso do Algoritmo 9-3, os parâmetros dos dois procedimentos (ambos identificados por **valor**) têm escopo local, como é o caso da variável **número**, que é local à parte principal do algoritmo. Além disso, são globais as declarações dos próprios procedimentos.

Algumas **regras de escopo** devem ser definidas para que não existam ambigüidades de interpretação quando da execução de um comando em qualquer parte do algoritmo.

Definem-se, assim, as seguintes regras:

1. Qualquer declaração somente é válida e visível após ter sido feita e se torna inválida depois de encerrado seu escopo.
2. Declarações locais têm como escopo somente a sub-rotina dentro da qual foram declaradas, assim como declarações locais à parte principal do algoritmo somente são válidas e visíveis neste trecho específico.
3. Parâmetros formais de sub-rotinas obedecem à mesma regra de escopo das declarações locais da sub-rotina.
4. Declarações globais são válidas e visíveis do ponto onde foram feitas até o final do algoritmo.
5. Identificadores com o mesmo nome podem ser usados, desde que pertençam a escopos diferentes.
6. Quando um mesmo identificador é definido para escopos diferentes e ambos possuem escopo em um dado local, o identificador válido é sempre o do escopo mais local.

A Figura 9-1 apresenta um algoritmo simplificado com várias declarações, todas válidas. Existem seis declarações globais: o tipo **tTipo**, as variáveis **umInteiro**, **valor** e **outroInteiro**, além dos procedimentos **meuProcedimento** e **seuProcedimento**. Estes dois últimos (os procedimentos) definem dois novos escopos locais. Para **meuProcedimento**, há os parâmetros **v1** (inteiro) e **v2** (real), mais uma variável local denominada **umInteiro**. O escopo local definido para **seuProcedimento** declara os parâmetros **umInteiro** e **valor** (ambos

[*] **Escopo**, no contexto deste texto, pode ter seu significado entendido como **área de abrangência**.

inteiros) e duas variáveis locais do tipo **tTipo**, identificadas por **v1** e **v2**. Há, finalmente, um terceiro escopo local, definido para a parte principal do algoritmo, na qual se declara a variável **valor** (do tipo **tTipo**).

A declaração de **tTipo** é válida para todo o algoritmo, tanto que é utilizada internamente a **seuProcedimento** e também na parte principal do algoritmo.

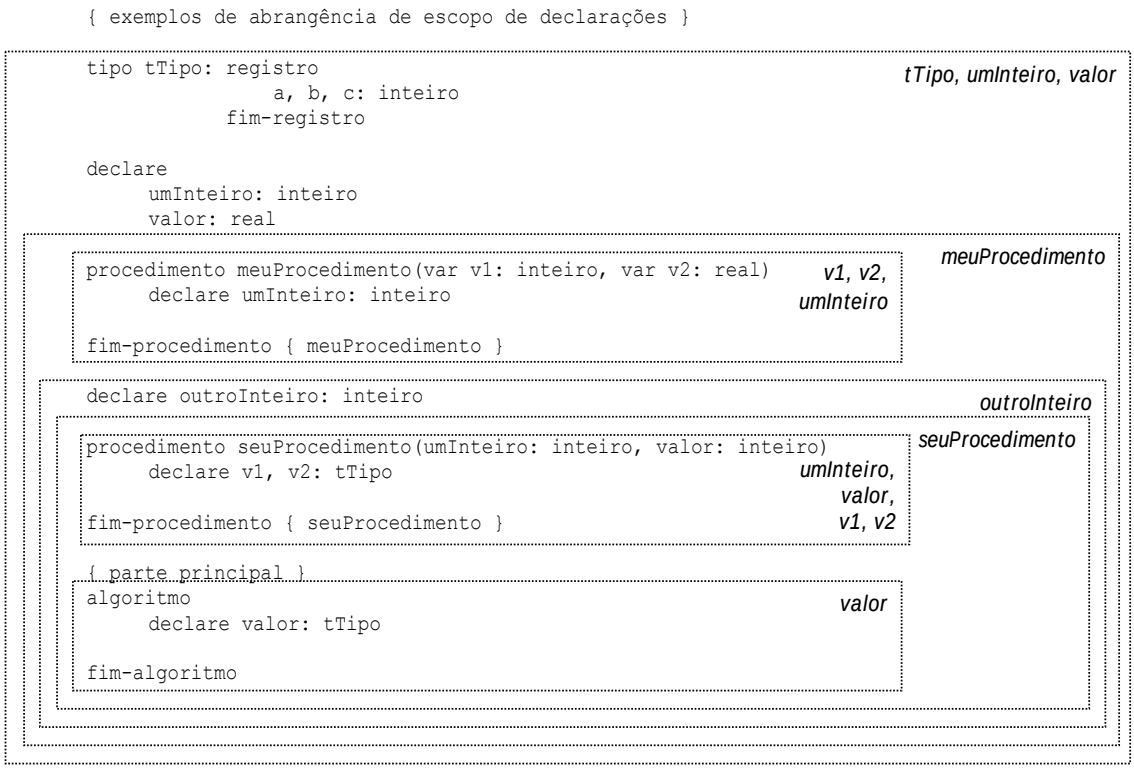


Figura 9-1. Exemplo de escopo de declarações.

As variáveis **umInteiro** e **valor** valem para todo o algoritmo, assim como **tTipo**. Porém, o identificador **umInteiro** é substituído pela variável local na sub-rotina **meuProcedimento** e pelo parâmetro, em **seuProcedimento**. Em cada escopo, passa a valer a declaração local, o que interrompe o acesso à variável global até o final de cada procedimento. Fora destes locais, a declaração global é que vale. O mesmo ocorre pela substituição da variável global **valor** dentro de **seuProcedimento** (onde é sobreposta pelo parâmetro) e dentro da parte principal do algoritmo (substituída por outra variável, agora do tipo **tTipo**).

A declaração da variável **outroInteiro** é também global, porém somente válida a partir de sua declaração; ou seja, não está definida para a sub-rotina **meuProcedimento**.

Finalmente, ainda é importante destacar que os procedimentos também têm seus escopos. Assim, **meuProcedimento** pode ser usado dentro de **seuProcedimento** e na parte principal do algoritmo. A sub-rotina **seuProcedimento**, por sua vez, é global e pode ser usada na parte principal, porém não é válida dentro de **meuProcedimento**, uma vez que, a esta altura, ainda não foi declarada.

9.3.4 Funções

Outro tipo de sub-rotina são as **funções**. Um procedimento substitui um comando, como se um novo comando fosse criado. As funções substituem um valor.

Na Unidade 3 foram apresentadas várias funções pré-definidas*. Esta seção apresenta uma forma de se escrever uma sub-rotina que pode ser usada como uma nova função. Para tanto, é usado o par **função/fim-função** para indicar o novo código. O formato é apresentado na seqüência.

```
função identificador(lista_de_parâmetros): tipo_da_função
{ comentário de descrição da função }
     declarações_locais

     conjunto_de_comandos
fim-função
```

Para exemplificar, o Algoritmo 9-4 ilustra as novas funções **menorInteiro**, **menorReal** e **módulo**.

A função **menorInteiro** resulta no menor entre dois valores inteiros; havendo igualdade, o valor comum é o resultado. **menorReal** funciona de modo similar, mas usa parâmetros e provê o resultado do tipo **real**. Finalmente, **módulo** é uma função que retorna o valor positivo de um argumento (no caso, um **real**). É claro que já existe a função **abs**, mas **módulo** é um exemplo de como ela poderia ter sido implementada.

Algoritmo 9-4

```
1  { exemplificação de sub-rotinas na forma de função e seu uso }
2
3  função menorInteiro(valor1: inteiro, valor2: inteiro): inteiro
4  { retorna o menor entre valor1 e valor2; se iguais retorna um deles }
5
6        se valor1 < valor2 então
7             retorne valor1
8        senão
9             retorne valor2
10       fim-se
11 fim-função
12
13 função menorReal(valor1: real, valor2: real): real
14 { retorna o menor entre valor1 e valor2; se iguais, retorna um deles }
15
16       se valor1 < valor2 então
17            retorne valor1
18       senão
19            retorne valor2
```

* Ver Tabela 3-I, sobre operações aritméticas e Tabela 3-II, sobre operações literais.

```
20       fim-se
21  fim-função
22
23  função módulo(valor: real): real
24  { retorna o valor absoluto do valor }
25
26       se valor < 0 então
27            valor ← -valor
28       fim-se
29       retorne valor
30  fim-função
31
32  { parte principal }
33  algoritmo
34       declare
35            primeiroInt, segundoInt: inteiro
36            primeiroReal, segundoReal: real
37
38       { entrada de dados }
39       leia(primeiroInt, segundoInt, primeiroReal, segundoReal)
40
41       { algumas saídas e manipulações }
42       escreva("O menor inteiro entre", primeiroInt, "e",
43                 segundoInt, "é", menor(primeiroInt, aegundoInt))
44
45       se menorReal(primeiroReal, segundoReal) ≠ primeiroReal então
46            escreva(segundoReal, "é menor que", primeiroReal)
47       fim-se
48
49       se módulo(primeiroReal) = primeiroReal e primeiroReal ≠ 0 então
50            escreva("O valor", primeiroReal, "é positivo")
51       senão
52            escreva("O valor", primeiroReal, "não é positivo")
53       fim-se
54
55       escreva("Considerando-se o módulo, tem-se que o menor entre",
56                 primeiroReal, "e", segundoReal, "é",
57                 menorReal(módulo(primeiroReal), módulo(segundoReal))
58  fim-algoritmo
```

Os principais pontos que diferenciam um procedimento de uma função são:

- Uma função deve ter um tipo explícito de retorno, o qual é indicado em sua declaração.
- O resultado (o valor final) de uma função é determinado pela execução do comando **retorne**.

O comando **retorne** somente faz sentido no interior de uma função e tem como argumento uma expressão qualquer, cujo valor final deve

coincidir com o tipo de retorno da função. Outro ponto importante é que sua execução encerra a sub-rotina, em qualquer posição que esteja.

Por fim, regras de escopo e passagem de parâmetros são idênticas tanto para procedimentos quanto para funções. Por exemplo, uma função, além de retornar um valor, como é seu papel, pode ainda modificar os argumentos que forem passados por referência.

9.4 Considerações finais

Uma das ferramentas mais importantes no desenvolvimento de algoritmos são as sub-rotinas. Estas permitem organizar os algoritmos de diversas maneiras, tanto pela redução de código repetido quanto pela organização em si.

Diferenciar procedimentos de funções também é importante. Uma forma simples de se pensar sobre o assunto é ver procedimentos como definições de novos comandos, enquanto as funções são usadas apenas como resultados em expressões.

Estruturados adequadamente, os algoritmos podem ter seus refinamentos transformados em sub-rotinas durante seu desenvolvimento ou estas podem ser inseridas quando o código final for avaliado para verificar se há espaço para melhorias. A prática faz com que, já no desenvolvimento, muitas partes da solução sejam diretamente estruturadas como sub-rotinas.

Muita atenção deve ser dada a este tópico dentro da disciplina. Como será notado, todo desenvolvimento, a partir de agora, já incorporará procedimentos e funções como solução para a proposta de algoritmos.

Unidade 10 - Estruturas compostas homogêneas: arranjos

10.1 Primeiras palavras

É bastante comum a situação em que um conjunto de dados de entrada é processado para gerar uma saída (resultado). É isso que foi apresentado até agora. Há situações, entretanto, em que o conjunto de dados não pode ser "esquecido" à medida que o processamento for ocorrendo.

Em algoritmos, é bastante comum o caso em que, feito um processamento, é preciso voltar ao conjunto de dados para checar uma situação ou refazer um cálculo.

As **variáveis compostas homogêneas** são uma forma importantíssima de organização de dados (ou **estruturação de dados**) que têm como função guardar vários dados do mesmo tipo e permitir fácil acesso a cada item em particular.

10.2 Lembrar é, muitas vezes, importante

Pode-se considerar, inicialmente, a seguinte proposição para um algoritmo: "Ao final de uma competição esportiva se dispõe, para cada país, do número de medalhas de ouro, de prata e de bronze conquistadas durante a competição. Deseja-se saber qual país obteve o melhor desempenho e qual obteve o pior. Deve-se considerar que não houve empates.". Para solucionar, basta fazer uma repetição obtendo os dados país a país, comparando se a situação de cada nova entrada é melhor que o melhor país até o momento ou pior que o lanterninha registrado; se for necessário, substituir o melhor ou o pior, prosseguindo a repetição. Ao final, escrever os dados armazenados para o melhor e para o pior desempenhos.

Porém, considere a mesma proposta, mas sem a restrição de que não haja empates. O problema muda consideravelmente. A solução, agora, não é mais armazenar o melhor e o pior. Pode haver dois países empatados na primeira posição e outros quatro empatados em último. A resposta, então, deve ser uma lista dos países em primeiro lugar no *ranking* e uma lista dos últimos classificados. Na realidade, pode ser que haja um empate e que todos obtenham o mesmo número de medalhas e todos empatem em primeiro (improvável, mas possível). Uma solução para o algoritmo, nesta nova situação, é (de novo) verificar as condições de medalhas que definem o primeiro e o último colocados; depois, a lista de todos os países deve ser repassada, listando todos aqueles que coincidem com o *status* de melhor de todos, fazendo um procedimento similar para verificar os que se encontram na situação de pior desempenho. Nesta solução, é preciso "se lembrar" de vários dos dados (nome do país, número de cada tipo de medalha), para confrontar no final.

10.3 Estruturação dos dados

As **variáveis compostas homogêneas** formam uma classe de organização de dados similar às variáveis compostas heterogêneas (ou registros, vistos na Unidade 8), pois têm como função usar uma única variável para armazenar vários dados individuais. As semelhanças terminam aí.

Para simplificar, o termo usado para identificar este novo tipo de variável será **arranjo**[*].

Assim, um arranjo é uma única variável, a qual é formada por um conjunto homogêneo (isto é, do mesmo tipo) de valores. Para identificar cada valor é usado um número inteiro, que corresponde à posição do valor em relação aos demais. Este número, apropriadamente, é chamado de **índice**.

Portanto, pode-se pensar em um arranjo com capacidade de armazenar, por exemplo, 10 nomes. O arranjo formado é um arranjo de literais. Cada nome individual é identificado por seu índice, para o qual se estabelece, como convenção, que a numeração se inicia em **zero**. Há, desta forma, um literal na **posição** zero, outro na posição um, seguindo assim até a última posição, que tem índice 9.

As diversas linguagens de programação existentes apresentam variações no modo como os índices são numerados. Algumas marcariam índices de 1 a 10, para o exemplo acima, outras permitiriam marcações menos intuitivas, como de -5 a 4. O ponto principal, em que a grande maioria das linguagens concorda, é que o índice é um valor inteiro e que os valores são sempre consecutivos (ou seja, não "pulam" nenhum valor).

Para os algoritmos usados no contexto desta disciplina, todo arranjo de *k* elementos terá cada um deles identificado pelos índices de 0 a *k* – 1, sem exceções.

10.3.1 Declaração e uso de arranjos unidimensionais

Para declarar um **arranjo unidimensional** em um algoritmo, é simplesmente usada uma indicação do número de elementos que a variável composta armazenará, o que é feito por um valor inteiro entre colchetes posicionado imediatamente após o identificador. Com esta indicação se distingue uma variável simples de uma variável composta.

O Algoritmo 10-1 ilustra o uso de uma variável composta, denominada **valor**, usada para armazenar 20 valores inteiros distintos. Na linha 7 é feita a declaração de duas variáveis do tipo **inteiro**: uma variável simples, **i**, e outra composta homogênea, **valor**, com 20 posições (ilustrado na Figura 10-1).

Enquanto o identificador **valor** é usado para a variável completa (com todos seus 20 valores), cada inteiro individual é identificado por seu índice, do zero ao 19. A indicação do índice também utiliza os colchetes logo após o identificador. Deste modo, **valor[0]** é o primeiro valor do conjunto, seguido por **valor[1]**, por **valor[2]** e terminando em **valor[19]**. Como o

[*] Outra nomenclatura bastante comum para arranjos é o termo inglês *array*.

algoritmo utiliza as 20 posições, é usada uma variável auxiliar (**i**) para determinar qual a posição em uso. Assim, na linha 12, em cada execução dentro do comando **para**, uma posição diferente do arranjo é utilizada.

Como as posições de um arranjo são independentes, a linha 20, ao utilizar índices diferentes, revisita cada valor lido individualmente, comparando-o em relação à média calculada.

Algoritmo 10-1

```
1  { calcular, para um conjunto de 20 valores inteiros, quantos destes
2    são maiores ou iguais à média do conjunto }
3
4  algoritmo
5        { declaração de variáveis }
6        Declare
7             soma, contador, i,
8             valor[20]: inteiro { i: um inteiro; valor: 20 inteiros }
9             média: real
10
11       { obtenção dos dados e cálculo da média }
12       soma ← 0
13       para i ← 0 até 19 faça
14            leia(valor[i])
15            soma ← soma + valor[i]
16       fim-para
17       média ← soma/20.0
18
19       { contagem de quantos valores são maiores ou iguais à média }
20       contador ← 0
21       para i ← 0 até 19 faça
22            se valor[i] ≥ média então
23                 contador ← contador + 1
24            fim-se
25       fim-para
26
27       { resultado }
28       escreva("Há", contador, "iten(s) maiore(s) que a média", média)
29 fim-algoritmo
```

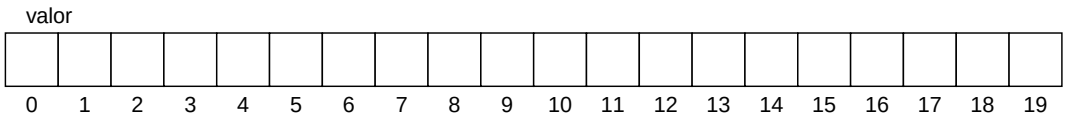


Figura 10-1. Ilustração do arranjo valor **do Algoritmo 10-1.**

Uma declaração de um arranjo é feita para um tamanho fixo de elementos. Em outras palavras, já na declaração se deve conhecer quantas posições existirão e este tamanho não poderá ser modificado. Embora muitas vezes desejável, não se podem usar variáveis para especificar o

número de posições que um arranjo terá, pois isso daria um caráter dinâmico em relação ao tamanho da variável.

Como qualquer outra variável, um arranjo, logo ao ser criado, contém em suas várias posições valores indefinidos ("lixo"). À semelhança das outras variáveis, também só faz sentido usar o conteúdo armazenado se já houver sido feita uma leitura ou uma atribuição àquela posição.

Para exemplificar, considere-se uma declaração de dados como a abaixo.

```
declare lista[5]: inteiro
```

O trecho de algoritmo seguinte ilustra o preenchimento do arranjo com valores, obedecendo uma regra, que é a de que cada posição contenha o dobro do valor da posição imediatamente anterior, acrescido de 1 unidade.

```
lista[0] ← 1              { primeiro da lista }
para i ← 1 até 4          { demais posições }
     lista[i] ← lista[i – 1] * 2 + 1
fim-para
```

A Tabela 10-I apresenta a situação do arranjo **lista** nos vários momentos em que um de seus valores internos é modificado. A primeira coluna da tabela mostra o conteúdo de cada uma das cinco posições existentes no arranjo, sendo a de índice zero indicada do lado esquerdo.

Tabela 10-I. Ilustração passo a passo dos valores de um arranjo para atribuições sucessivas.

conteúdo					Comentário
?	?	?	?	?	*Conteúdo da variável logo após sua declaração. A interrogação indica que o valor é desconhecido.*
1	?	?	?	?	*Valores após a primeira atribuição, anterior ao **para**.*
1	3	?	?	?	*Valores após a primeira atribuição dentro da repetição do comando **para**, quando **i** é igual a 1.*
1	3	7	?	?	*Valores após a atribuição, quando **i** é igual a 2.*
1	3	7	15	?	*Valores após a atribuição, quando **i** é igual a 3.*
1	3	7	15	31	*Valores após a última atribuição do **para**, quando **i** é igual a 4.*

Um comentário final é que os arranjos unidimensionais também são freqüentemente chamados de **vetores**, em uma associação com os vetores utilizados em Geometria Analítica. O termo vetor é usado mesmo quando os valores armazenados não são reais, sendo comum encontrar "um vetor de inteiros" ou "um vetor de valores lógicos".

10.3.2 Aplicações práticas de arranjos unidimensionais

Vetores são de uso amplo para a proposta de solução de problemas. Neste tópico são apresentados alguns exemplos que ilustram sua utilidade em aplicações e como sua manipulação pode ser feita.

Uma primeira proposta: "Escreva um algoritmo para listar o melhor aluno em uma turma com 40 estudantes. Estão disponíveis as notas de três provas, todas com o mesmo peso, além do nome de cada aluno.".

Para sua solução, observa-se que é necessário calcular a média para cada aluno e selecionar a maior nota, além do nome do aluno. Porém, havendo dois alunos com a mesma média, apenas um seria listado. Como pode haver empate, uma proposta seria armazenar todos os nomes e todas as médias calculadas e, após se obter a melhor média, listar todos os alunos que a possuem. O Algoritmo 10-2 ilustra esta opção de resolução.

Algoritmo 10-2

```
1  { listar o(s) melhor(es) aluno(s) de uma turma com 40 alunos, dadas
2    as notas de três provas e sabendo-se que todas possuem o mesmo peso
3    no cálculo da média }
4
5  algoritmo
6        { declarações }
7        constante númeroAlunos: inteiro = 40
8        declare
9             i: inteiro
10            notaProva1, notaProva2, notaProva3, melhorMédia: real
11            nomeAluno[númeroAlunos]: literal
12            média[númeroAlunos]: real
13
14       { obtenção dos dados, cálculo das médias e seleção da melhor }
15       melhorMédia ← -1  { força substituição logo para o primeiro aluno }
16       para i ← 0 até númeroAlunos - 1 faça
17            leia(nomeAluno[i], notaProva1, notaProva2, notaProva3)
18            média[i] ← (notaProva1 + notaProva2 + notaProva3)/3
19
20            se média[i] > melhorMédia então
21                 melhorMédia ← média[i]
22            fim-se
23       fim-para
24
25       { apresentação dos resultados }
26       escreva("Melhor nota final:", melhorMédia)
27       para i ← 0 até númeroAlunos - 1 faça
28            se média[i] = melhorMédia então
29                 escreva("Nome do aluno:", nomeAluno[i])
30            fim-se
31       fim-para
32 fim-algoritmo
```

Neste algoritmo de solução, deve-se observar que há uma correspondência entre as posições do arranjo de nomes, **nomeAluno**, e do arranjo de médias, **média**. Para um dado índice *k*, **nomeAluno[*k*]** obteve a média de provas **média[*k*]**. Outro ponto de destaque é o uso da declaração de constante na linha 7, que ajuda a organizar o algoritmo. Caso a proposta do número de alunos se modifique de 40 para 100, esta é a única linha que deve ser modificada para adequar o algoritmo (além de alterar o comentário da documentação, é claro).

Uma outra proposta de problema: "Escreva um algoritmo para listar o melhor aluno em uma turma de estudantes. Estão disponíveis o número total de alunos, as notas de três provas de cada um, todas com o mesmo peso, além dos nomes de cada um. Deve-se considerar que haja, no máximo, 150 alunos em uma turma.".

Com a modificação da proposta, a solução, embora siga uma linha semelhante, requer algumas modificações. A primeira é a questão de que o número de estudantes é variável. Como não é possível criar arranjos de tamanhos variáveis, vale-se da informação de que o número máximo é 150 alunos. A solução, assim, pode criar um arranjo de 150 posições para nomes e para médias, mas somente o número de posições necessárias são utilizadas (o que corresponde ao número de alunos da turma). As demais posições existem, mas são simplesmente ignoradas durante a execução.

Algoritmo 10-3

```
1  { listar o(s) melhor(es) aluno(s) de uma turma de alunos, dadas
2    as notas de três provas (sabendo-se que todas possuem o mesmo peso
3    no cálculo da média) e tendo-se o número total de alunos }
4
5  algoritmo
6        { declarações }
7        constante máximoAlunos: inteiro = 150
8        declare
9             i: inteiro
10            notaProva1, notaProva2, notaProva3, melhorMédia: real
11            nomeAluno[máximoAlunos]: literal
12            média[máximoAlunos]: real
13
14       { obtenção do número de alunos }
15       leia(númeroAlunos)
16
17       se númeroAlunos > máximoAlunos então
18            escreva("Número máximo de alunos é", máximoAlunos)
19       senão
20            { obtenção dos dados, cálculo das médias e seleção da melhor }
21            melhorMédia ← -1  { força substituição }
22            para i ← 0 até númeroAlunos - 1 faça
23                 leia(nomeAluno[i], notaProva1, notaProva2, notaProva3)
24                 média[i] ← (notaProva1 + notaProva2 + notaProva3)/3
```

```
25
26                  se média[i] > melhorMédia então
27                       melhorMédia ← média[i]
28                  fim-se
29             fim-para
30        fim-se
31
32        { apresentação dos resultados }
33        escreva("Melhor nota final:", melhorMédia)
34        para i ← 0 até númeroAlunos - 1 faça
35             se média[i] = melhorMédia então
36                  escreva("Nome do aluno:", nomeAluno[i])
37             fim-se
38        fim-para
39 fim-algoritmo
```

A solução apresentada no Algoritmo 10-3 incorpora uma técnica comum no uso de arranjos. É feito o dimensionamento para um valor máximo considerado razoável para a solução (o número máximo de alunos, no exemplo acima) e o uso é feito somente para o necessário. Claramente há a introdução de um desperdício de memória, pois há a reserva de espaço para vários dados sem que esta área seja efetivamente usada. É um preço a se pagar, entretanto. A prática no desenvolvimento e as particularidades do problema que está sendo tratado ajudam em estimativas que podem ser consideradas razoáveis para os tamanhos de arranjos que podem ser alocados[*].

De modo a ilustrar outra situação diferenciada, um novo problema é proposto: "Um professor de matemática deseja exemplificar para seus alunos o conceito de união de conjuntos. Para isso, quer que seja desenvolvido um programa simples de computador que permita que os alunos entrem os dados de dois conjuntos (valores reais) e visualizem o resultado da união. Escreva um algoritmo para a implementação de tal programa.".

O desenvolvimento desta solução passa por lembrar que conjuntos podem possuir de zero a infinitos elementos e que o caso do "infinito" seria um impedimento para o programa. Uma segunda conversa com o professor de matemática traz a luz de que conjuntos com até 50 elementos são mais que satisfatórios para seu trabalho com os alunos.

O algoritmo precisa usar dois conjuntos e gerar um terceiro. Como cada conjunto precisa guardar até 50 elementos e também deve saber quantos destes são efetivamente usados, opta-se por usar um registro para estruturar cada conjunto. Este registro mantém um inteiro para guardar o número de elementos no conjunto (de zero a 50, no caso) e um arranjo de reais, que armazenam os valores em si.

[*] Quando um algoritmo for implementado em uma linguagem de programação, a memória real do equipamento e as limitações do sistema operacional também serão fatores determinantes para os tamanhos de arranjos.

Esta solução algorítmica parece ter muitos detalhes e, assim, opta-se por organizar a solução usando sub-rotinas. Outro fato que justifica sub-rotinas é a necessidade de repetição de código, como é o caso da leitura dos conjuntos, que deve ser feita para os dois conjuntos iniciais.

A solução final para o problema da união de conjuntos é apresentada no Algoritmo 10-4.

Algoritmo 10-4

```
1  { exemplificação de teoria de conjuntos para a operação de união,
2    usando a leitura de dois conjuntos de valores reais e mostrando o
3    resultado da união }
4
5  constante máximoElementos: inteiro = 50  { número máximo de elementos
6                                                 no conjunto }
7
8        tipo tConjunto: registro
9                            númeroElementos: inteiro
10                           elementos[máximoElementos]: real
11                       fim-registro
12
13 procedimento insereConjunto(elemento: real, var conjunto: tConjunto,
14                                var sucesso: lógico)
15 { insere um novo elemento no conjunto, retornando sucesso=verdadeiro
16   se a inserção ocorrer; retorna falso se não houver espaço ou se o
17   elemento for repetido }
18       declare
19            i: inteiro
20            achou: lógico
21
22       { ajusta o resultado padrão }
23       sucesso ← falso
24
25       se conjunto.númeroElementos < máximoElementos então
26            { verifica se o elemento já está no conjunto }
27            i ← 0
28            achou ← falso
29            enquanto não achou e i < conjunto.númeroElementos faça
30                 achou ← elemento = conjunto.elemento[i]
31            fim-se
32
33            se não achou então
34                 { insere o novo elemento no conjunto }
35                 conjunto.elemento[conjunto.númeroElementos] ← elemento
36                 conjunto.númeroElementos ← conjunto.númeroElementos + 1
37
38                 sucesso ← verdadeiro
39            fim-se
40       fim-se
41 fim-procedimento
```

```
42
43 procedimento leiaConjunto(var conjunto: tConjunto)
44 { leitura dos elementos de um conjunto, evitando as repetições }
45       declare
46            númeroElementos: inteiro
47            elemento: real
48            inserçãoOk: lógico
49
50       { criação de um conjunto inicialmente vazio }
51       conjunto.númeroElementos ← 0
52
53       { obtenção do número de elementos do conjunto (com limitação) }
54       faça
55            leia(númeroElementos)
56            se númeroElementos > máximoElementos então
57                 escreva("O limite é de zero a ", máximoElementos)
58            fim-se
59       enquanto númeroElementos < 0 ou númeroElementos > máximoElementos
60
61       { entrada de cada elemento }
62       enquanto conjunto.númeroElementos < númeroElementos faça
63            leia(elemento)
64
65            insereConjunto(elemento, conjunto, inserçãoOk)
66            se não inserçãoOk então
67                 escreva("O elemento já existe; digite outro.")
68            fim-se
69       fim-enquanto
70 fim-procedimento
71
72 procedimento escrevaConjunto(conjunto: tConjunto)
73 { escrita dos elementos de um conjunto }
74       declare i: inteiro
75
76       se conjunto.númeroElementos = 0 então
77            escreva ("Conjunto vazio")
78       senão
79            para i ← 0 até conjunto.númeroElementos - 1 faça
80                 escreva(conjunto.elemento[i])
81            fim-para
82       fim-se
83 fim-procedimento
84
85 procedimento unaConjuntos(var resultado: tConjunto,
86                           conjunto1, conjunto2: tConjunto)
87 { cria o conjunto resultado a partir da união dos conjuntos 1 e 2 }
88       declare
89            i: inteiro
90            inserçãoOk: lógico
91
```

```
92        { cria resultado inicialmente igual ao primeiro conjunto }
93        resultado ← conjunto1
94
95        { insere os elementos do conjunto 2 no resultado, sem repetições }
96        inserçãoOk ← verdadeiro
97        i ← 0
98        enquanto i < conjunto2.númeroElementos e inserçãoOk faça
99             insereConjunto(conjunto2.elemento[i], resultado, inserçãoOk)
100            i ← i + 1
101       fim-para
102
103       se não inserçãoOk então
104            escreva("Não houve espaço para a união; o resultado",
105                    "não contém todos os elementos")
106       fim-se
107 fim-procedimento
108
109 { parte principal }
110 algoritmo
111       declare conjunto1, conjunto2, conjuntoUnião: tConjunto
112
113       { entrada de dados }
114       leiaConjunto(conjunto1)
115       leiaConjunto(conjunto2)
116
117       { união }
118       unaConjuntos(conjuntoUnião, conjunto1, conjunto2)
119
120       { resultado }
121       escreva("Resultado da união de")
122       escrevaConjunto(conjunto1)
123       escreva("e")
124       escrevaConjunto(conjunto2)
125       escreva("contém os elementos")
126       escrevaConjunto(conjuntoUnião)
127 fim-algoritmo
```

Seguem alguns comentários pertinentes sobre o Algoritmo 10-4. Na parte principal é feita a leitura de cada um dos conjuntos, seguida da união e da apresentação dos resultados (que inclui ambos os conjuntos e o conjunto resultante da união).

O procedimento **leiaConjunto** pega um registro do tipo **tConjunto** passado por referência. Faz primeiro a leitura do número de elementos que serão digitados, garantindo que estejam no intervalo [0, **máximoElementos**]. O conjunto é ajustado para que seja vazio (ou seja, com número de elementos igual a zero) e cada elemento lido é inserido no conjunto, atualizando o campo **númeroElementos**, que controla a quantidade de elementos válidos. Um procedimento específico para colocar um novo elemento no conjunto (**insereConjunto**) tem a

função de acrescentar um item ao arranjo, desde que haja espaço e que o elemento ainda não exista. O sucesso ou não da inserção é indicado por um parâmetro de saída (**sucesso**).

A sub-rotina **escrevaConjunto** faz simplesmente a apresentação dos dados do conjunto.

Para que a união seja realizada, é feita uma cópia de um dos conjuntos e, a esta cópia, são acrescentados, um a um, os elementos do segundo conjunto. O procedimento **insereConjunto** é usado novamente, pois já verifica se há repetições, além de checar por espaço disponível.

Em relação a aplicações de arranjos, o próprio Algoritmo 10-4 ilustra algumas delas. Podem ser citadas:

- Utilização de registros como um dos campos de registro, criando uma estrutura mais autônoma e completa, já que representa, com seus vários campos, uma única entidade.
- Uso de arranjo como repositório de elementos, incluindo um procedimento de inclusão de um novo elemento.
- Pesquisa por um elemento em um arranjo, verificando sua existência.

Outros tipos de uso incluem usar o arranjo para manter os dados em ordem crescente, possibilitar a remoção de um elemento do arranjo (com consecutivo decremento no contador de número de elementos).

O Algoritmo 10-5 apresenta outra versão para o procedimento **insereConjunto** do Algoritmo 10-4, o qual mantém os elementos do conjunto organizado de forma crescente. Já o Algoritmo 10-6 ilustra um procedimento para, dado um elemento, removê-lo do conjunto.

Algoritmo 10-5

```
1  procedimento insereConjunto(elemento: real, var conjunto: tConjunto,
2                                  var sucesso: lógico)
3  { insere novo elemento no arranjo, mantendo ordem crescente nos
4    elementos; sucesso retorna falso se não há espaço ou se há repetição,
5    e verdadeiro caso contrário }
6        declare
7             i, posição: inteiro
8
9        { valor padrão para sucesso }
10       sucesso ← falso
11
12       se conjunto.númeroElementos < máximoElementos então
13            { busca se já existe }
14            posição ← 0
15            enquanto posição < conjunto.númeroElementos e
16                      conjunto.elementos[posição] < elemento faça
17                 posição ← posição + 1
18            fim-enquanto
19
20            se posição ≥ conjunto.númeroElementos ou
```

```
21              conjunto.elementos[posição] ≠ elemento então
22                { faz o deslocamento dos itens maiores que o novo }
23                i ← conjunto.númeroElementos – 1
24                enquanto i ≥ posição faça
25                     conjunto.elementos[i + 1] ← conjunto.elementos[i]
26                     i ← i - 1
27                fim-enquanto
28
29                { insere no arranjo }
30                conjunto.elementos[posição] ← elemento
31                conjunto.númeroElementos ← conjunto.númeroElementos + 1
32
33                sucesso ← verdadeiro
34           fim-se
35      fim-se
36 fim-algoritmo
```

Algoritmo 10-6

```
1  procedimento removerConjunto(elemento: real, var conjunto: tConjunto,
2                                    var sucesso: lógico)
3  { busca e remove um dado elemento do conjunto retornando sucesso como
4    verdadeiro; se o item não existir, sucesso retorna falso }
5       declare i: inteiro
6
7       { localiza o elemento }
8       i ← 0
9       enquanto i < conjunto.númeroElementos e
10                elemento ≠ conjunto.elementos[i] faça
11           i ← i + 1
12      fim-enquanto
13
14      { verifica presença ou ausência do elemento }
15      se i ≥ conjunto.númeroElementos ou
16         elemento = conjunto.elementos[i] então
17           sucesso ← falso
18      senão
19           { remove o item }
20           enquanto i < conjunto.númeroElementos e
21                     i < máximoElementos - 1 faça
22                conjunto.elementos[i] ← conjunto.elementos[i + 1]
23                i ← i + 1
24           fim-enquanto
25           conjunto.númeroElementos ← conjunto.númeroElementos - 1
26
27           sucesso ← verdadeiro
28      fim-se
29 fim-procedimento
```

10.3.3 Declaração e uso de arranjos bidimensionais

Uma estrutura bastante comum em matemática são as matrizes. Uma matriz é um arranjo de duas dimensões (linhas e colunas) de elementos, todos do mesmo tipo. Em algoritmos é possível usar **arranjos bidimensionais**, também conhecidos genericamente como **matrizes** (independentemente do tipo de dado dos elementos).

A declaração de arranjos bidimensionais é feita com a especificação de mais uma dimensão, o que é representado por mais um valor incluído entre colchetes. Assim, um novo par de colchetes deve ser acrescentado. São apresentados, abaixo, alguns exemplos de matrizes.

```
        declare
             idade[10][20]: inteiro   { matriz 10 linhas por 20 colunas }
             notas[50][3]: real       { matriz 50 x 3 }
             nomes[2][5]: literal     { matriz 2 x 5 }
```

Como padrão, adota-se a primeira dimensão para simbolizar as linhas e a segunda para as colunas. Deste modo, **idade[2][5]** é o valor armazenado na terceira linha, sexta coluna. Novamente, os índices se iniciam em zero, terminando no valor da dimensão menos 1. No exemplo acima, a matriz **notas** possui 150 elementos reais, com índices variando de **notas[0][0]** a **notas[49][2]**.

10.3.4 Aplicações práticas de arranjos bidimensionais

O exemplo inicial de aplicação é a representação de matrizes de números reais, usadas em matemática. No Algoritmo 10-7 é apresentada uma solução para exemplificar o uso das matrizes, com os procedimentos básicos de leitura e escrita e algumas funções (traço e verificação se a matriz é quadrada). Na parte principal é lido um grupo de matrizes diferentes e alguns dados são apresentados.

Algoritmo 10-7

```
1  { exemplo de utilização de algumas operações sobre matrizes matemáticas }
2
3  constante
4        máximoLinhas: inteiro = 100
5        máximoColunas: inteiro = 100
6
7  tipo tMatriz: registro
8                       númeroLinhas, númeroColunas: inteiro
9                       dados[máximoLinhas][máximoColunas]: real
10              fim-registro
11
12 procedimento leiaMatriz(var matriz: tMatriz)
13 { obtenção dos dados da matriz, incluindo as dimensões }
14       declare i, j: inteiro
15
16       { obtenção das dimensões }
```

```
17       faça
18            leia(matriz.númeroLinhas, matriz, númeroColunas)
19       enquanto matriz.númeroLinhas < 0 ou
20                 matriz.númeroLinhas > máximoLinhas ou
21                 matriz.númeroColunas < 0 ou
22                 matriz.númeroColunas > máximoColunas
23
24       { leiura dos dados }
25       para i ← 0 até matriz.númeroLinhas – 1 faça
26            para j ← 0 até matriz.númeroColunas – 1 faça
27                 leia(matriz.dados[i][j])
28            fim-para
29       fim-para
30  fim-procedimento
31
32  procedimento escrevaMatriz(matriz: tMatriz)
33  { escrita dos dados armazenados na matriz }
34       declare i, j: inteiro
35
36       { saída }
37       se matriz.númeroLinhas ≤ 0 ou matriz.númeroColunas ≤ 0 então
38            escreva("Matriz inválida")
39       senão
40            { apresentação da matriz }
41            para i ← 0 até matriz.númeroLinhas – 1 faça
42                 para j ← 0 até matriz.númeroColunas – 1 faça
43                      escreva(matriz.dados[i][j])
44                 fim-para
45            fim-para
46       fim-se
47  fim-procedimento
48
49  função éQuadrada(matriz: tMatriz): lógico
50  { verifica se uma matriz é quadrada }
51
52       retorne matriz.NúmeroLinhas = matriz.númeroColunas
53  fim-função
54
55  função traçoMatriz(matriz: tMatriz): real
56  { calcula o traço da matriz; se não for quadrada, retorna 0 }
57       declare
58            i: inteiro
59            soma: real
60
61       { soma os elementos da diagonal principal }
62       soma ← 0
63       se éQuadrada(matriz) então
64            para i ← 0 até matriz.númeroLinhas faça
65                 soma ← soma + matriz.dados[i][i]
66            fim-para
```

```
67       fim-se
68
69       { resultado }
70       retorne soma
71  fim-função
72
73  procedimento multipliqueMatriz(var resultado: tMatriz,
74                                 matriz1, matriz2: tMatriz)
75  { calcula a multiplicação de duas matrizes; se as dimensões não forem
76    compatíveis, retorna uma matriz inválida (0 x 0) }
77       declare
78
79       { verificação das dimensões }
80       se matriz1.númeroColunas ≠ matriz1.númeroLinhas então
81            { não é possível multiplicar }
82            resultado.númeroLinhas ← 0
83            resultado.númeroColunas ← 0
84       senão
85            { calcula a multiplicação }
86            resultado.númeroLinhas ← matriz1.númeroLinhas
87            resultado.númeroColunas ← matriz2.númeroColunas
88            para i ← 0 até resultado.númeroLinhas - 1 faça
89                 para j ← 0 até resultado.númeroColunas - 1 faça
90                      resultado.dados[i][j] ← 0  { usa como acumulador }
91                      para k ← 0 até matriz1.númeroColunas - 1 faça
92                           resultado.dados[i][j] ← resultado.dados[i][j] +
93                                matriz1.dados[i][k] * matriz2.dados[k][j]
94                      fim-para
95                 fim-para
96            fim-para
97       fim-se
98  fim-procedimento
99
100 função temDiagonalZero(matriz: tMatriz): lógico
101 { retorna verdadeiro se todos os elementos da diagonal principal
102   forem nulos; se não forem nulos ou a matriz não for quadrada,
103   retorna falso }
104      declare
105           tudoCerto: lógico
106           i: inteiro
107
108      { verificação }
109      tudoCerto ← éQuadrada(matriz)
110      i ← 0
111      enquanto i < matriz.númeroLinhas e tudoCerto faça
112           tudoCerto ← matriz.dados[i][i] = 0
113           i ← i + 1
114      fim-enquanto
115
116      retorne tudoCerto
```

```
fim-função

algoritmo
     declare
          númeroLeituras: inteiro
          traço, maiorTraço: real
          matriz, matrizSelecionada: tMatriz

     { iniciação }
     maiorTraço ← -1E20       { "quase" menos infinito }

     { leitura dos dados }
     leia(númeroLeituras)
     para i ← 1 até númeroLeituras faça
          { leitura dos dados da matriz }
          leiaMatriz(matriz)

          { seleção da matriz de maior traço }
          traço ← traçoMatriz(matriz)
          se traço > maiorTraço então
               maiorTraço ← traço
               matrizSelecionada ← matriz
          fim-se

          { alguns dados sobre a matriz digitada }
          escreva("O traço é", traço)
          se não éQuadrada(matriz) então
               escreva("A matriz não é quadrada")
          senão
               escreva("A matriz é quadrada")
               se temDiagonalZero(matriz) então
                    escreva("Todos os valores da diagonal são zero")
               senão
                    escreva("Algum valor na diagonal não é nulo")
               fim-se
          fim-se
     fim-para

     { mais uma informação }
     { obs.: assume-se que nenhum traço resulte em 10^-20 }
     se maiorTraço = -1E20 então
          escreva("Nenhuma matriz digitada foi quadrada")
     senão
          escreva("O maior traço entre as matrizes foi", maiorTraço)
          escreva("Matriz de maior traço:")
          escrevaMatriz(matrizSelecionada)
     fim-se
fim-algoritmo
```

Além das matrizes de números reais, são úteis também matrizes com outros tipos, pois podem representar adequadamente uma informação importante para que o algoritmo resolva um determinado problema. Por exemplo, uma matriz de valores lógicos pode representar um tabuleiro do jogo de *Batalha Naval*, sendo que **falso** indica *água* e **verdadeiro** representa uma parte de uma embarcação em uma determinada coordenada.

10.3.5 Arranjos multidimensionais

A extensão da dimensionalidade de um arranjo é direta. Acrescentando-se mais dimensões, ou seja, especificando mais índices, podem ser declarados arranjos de três dimensões ou mais. Usá-los pode ou não ser adequado para representar algumas informações.

10.4 Considerações finais

Os arranjos, de qualquer dimensão, são ferramentas essenciais para o desenvolvimento de algoritmos. Muitos dados têm que existir durante grande parte do algoritmo e, portanto, devem ter todos seus valores preservados para consulta em diversos momentos distintos. É possível calcular a soma de duas matrizes de reais (soma elemento a elemento) sem usar arranjos bidimensionais. Para isso, basta que sejam lidos os elementos das matrizes intercaladamente, já apresentando o resultado (lêem-se os elementos a_{00} e o b_{00}, calculando-se o resultado c_{00}; então são lidos a_{01} e o b_{01}, calculando-se c_{01}, e assim sucessivamente). Não é, porém, uma forma nem elegante nem prática de resolver o problema. Mas é uma solução possível. O cálculo da multiplicação das matrizes (codificada no Algoritmo 10-7), por sua vez, é impossível de ser feito sem que ambas as matrizes estejam disponíveis em vários momentos, pois os dados são consultados diversas vezes, sendo preciso estarem à mão.

Um hábito que muitos desenvolvedores de código adquirem, porém, é o de usar arranjos quase que indiscriminadamente, declarando vetores e matrizes mesmo quando não são necessários. Como um exemplo desta situação, pense no problema básico de calcular a média de um conjunto de temperaturas, para o qual a solução pode ler um vetor de temperaturas e varrê-lo para calcular a média dos valores. Nesta situação, cada valor de temperatura é necessário uma única vez e, assim, bastaria que uma única variável (e não um arranjo inteiro) fosse empregada.

É preciso sempre lembrar que arranjos tendem a consumir um recurso importante, que é a memória. Uma matriz 10x10, por exemplo, possui 100 elementos; uma 100x100 já passa a armazenar 10.000 itens. Uma matriz de três dimensões 100x100x100, por sua vez, define 1.000.000. Quando o algoritmo for implementado em uma linguagem de programação em um computador real, estas limitações passam a ser críticas. Como um caso prático, pode-se considerar que um número real simples, em uma linguagem de programação, utiliza 4 bytes. Isso leva uma matriz 100x100 a consumir 40.000 bytes. Para realizar uma multiplicação de matrizes usando o procedimento **multipliqueMatriz** do Algoritmo 10-7, a parte principal do algoritmo deve usar três matrizes (duas para leitura dos

dados e uma para armazenamento do resultado); ao ser feita a chamada da sub-rotina, duas matrizes são criadas (a passagem é por valor e é feita uma cópia dos dados), mas o resultado não requer outra matriz (a área de memória é compartilhada, pois é chamada por referência). No saldo final, são necessárias cinco matrizes, o que dá um total de 200.000 bytes somente para armazenar os elementos das diversas matrizes. Um exercício para o leitor é refazer os cálculos para um arranjo tridimensional 100x100x100.

Para resumir, é possível sempre lançar mão de um recurso importante como arranjos, mas nunca se deve deixar de verificar sua real necessidade e de se ter noção que o consumo de memória pode ser muito elevado. Use com moderação.

Unidade 11 - Estruturas compostas mistas: homogêneas e heterogêneas

11.1 Primeiras palavras

A organização de dados em agrupamentos é feita de dois modos em algoritmos: na forma de registros (heterogêneos) e na forma de arranjos (homogêneos).

Em muitas situações um desenvolvedor cria uma matriz para guardar dados reais e, em um determinado momento, pesa algo como "bem que esta última coluna poderia ser de valores lógicos, e não de reais". Para estas (e outras situações) um arranjo de dados homogêneos não resolve o problema, mas é viável pensar em uma estrutura heterogênea.

Deste modo, passa a ser interessante a utilização de um arranjo de registros, que é o tema desta última Unidade da disciplina.

11.2 Listas mais sofisticadas

A utilização de conjuntos de dados que precisam ser mantidos em memória durante parte do tempo de execução de um algoritmo é uma necessidade real.

Pode-se considerar o seguinte problema: "Uma adega mantém dados sobre vinhos, que incluem nome do produto, nome do fabricante, nome do distribuidor, tipo de vinho (tinto, branco ou *rosé*), ano da safra, ano do envasamento, preço da garrafa e número de unidades vendidas em um dado período. Deseja-se, a partir destes dados, determinar o vinho de maior saída em valor (número de unidades multiplicada pelo valor) e a média, em valor, da saída geral de vinhos (média dos produtos de quantidade por valor unitário). Precisa-se, também, ao final, dos dados de todos os vinhos que estejam abaixo desta média, para que a gerência decida sobre a continuidade das compras.".

É possível, na proposição de uma solução algorítmica, criar um vetor para cada informação (um vetor de nomes, um vetor de produtores, um vetor de preços e assim por diante). A partir destes vetores, calcular o vinho de maior saída, calcular a média e listar os dados desejados dos vinhos "abaixo da média".

Uma solução mais elegante (não necessariamente computacional) é ter todos os dados tabulados, fazer os cálculos necessários e gerar os dados. Para o algoritmo, o conceito reside em ter um único vetor de dados, sendo que cada posição corresponde a um vinho, ou seja, a um registro no qual todos os dados daquele vinho estejam armazenados.

11.3 Declaração e uso de arranjos de registros

O uso de tipos é a forma mais elegante, bem documentada e clara de usar declarações para **arranjos de registros**.

Como exemplificação para sua utilização, é apresentada uma solução para o problema dos vinhos da seção anterior. Como restrições ao problema

se estabelece que o número máximo de itens na lista seja de 300 tipos diferentes e que se saiba o número de itens existente.

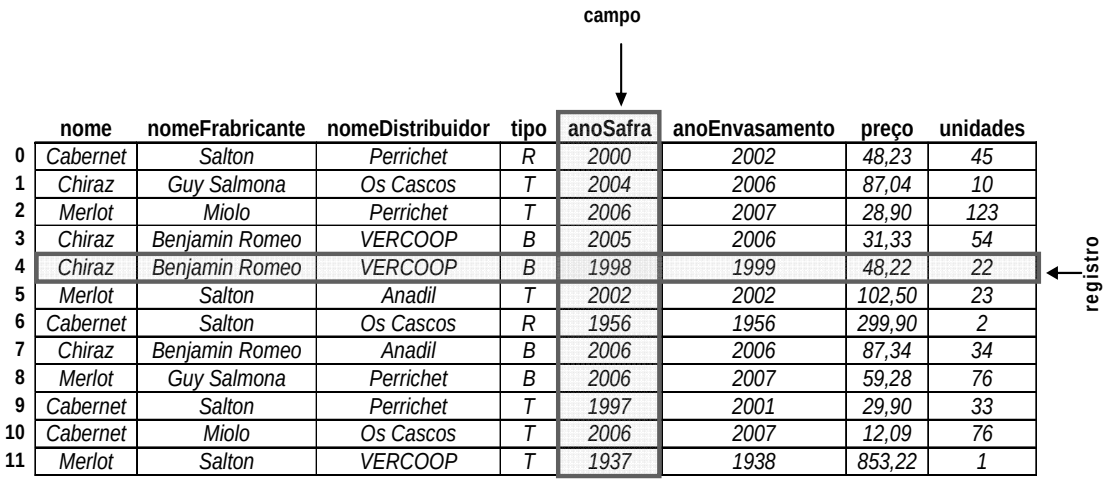
Algoritmo 11-1

```
1  { obter, para uma relação de dados sobre vinhos, o vinho de maior venda
2    e a lista de todos os vinhos que tenham venda abaixo da média geral do
3    grupo }
4  constante máximoElementos: inteiro = 300
5
6  tipo tVinho: registro
7                  nome,
8                  nomeFabricante,
9                  nomeDistribuidor,
10                 tipo: literal
11                 anoSafra,
12                 anoEnvasamento: inteiro
13                 preço: real
14                 unidades: inteiro
15               fim-registro
16       tListaVinhos[máximoElementos]: tVinhos  { arranjo de registros }
17
18 procedimento leiaVinho(var vinho: tVinho)
19 { leitura de um registro de vinho }
20
21       leia(vinho.nome, vinho.nomeFabricante, vinho.nomeDistribuidor,
22            vinho.tipo, vinho.anoSafra, vinho.anoEnvasamento,
23            vinho.preço, vinho.unidades)
24 fim-procedimento
25
26 procedimento escrevaVinho(vinho: tVinho)
27 { leitura de um registro de vinho }
28
29       escreva(vinho.nome, vinho.nomeFabricante, vinho.nomeDistribuidor,
30            vinho.tipo, vinho.anoSafra, vinho.anoEnvasamento,
31            vinho.preço, vinho.unidades)
32 fim-procedimento
33
34 função calculeVendas(vinho: tVinho): real
35 { retorna o produto do preço pelo número de unidades vendidas }
36
37       retorne vinho.unidades * vinho.preço
38 fim-função
39
40 { parte principal do algoritmo }
41 algoritmo
42       declare
43            i, númeroItens: inteiro
44            somaVendas, maiorVenda, médiaVinhos: real
45            vinho: tListaVinhos
46
```

```
47       { leitura da relação de dados e cálculo da média e da maior venda }
48       leia(númeroItens)  { assume-se maior ou igual a 1 }
49       somaVendas ← 0
50       maiorVenda ← -1
51       para i ← 0 até númeroItens – 1 faça
52            leiaVinho(vinho[i])
53
54            somaVendas ← somaVendas + calculeVendas(vinho[i])
55            se calculeVendas(vinho[i]) > maiorVenda então
56                 maiorVenda ← calculeVendas(vinho[i])
57            fim-se
58       fim-para
59       médiaVendas ← somaVendas/númeroItens
60
61       { apresentação da melhor venda }
62       para i ← 0 até númeroItens – 1 faça
63            se calculeVendas(vinho[i]) = maiorVenda então
64                 escrevaVinho(vinho[i])
65            fim-se
66       fim-para
67
68       { apresentação dos vinhos abaixo da média calculada }
69       para i ← 0 até númeroItens – 1 faça
70            se calculeVendas(vinho[i]) < médiaVinhos então
71                 escrevaVinho(vinho[i])
72            fim-se
73       fim-para
74  fim-algoritmo
```

Para esta solução pode-se visualizar o arranjo de registros como sendo uma tabela, na qual cada linha é uma posição do arranjo, organizando-se os campos como colunas. A Figura 11-1 mostra uma representação visual do arranjo (mostrado verticalmente, com suas posições identificadas pelos índices), com os diversos dados de cada campo[*].

[*] Os enólogos devem perdoar os dados ficcionais desta tabela de vinhos.



	nome	nomeFrabricante	nomeDistribuidor	tipo	anoSafra	anoEnvasamento	preço	unidades
0	*Cabernet*	*Salton*	*Perrichet*	*R*	*2000*	*2002*	*48,23*	*45*
1	*Chiraz*	*Guy Salmona*	*Os Cascos*	*T*	*2004*	*2006*	*87,04*	*10*
2	*Merlot*	*Miolo*	*Perrichet*	*T*	*2006*	*2007*	*28,90*	*123*
3	*Chiraz*	*Benjamin Romeo*	*VERCOOP*	*B*	*2005*	*2006*	*31,33*	*54*
4	*Chiraz*	*Benjamin Romeo*	*VERCOOP*	*B*	*1998*	*1999*	*48,22*	*22*
5	*Merlot*	*Salton*	*Anadil*	*T*	*2002*	*2002*	*102,50*	*23*
6	*Cabernet*	*Salton*	*Os Cascos*	*R*	*1956*	*1956*	*299,90*	*2*
7	*Chiraz*	*Benjamin Romeo*	*Anadil*	*B*	*2006*	*2006*	*87,34*	*34*
8	*Merlot*	*Guy Salmona*	*Perrichet*	*B*	*2006*	*2007*	*59,28*	*76*
9	*Cabernet*	*Salton*	*Perrichet*	*T*	*1997*	*2001*	*29,90*	*33*
10	*Cabernet*	*Miolo*	*Os Cascos*	*T*	*2006*	*2007*	*12,09*	*76*
11	*Merlot*	*Salton*	*VERCOOP*	*T*	*1937*	*1938*	*853,22*	*1*

Figura 11-1. Ilustração de um arranjo de registros como uma tabela.

Um ponto importante do uso de arranjos de serviço é o modo de acesso aos dados. O arranjo é uma única variável e corresponde a uma coleção ordenada (e indexada) de elementos do mesmo tipo (como o tipo **tListaVinhos**, no Algoritmo 11-1). Cada um de seus elementos corresponde a um registro do tipo **tVinho**. Assim, ao se declarar a variável **vinho** na linha 45, o identificador **vinho** passa a ser todo o arranjo. A especificação **vinho[i]** (linha 52, por exemplo), corresponde a um único registro e tem o tipo **tVinho**. Como, ainda na linha 52, **vinho[i]** é um registro, pode ser passado como argumento (por referência, no caso) para o procedimento **leiaVinho**, que aceita como parâmetro exatamente um registro do tipo **tVinho**, como se especifica na linha 18. Um cuidado em especial deve ser tomado aqui para distinguir o parâmetro **vinho** (do tipo **tVinho**), que é parte do procedimento **leiaVinho** da variável **vinho**, da parte principal do algoritmo, que é um arranjo de 300 posições do tipo **tListaVinhos**; os escopos são diferentes.

Pode-se, ainda, enfatizar que na parte principal do algoritmo seria válido escrever **vinho[i].nome**, já que **vinho[i]** é um registro e **nome** é um de seus campos. Assim, a linha 70 teria exatamente o mesmo significado se fosse escrita como abaixo, apenas substituindo a chamada da função pelo cálculo que esta efetivamente faz.

```
        se vinho[i].unidades * vinho[i].preço < médiaVinhos então
```

Ainda, é preciso atenção para diferenciar o que seria **vinho[i].nome** de **vinho.nome[i]**. No primeiro caso, **vinho** é um arranjo de registros, **vinho[i]** é o registro de índice **i** e, finalmente, **vinho[i].nome** é o literal contido neste campo para a posição **i** do arranjo. Na segunda especificação, **vinho.nome[i]**, **vinho** tem que ser um registro, para o qual um dos campos é **nome**, sendo **nome** um arranjo, no qual se escolhe a posição **i**. Portanto, nesta segunda situação, existe um arranjo dentro do registro, e não um arranjo de registros, como se está discutindo nesta Unidade.

11.4 Considerações finais

Arranjos de registros formam uma das estruturas de dados que melhor permite organizar dados de certa complexidade em coleções de elementos iguais. Permitem que esta organização traga clareza para o que os dados representam e como deve ser feito o acesso a cada campo.

É claro que, no lugar de um arranjo de registros, seria viável ter vários arranjos separados, um para cada campo. Porém, haveria a introdução de um grau de "desorganização" (ou "sub-organização"), pois dados que constituem unidades seriam representados por variáveis completamente separadas. Além disso, seria inviabilizada a possibilidade de leitura de todos os dados, como é feito no exemplo usado nesta unidade. Seria preciso, para o procedimento de leitura, passar como argumento cada dado separadamente, como na linha seguinte.

```
leia(nome[i], nomeFabricante[i], nomeDistribuidor[i], tipo[i],
     anoSafra[i], anoEnvasamento[i], preço[i], unidades[i])
```

Da mesma forma, cada parâmetro teria que ser passado separadamente para outros procedimentos ou funções, uma vez que não existiria a unidade proporcionada pelo registro.

Recomenda-se fortemente, assim, o emprego de arranjos (ou até matrizes, se for o caso) de registros no lugar de vários arranjos (ou matrizes) separadamente.