



Lógica e Linguagem de Programação

Introdução ao Desenvolvimento de Software

Humberto Martins Beneduzzi

É professor do Instituto Federal do Paraná – IFPR. Graduado em Sistemas de Informação pelo CEFET-PR e especialista em Metodologia do Ensino Superior pelo IBPEX. É desenvolvedor de software desde 1995. Já participou de projetos de desenvolvimento de software em empresas de pequeno, médio e grande portes, utilizando diversas tecnologias e linguagens de programação.

João Ariberto Metz

Técnico em Programação de Computadores pela SPEI e graduado em Análise de Sistemas, pela ESSEI. Trabalhou por mais de 20 anos na função de Programador de Computador no antigo CEFET-PR, hoje Universidade Tecnológica Federal do Paraná.

Direção Geral	Jean Franco Sagrillo
Direção Editorial	Jeanine Grivot
Edição	Leonel Francisco Martins Filho
Gerência de Produção e Arte	Marcia Tomeleri
Revisão	Jeferson Turbay Braga Simone Venske
Revisão Comparativa	Renee Cleyton Faletti
Projeto Gráfico	Adriana de Oliveira
Editoração eletrônica	Fábio Roberto Hancke

Beneduzzi, Humberto Martins.

Lógica e linguagem de programação: introdução ao desenvolvimento de software / Humberto Martins Beneduzzi; João Ariberto Metz – Curitiba: Editora do Livro Técnico, 2010.

144 p.
ISBN: 978-85-63687-11-1

1. Informática. 2. Software. 3. Linguagem de programação.
4. Tecnologia. I. Metz, João Ariberto. II. Título.

CDD 005.13

Eutália Cristina do Nascimento Moreto CRB-9/947

2010

Todos os direitos reservados pela Editora do Livro Técnico
Edifício Comercial Sobral Pinto
Avenida Cândido de Abreu 469, 2º andar, conj. nºs 203-205
Centro Cívico – CEP: 80530-000
Tel.: 41 3027-5952 / Fax: 41 3076-8783
www.editoralt.com.br
Curitiba – PR



Em conformidade com o Catálogo Nacional de Cursos Técnicos, este livro é indicado, entre outros, para os seguintes cursos:

Eixo Tecnológico: Informação e Comunicação

- Técnico em Informática
- Técnico em Informática para a Internet
- Técnico em Manutenção e Suporte em Informática
- Técnico em Programação de Jogos Digitais
- Técnico em Redes de Computadores

Eixo Tecnológico: Ambiente, Saúde e Segurança

- Técnico em Meteorologia

Eixo Tecnológico: Infraestrutura

- Técnico em Geoprocessamento

Eixo Tecnológico: Controle e Processos Industriais

- Técnico em Automação Industrial
- Técnico em Mecatrônica

Apresentação

Este livro foi didaticamente pensado para o aluno de nível técnico. Nele, foram privilegiadas as abordagens mais práticas, baseadas em situações do dia a dia, em detrimento das abordagens excessivamente teóricas e distantes da realidade dos alunos. Entendemos que a abordagem é tão importante para o aprendizado quanto a definição adequada do conteúdo a ser ensinado.

A tentativa de realizar uma abordagem mais leve do assunto, não nos fez esquecer do nível de profundidade com que os conteúdos devem ser abordados. O objetivo é fornecer uma obra didática, de fácil acesso, mas, ao mesmo tempo, completa em termos de diversidade e profundidade do conteúdo.

Por meio deste livro, procuramos contribuir para a formação dos futuros profissionais da área de desenvolvimento de software.

Sumário

CAPÍTULO 1 – Introdução à Lógica de Programação	9
Algoritmos	10
Representação de Algoritmos.....	11
Atividades	15
CAPÍTULO 2 – Variáveis, Tipos de Dados e Constantes	16
Variáveis	16
Tipos de Dados	18
Declaração de Variáveis.....	19
Atribuição e Inicialização de Variáveis	20
Constantes	22
Atividades.....	23
CAPÍTULO 3 – Operadores e Expressões	24
Operadores Aritméticos.....	24
Operadores Relacionais	27
Operadores Lógicos.....	28
Operador Literal.....	32
Teste de Mesa.....	34
Atividades.....	36
CAPÍTULO 4 – Estruturas de Controle	37
Seleção.....	37
Repetição	44
Atividades.....	57
CAPÍTULO 5 – Estruturas de Dados Homogêneas	59
Vetores.....	59
Matrizes.....	64
Atividades.....	69
CAPÍTULO 6 – Estruturas de Dados Homogêneas: Ordenação e Pesquisa	70
Ordenação de Vetores	70
Pesquisa Sequencial	76
Pesquisa Binária	79
Atividades.....	83

CAPÍTULO 7 – Estruturas de Dados Heterogêneas	84
Registros	84
Atividades.....	91
CAPÍTULO 8 – Sub-rotinas	92
Procedimentos.....	93
Funções.....	98
Escopo de Variáveis.....	101
Considerações sobre o Uso de Sub-rotinas.....	105
O Uso de Parâmetros	106
Atividades.....	112
CAPÍTULO 9 – Introdução à Programação	115
Linguagem de Máquina	115
Linguagem de Programação.....	116
Atividades.....	117
CAPÍTULO 10 – Linguagem Pascal	118
Fundamentos	118
Estruturas de Dados.....	127
Sub-rotinas.....	130
Atividades.....	133
Apêndice 1	134
Apêndice 2	135
Referências Bibliográficas	144

Introdução à Lógica de Programação

Os *softwares*, que antigamente só podiam ser executados em computadores, estão cada vez mais presentes nos dispositivos que utilizamos em nosso dia a dia. Celulares, videogames, aparelhos de GPS e, até mesmo, fornos micro-ondas possuem *software* rodando internamente.

A demanda por *software* nunca foi tão grande quanto hoje e tende a crescer cada vez mais. Por isso, o desenvolvimento de *software* é uma das áreas que mais ganhou espaço no mercado, nos últimos tempos.

A quantidade de mão de obra especializada é consideravelmente menor do que a demanda, e muitas empresas trabalham para formar seus próprios profissionais por causa da dificuldade de encontrá-los no mercado.

Estes são alguns dos motivos que têm levado milhares de pessoas a se interessarem pelo campo de desenvolvimento de *software*. E a principal procura é por aprender a programar.

Neste contexto, nota-se que muitas vezes a urgência em se estar pronto para o mercado ou, então, em “começar a produzir”, leva o estudante a tentar partir diretamente para o aprendizado de uma linguagem de programação. Obviamente, é possível aprender e se tornar um bom profissional partindo deste caminho. Mas talvez esta não seja a forma mais indicada.

A falta de uma base conceitual acaba, por diversas vezes, dificultando o trabalho de muitos desenvolvedores e, em alguns casos, até comprometendo a qualidade de seu trabalho.

Por isso, o aprendizado da lógica de programação é de extrema importância para aqueles que desejam se tornar profissionais da área de desenvolvimento de *software*. A lógica de programação serve como base para o aprendizado de qualquer linguagem de programação, pois os conceitos estudados se aplicam à grande maioria das linguagens.

No nosso dia a dia, nos deparamos com uma série de situações que nos levam a utilizar a lógica. Estas situações podem ser consideradas pequenos problemas, que serão solucionados com maior ou menor eficiência e eficácia, dependendo do uso que fizermos da lógica quando elaborarmos suas soluções. Por exemplo, escolher o trajeto a ser feito para ir de um local A até um local B, ou a maneira com que deveremos dispor os materiais para que caibam na mochila, são questões que nos convidam a utilizar a lógica em sua resolução.

Agora vamos transportar esta abordagem para os programas de computador. Os *softwares* também se propõem a resolver problemas e, para tal, dependem da lógica que o programador elaborou quando os desenvolveu.

Possuindo conhecimentos de lógica de programação o profissional poderá oferecer soluções melhores na resolução de problemas computacionais. Deste modo, a lógica torna-se indispensável na formação de um bom programador, pois os conhecimentos adquiridos nesta disciplina serão utilizados para realizar o trabalho, independentemente da tecnologia ou linguagem de programação escolhida.

Algoritmos

Um algoritmo é um conjunto de instruções, dispostas em uma sequência lógica, que levam à resolução de um problema. Em outras palavras, um algoritmo é uma espécie de passo a passo para se chegar à solução de um problema.

Por exemplo, este simples algoritmo descreve a sequência de passos necessária para substituir as pilhas do controle remoto:

1. Pegar as pilhas novas.
2. Abrir o controle remoto.
3. Retirar as pilhas usadas.
4. Colocar as pilhas novas.
5. Fechar o controle remoto.
6. Testar o controle remoto.
7. Colocar as pilhas usadas no lixo apropriado.

É importante entender que um algoritmo descreve uma possível solução para um problema. Isto significa que um algoritmo pode não resolver o problema de forma satisfatória, mas também significa que mesmo que o algoritmo gere o resultado esperado, ele não é a única forma de se resolver o problema.

É extremamente comum, e até esperado, que duas pessoas escrevam algoritmos diferentes para resolver um mesmo problema. Logo, o nível de eficiência e eficácia de cada um dos algoritmos será influenciado por vários fatores, porém talvez os dois principais sejam o domínio que a pessoa tem sobre o problema e o seu nível de conhecimento sobre **lógica de programação**.

Por isso, se dois algoritmos diferentes resolvem o mesmo problema, não há como dizer que um está correto e o outro incorreto. O que é possível é que um algoritmo seja mais eficiente do que o outro.

Existem algumas características que definem os algoritmos, e que devem ser consideradas quando da sua criação:

- **Finitude:** um algoritmo deve ter um número finito de passos.
- **Exatidão ou definição:** todas as etapas que compõem um algoritmo devem ser claramente definidas e ordenadas, sem margem para interpretações ambíguas.
- **Entradas e saídas determinadas:** todos os dados de entrada e saída do algoritmo devem estar explicitados. Um algoritmo pode ter zero ou mais entradas mas deve ter ao menos uma saída.
- **Efetividade:** o algoritmo deve solucionar o problema a que se propõe.
- **Eficiência:** o algoritmo deve ser o mais eficiente possível, buscando sempre a melhor combinação de três fatores: tempo, esforço e recursos necessários.

Algoritmo X Software

De uma maneira bastante simples, podemos dizer que um *software* é composto por um conjunto de instruções, escritas em uma sequência lógica, que permitem ao computador resolver um determinado problema. Ou seja, os *softwares* são baseados em algoritmos.

Mas, para que o computador consiga interpretar as instruções, estas devem ser descritas em linguagens específicas, chamadas de **linguagens de programação**. Um algoritmo, quando escrito em linguagem de programação, é chamado de código fonte.

No entanto, quando estamos pensando na melhor forma de resolver um problema computacional, é melhor utilizarmos uma linguagem mais próxima à natural, principalmente pela facilidade de compreensão. E para isso, existem diversas formas de representação de algoritmos.

Representação de Algoritmos

Um algoritmo pode ser representado de várias formas, dependendo dos objetivos e das preferências de seu desenvolvedor. As principais formas de representação de algoritmos são:

- Descrição Narrativa;
- Pseudocódigo;
- Fluxograma;
- Diagrama de Chapin.

Descrição Narrativa

É a descrição dos passos a serem executados pelo algoritmo, feita diretamente em linguagem natural. Os passos são listados um após o outro, na sequência em que devem ser executados, cada um em uma nova linha de texto.

Exemplo de algoritmo para trocar uma lâmpada:

1. Pegar a escada.
2. Posicionar a escada sob a lâmpada.
3. Pegar a lâmpada nova.
4. Subir na escada.
5. Remover a lâmpada antiga.
6. Colocar a lâmpada nova.
7. Descer da escada.
8. Colocar a lâmpada antiga no lixo.
9. Guardar a escada.

A descrição narrativa, embora de fácil compreensão, é totalmente informal e não padronizada. E por ser desprovida de regras, tende a tornar a descrição do algoritmo bastante imprecisa. O resultado acaba sendo a criação de algoritmos que não atendem as características esperadas.

Pseudocódigo

O pseudocódigo, que também é conhecido como português estruturado ou **portugol**, é uma técnica mais formal e estruturada uma vez que possui algumas regras definidas. Normalmente, estas regras são próximas às adotadas pelas linguagens de programação, o que é bastante positivo para o estudante, pois quando ele tiver contato com as linguagens de programação seu aprendizado será mais fácil. Por isso, o pseudocódigo será bastante utilizado ao longo deste livro.

Talvez um dos maiores benefícios do pseudocódigo esteja no fato de ele nos oferecer a estruturação básica utilizada nas linguagens de programação, aliada à facilidade de compreensão da nossa linguagem natural, o que possibilita uma maior concentração no entendimento do problema e dos passos necessários para sua resolução. Este é o principal motivo pelo qual o trabalho com pseudocódigo facilita o entendimento e a aprendizagem da lógica de programação.

As linguagens de programação são praticamente todas compostas por comandos representados por palavras da língua inglesa. O pseudocódigo permite que descrevamos os comandos que compõe o algoritmo em uma linguagem mais amigável para nós, programadores, analistas, etc.

Mais adiante, veremos que os comandos que utilizamos no pseudocódigo equivalem aos comandos de linguagem de programação. Ou seja, o português estruturado nada mais é do que uma forma de utilizar o português para representar comandos que serão passados ao computador. Posteriormente, iremos traduzir alguns dos nossos programas feitos em pseudocódigo para uma linguagem real de programação e, aí sim, os programas poderão ser executados no computador.

Como a lógica já estará pronta, teremos apenas que substituir os comandos do português estruturado pelos comandos reais da linguagem de programação a ser utilizada, que, no nosso caso, será a linguagem **Pascal**.

Veja o mesmo algoritmo do exemplo anterior, representado em pseudocódigo:

```
{Algoritmo que descreve os passos para trocar uma lâmpada}
Algoritmo TrocaLampada
    Início
        Pegar a escada;
        Posicionar a escada sob a lâmpada;
        Pegar a lâmpada nova;
        Subir na escada;
        Remover a lâmpada antiga;
        Colocar a lâmpada nova;
        Descer da escada; {este é outro comentário}
        Colocar a lâmpada antiga no lixo;
        Guardar a escada;
    Fim
```

Padrões para Representação de Algoritmos Por Meio de Pseudocódigo

- Na primeira linha, após a palavra **Algoritmo**, é informado o nome do algoritmo que está sendo descrito.
- O começo dos passos de execução do algoritmo é demarcado pela palavra **Início**.
- O término dos passos de execução do algoritmo é demarcado pela palavra **Fim**.
- O final de cada passo é marcado por um ponto e vírgula (;).
- Todos os comandos entre **Início** e **Fim** estão levemente deslocados para a **direita**. Este deslocamento se chama **Identação** e, normalmente, ocupa o espaço equivalente a três letras.

Existe também a possibilidade de acrescentarmos comentários nos algoritmos, a fim de facilitar seu entendimento para outras pessoas. Além disso, é bastante comum a própria pessoa que desenvolveu um algoritmo depois de um tempo não lembrar o porquê de uma determinada sequência de comandos ter sido definida. Isso acontece geralmente em algoritmos mais complexos. Por isso, é uma boa prática sempre incluir comentários nos seus algoritmos.

Para incluir um comentário em um algoritmo basta utilizar chaves ({ }), envolvendo o texto do comentário. Observe que no algoritmo anterior existem dois comentários, um deles utilizando uma linha inteira e o outro inserido depois de um passo do algoritmo. Um comentário também pode ocupar várias linhas de texto.

Neste primeiro exemplo de pseudocódigo, os passos a serem executados estão escritos de forma bastante informal. Porém, quando estivermos construindo um algoritmo que represente um *software*, os passos deverão ser informados de forma mais concisa e estruturada, respeitando algumas regras, pois representarão comandos que serão executados pelo computador.

Regras para Nomenclatura de Algoritmos

- Nomes de algoritmos devem conter apenas letras, números e o caractere *underscore* (_).
- Embora seja possível declarar todo o nome em letras maiúsculas, é recomendado, por convenção, utilizar letras maiúsculas apenas no início de cada palavra que compõe o nome do algoritmo. Estas palavras podem ser escritas todas juntas, ou separadas pelo caractere *underscore*.

Exemplos de nomes válidos	Exemplos de nomes inválidos
Calcular_IMC	Calcular_IMC
RealizarSaque	1saque
FECHAR_CONTA	#Fechar_conta

Todos estes padrões facilitam a criação de algoritmos consistentes e de boa legibilidade e por isso iremos utilizá-los daqui para a frente, com outras regras que serão apresentadas ao longo do livro.

Fluxograma

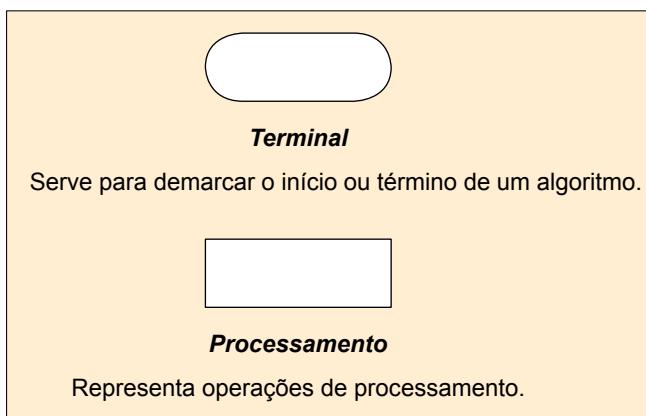
Os **fluxogramas**, que também são chamados de diagramas de blocos, nos permitem dar uma representação visual para o algoritmo, facilitando sua compreensão.

Para criar um fluxograma utilizam-se figuras geométricas, cada uma com um significado diferente, e dentro das quais são colocadas as instruções referentes a cada passo do algoritmo.

A indicação do fluxo de execução é feita por meio de setas, estando sempre o início do algoritmo na parte de cima da figura e seu término na parte de baixo.

À direita, temos o algoritmo referente à troca da lâmpada, representado em um fluxograma.

Por se tratar de um algoritmo bastante simples, foram necessários apenas dois tipos de figuras:



Uma desvantagem dos fluxogramas é que eles consomem bastante espaço, o que limita sua utilização a algoritmos de pequeno e médio porte.

Existem várias outras figuras utilizadas na construção de fluxogramas, as quais iremos conhecendo conforme avançarmos no conteúdo deste livro. Uma listagem completa de todas as figuras utilizadas nos fluxogramas ao longo do livro está disponível no Apêndice 1, na página 134.

O fluxograma e o pseudocódigo são as duas formas mais utilizadas na representação de algoritmos.

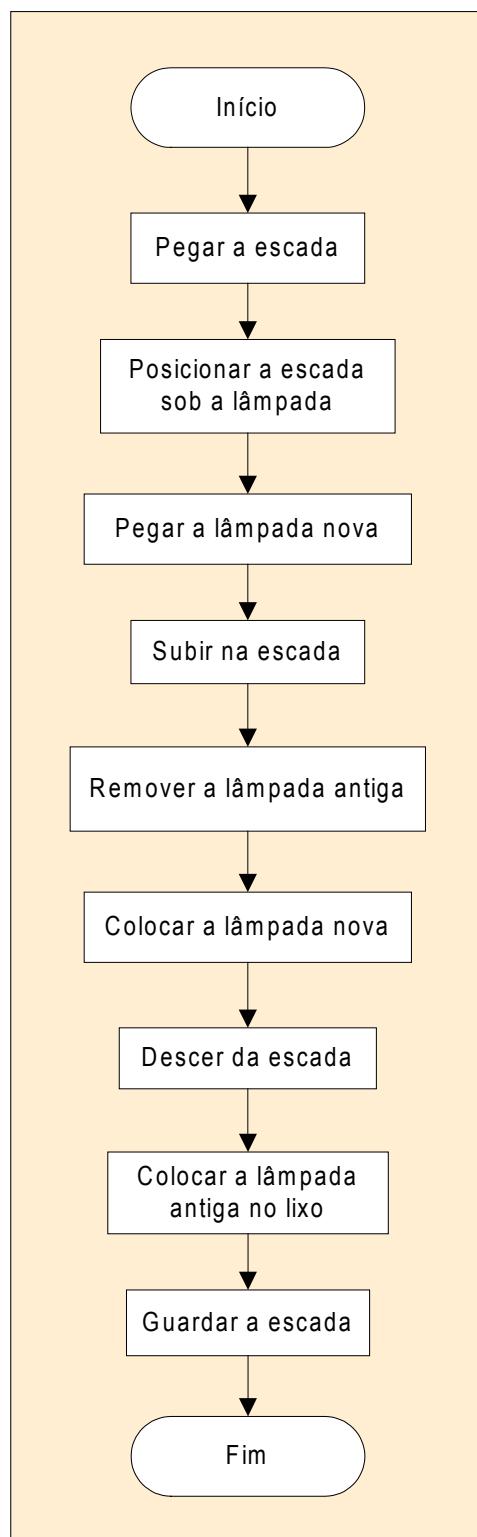


Diagrama de Chapin

O diagrama de Chapin, também conhecido como diagrama de Nassi-Schneiderman, é uma forma de representação hierárquica da lógica do programa. Este diagrama é construído em um grande quadro, dividido em blocos à medida em que as ações de processamento vão sendo inseridas.

Veja o algoritmo para troca de uma lâmpada, agora representado no diagrama de Chapin:

Início
Pegar a escada
Posicionar a escada sob a lâmpada
Pegar a lâmpada nova
Subir na escada
Remover a lâmpada antiga
Colocar a lâmpada nova
Descer da escada
Colocar a lâmpada antiga no lixo
Guardar a escada
Fim

Novamente, por se tratar de um algoritmo bastante simples, o quadro que compõe o algoritmo está apenas dividido em blocos horizontais para abrigar cada um dos passos de processamento.

Veremos exemplos e outros detalhes sobre este diagrama mais adiante.



15

Atividades

- 1) Crie um algoritmo, utilizando a descrição narrativa, que descreva os passos necessários para trocar o pneu de um carro.
- 2) Crie um algoritmo, utilizando a descrição narrativa, que descreva os passos necessários para preparar café, utilizando uma cafeteira.
- 3) Crie um algoritmo, utilizando a descrição narrativa, que descreva os passos necessários para escovar os dentes.
- 4) Escreva um algoritmo, utilizando a descrição narrativa, que represente outra possível solução para o problema da troca das pilhas do controle remoto.
- 5) No exercício 1, você criou um algoritmo que descrevia os passos necessários para trocar o pneu de um carro, utilizando a descrição narrativa. Agora, represente este algoritmo, utilizando as três formas de representação estudadas neste capítulo:
 - a. pseudocódigo;
 - b. fluxograma;
 - c. diagrama de Chapin.

Variáveis, Tipos de Dados e Constantes

Variáveis

Ao longo dos passos executados pelos algoritmos, muitas vezes temos a necessidade de armazenar temporariamente algumas informações, que vão ser utilizadas ao longo do processamento. Na programação, estes dados armazenados temporariamente durante a execução de um algoritmo são chamados de **variáveis**.

Imagine, por exemplo, que iremos criar um algoritmo para calcular o índice de massa corporal de uma pessoa, dizendo, no final, se ela está abaixo, acima do peso ou no peso ideal. Precisaríamos saber o peso e a altura da pessoa para fazer o cálculo. Essas informações, após serem fornecidas pela pessoa, teriam que ficar armazenadas temporariamente em algum lugar até a realização do cálculo. É, portanto, para isso que servem as variáveis.

É o mesmo que anotarmos um valor, um nome ou um telefone no papel, de forma temporária, durante a execução de alguma tarefa.

De forma simples, podemos dizer que uma variável é um local que serve para armazenar valores e que é identificado por meio de um nome único. Assim, sempre que quisermos atribuir (escrever) ou ler o valor armazenado em uma variável, precisaremos saber o seu nome.

Como o próprio nome diz, uma variável pode ter seu valor alterado ao longo da execução do programa.

Falando de forma um pouco mais técnica, a memória do computador é organizada em pequenos compartimentos, cada um com um endereço. Pense nisso como uma rua, onde cada casa tem um número que serve para identificá-la. Pois bem, quando criamos uma variável, estamos dizendo ao computador que queremos utilizar um endereço de memória para armazenar valores e informamos o nome que queremos utilizar para ter acesso a esse endereço. O nome da variável nada mais é do que um identificador que serve para referenciá-la.

Existem regras para as definições de variáveis, as quais variam de uma linguagem de programação para outra. Algumas destas regras, porém, são comuns à maioria das linguagens e, por isso, também são adotadas na representação da declaração de variáveis em pseudocódigo.

Para declarar variáveis no pseudocódigo, utilizaremos os seguintes padrões:

- Nomes de variáveis podem conter apenas caracteres alfanuméricos (alfabéticos e numéricos) e o caractere *underscore* (`_`). Não se pode utilizar acentos, espaços, sinais de pontuação ou outros símbolos.
- Um nome de variável não pode iniciar com números.
- Para nomear variáveis, podem-se utilizar letras minúsculas ou maiúsculas. O recomendado, porém, é utilizar letras minúsculas, por ser a convenção mais utilizada na maioria das linguagens de programação.
- Quando a variável for receber um nome composto (possuir mais de uma palavra), utiliza-se o caractere *underscore* para separar as palavras. Outra opção, considerando a utilização de letras minúsculas no nome da variável, é unir as palavras digitando em maiúscula a letra que fará a ligação com a palavra anterior (atenção, a primeira letra da primeira palavra continua minúscula). Quando o nome da variável possui mais de uma palavra, dizemos que a variável possui um identificador composto.
- As variáveis devem ser declaradas em bloco próprio, no início do algoritmo ou na sub-rotina a que pertencem.

Exemplos de nomes válidos	Exemplos de nomes inválidos
<code>nome_candidato</code>	<code>nome candidato</code>
<code>endereco</code>	<code>endereço</code>
<code>RG</code>	<code>R.G.</code>
<code>mes_ferias</code>	<code>mês_férias</code>
<code>dataNasc</code>	<code>data-Nasc</code>
<code>fone1</code>	<code>1fone</code>

Além de respeitar as regras acima, quando for declarar variáveis em um algoritmo procure sempre utilizar nomes curtos e significativos. Exemplos:

- Ao invés de utilizar **temp1** e **temp2**, seria mais intuitivo **altura** e **peso**.
- Ao invés de **altura_maxima_permitida_veiculo**, utilizar **altura_max** ou **alturaMaxima**, que seriam boas opções.

Tipos de Dados

Quando declaramos uma variável, precisamos indicar o tipo de informação que desejamos armazenar nela. Existem diversos tipos de dados e muitos deles são comuns na grande maioria das linguagens de programação. No nosso estudo de lógica de programação, porém, utilizaremos apenas alguns dos principais.

Os tipos de dados básicos com os quais iremos trabalhar são:

- **Inteiro:** permite armazenar números inteiros, positivos ou negativos.

Exemplos de informações válidas em uma variável do tipo Inteiro

5
0
-2
1430

- **Real:** permite armazenar números inteiros ou fracionários, positivos ou negativos.

Exemplos de informações válidas em uma variável do tipo Real

5
0
-2
1,5
-3,72

- **Caractere:** permite armazenar caracteres alfanuméricos (ou seja: letras, números, espaços, sinais de pontuação e outros símbolos). Este tipo também é chamado de tipo Literal ou tipo **String**. Valores do tipo caractere são sempre representados entre aspas ("").

Exemplos de informações válidas em uma variável do tipo Caractere

"Av. Brasil, 1500"

"5"

"%"

"O tipo Caractere aceita tudo! @#\$%âõç."

- **Lógico:** permite armazenar valores lógicos, do tipo Verdadeiro ou Falso, os quais representaremos respectivamente por **V** e **F**.

Exemplos de informações válidas em uma variável do tipo Lógico

V
F

Declaração de Variáveis

Para que uma variável passe a existir e possa ser utilizada no contexto de um algoritmo, é necessário que ela seja declarada. A declaração nada mais é do que a definição de uma variável, aonde informamos o seu nome e o tipo de informação que ela deverá ser capaz de armazenar.

Para declarar variáveis no pseudocódigo, adotaremos o seguinte padrão:

```
<nome da variável> : <tipo de dado>;
```

A declaração deverá ser feita no começo do algoritmo, em um bloco nomeado **Variáveis**, antes do demarcador **Início**.

Veja um exemplo:

```
{Exemplo de declaração de variáveis I}
Algoritmo ExemploVariaveis
    Variáveis
        nome : Caractere;
        endereço : Caractere;
        altura : Real;
        peso : Real;
        telefone : Caractere; {Declaramos como caractere para permitir que
                               o telefone seja digitado com formatação. Ex: (00)1234-1234,
                               e também que seja possível informar ramal}
    Início
        ...
    Fim
```

19

Veja que, no exemplo acima, cada variável foi declarada em uma nova linha. Também é possível agrupar variáveis do mesmo tipo em uma mesma linha, declarando-as todas juntas.

Veja o exemplo a seguir:

```
{Exemplo de declaração de variáveis II}
Algoritmo ExemploVariaveisAgrupadas
    Variáveis
        nome, endereço, telefone : Caractere;
        altura, peso : Real;
    Início
        ...
    Fim
```

Lembre-se de que uma vez que uma variável tenha sido declarada, não é possível alterar seu nome nem seu tipo, mas apenas o valor que ela guarda.

Nos fluxogramas e diagramas de Chapin, a declaração de variáveis não costuma ser representada.

Atribuição e Inicialização de Variáveis

Atribuição é o ato de definir o valor de uma variável. Tecnicamente, isto significa escrever uma nova informação no espaço de memória identificado pelo nome que demos à variável no momento de sua declaração.

Para atribuir um novo valor a uma variável, adotaremos o seguinte padrão:

```
<nome da variável> := <valor>;
```

Veja os exemplos:

```
Algoritmo ExemploAtribuicao
Variáveis
    ano : Inteiro;
    nomeAluno : Caractere;
    altura : Real;
Início
    ano := 2010; {atribui o valor inteiro 2010 à variável ano}
    nomeAluno := "Pedro da Silva"; {atribui a sequência de
                                    caracteres "Pedro da Silva" à variável nomeAluno}
    altura := 172,5; {atribui o valor real 172,5 à variável altura}
Fim
```

20

Também fique atento ao fato de que a operação de atribuição apaga qualquer informação existente na variável, sobrepondo-a com o novo valor.

Por exemplo:

```
Algoritmo ExemploAtribuicao2
Variáveis
    x : Inteiro;
Início
    x := 1; {neste momento x vale 1}
    x := 5; {neste momento x passou a valer 5}
Fim
```

Iniciar uma variável significa atribuir-lhe um valor inicial, o que deve ser feito no começo do algoritmo, logo após o demarcador **Início**.

O ideal é sempre inicializarmos todas as variáveis, mesmo aquelas cujos valores ainda não sabemos quais serão. Neste caso, inicializamos variáveis dos tipos Inteiro e Real com zero (**0**), variáveis do tipo Caractere com “” (vazio) e variáveis do tipo Lógico com **F** (falso).

Embora seja fortemente recomendada, a inicialização de variáveis é uma convenção, considerada uma boa prática, e não uma regra. Por isso, caso não realizemos a inicialização de algumas variáveis, elas serão inicializadas automaticamente com o valor *default* (padrão), de acordo com o tipo de dado a que pertencem. Estes valores são:

Tipo da Variável	Valor Default
Inteiro	0
Real	0
Caractere	“” (vazio)
Lógico	F (falso)

Vejamos um novo exemplo de algoritmo, representado em pseudocódigo, agora contemplando o uso de variáveis:

```
{Calcula a quantidade de gasolina dado um valor em R$}
Algoritmo QuantidadeGasolina
Variáveis
    precoGasolina, valorDesejado, qtdeLitros : Real; {declaração}
Início
    precoGasolina := 2,48; {inicialização}
    valorDesejado := 0; {inicialização}
    qtdeLitros := 0; {inicialização}
    Escreva ("Informe o valor desejado pelo cliente:"); {saída em tela}
    Leia(valorDesejado); {entrada via teclado}
    qtdeLitros := valorDesejado / precoGasolina; {realiza o cálculo e atribui o resultado na variável qtdeLitros}
    Escreva("A quantidade de litros para este valor é:");
    Escreva(qtdeLitros);
Fim
```

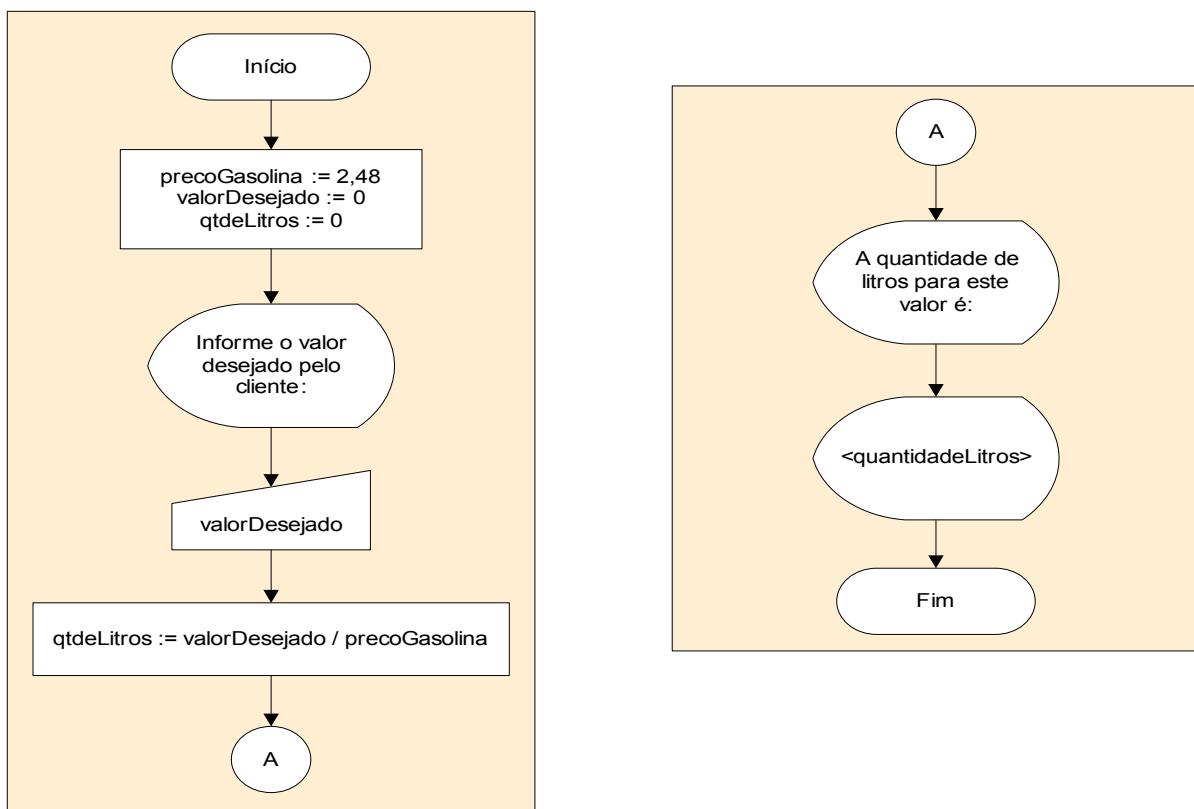
O exemplo anterior ainda apresenta duas outras novidades, os comandos de leitura e escrita que simulam a interação com o usuário:

- Para simular entradas de teclado, utilizamos o comando **Leia**;
- Para simular saídas na tela, utilizamos o comando **Escreva**.

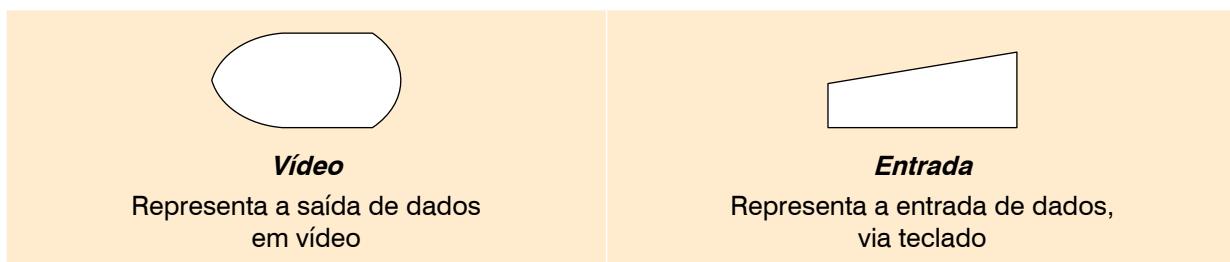
Observe que os dois comandos utilizam parênteses para delimitar o conteúdo a que se referem.

O algoritmo anterior, embora bastante simples, já poderia ser transformado em um programa de computador, pois simula todos os itens necessários para cumprir o seu propósito, que é calcular quanto deve ser colocado de gasolina para atingir um determinado valor em Reais.

Veja como ficaria o Fluxograma do algoritmo QuantidadeGasolina:



Perceba que foram utilizadas duas novas figuras, representando a interação do sistema com o usuário:



Além disso, utilizamos um conector para representar a continuação do algoritmo em outro ponto da página.

Constantes

As **constantes** têm basicamente a mesma função das variáveis, isto é, armazenar valores temporariamente, enquanto um algoritmo está sendo executado. Existe, porém, uma diferença crucial entre elas: as constantes, uma vez declaradas, não podem ter seu valor modificado (e este é o motivo pelo qual elas se chamam constantes). Ou seja, as constantes só podem ser usadas para armazenar valores que não se alterarão ao longo do algoritmo.

Assim como as variáveis, as constantes também devem ser declaradas em bloco próprio, no início do algoritmo.

Com relação à nomenclatura, valem as mesmas regras listadas anteriormente para a declaração de variáveis, porém com uma recomendação:

- Procurar sempre declarar as constantes, utilizando letras maiúsculas, como forma de diferenciá-las das variáveis de maneira mais fácil ao longo do algoritmo. Neste caso, a separação de palavras dos nomes compostos é feita por meio do caractere *underscore*. Esta não é uma regra e, sim, uma convenção utilizada por grande parte dos programadores.

Exemplos de nomes recomendados	Exemplos de nomes não recomendados
FORCA_ACELERACAO	forca_aceleracao
PI	pi
OPERACAO_PADRAO	operacaoPadrao

Exemplo de pseudocódigo, contendo declaração de constantes:

```
Algoritmo ExemploConstantes
    Constantes
        PI := 3,14; {constante do tipo Real}
        MAX_GRAUS := 90; {constante do tipo Inteiro}
        INSTITUICAO := "Escola X"; {constante do tipo Caractere}
        RELATORIO_ATIVADO := F; {constante do tipo Lógico}
    Variáveis
        ...
    Início
        ...
    Fim
```

23

Do mesmo modo que as variáveis, a declaração de constantes também não costuma ser representada nos fluxogramas nem nos diagramas de Chapin.



Atividades

- 1) Crie um algoritmo para calcular a média de consumo de combustível de um veículo qualquer. O usuário deverá informar: quilometragem inicial, quilometragem final e a quantidade de litros abastecida. Represente seu algoritmo utilizando pseudocódigo e fluxograma.
- 2) Crie um algoritmo para calcular a velocidade média atingida por um veículo durante uma viagem. Reflita sobre os dados que serão necessários solicitar ao usuário. Represente seu algoritmo utilizando pseudocódigo e fluxograma.

Operadores e Expressões

Operador é um elemento que indica a realização de uma operação sobre componentes de uma expressão, gerando consequentemente um resultado.

Os operadores estão presentes em qualquer tipo de expressão e são utilizados para realizar cálculos, comparações ou definir critérios. Qualquer expressão, por menor que seja, sempre possuirá ao menos um operador.

Conforme formos avançando nos exemplos, veremos que é possível utilizar mais de um tipo de operador em uma mesma expressão, e que isso é bastante comum em expressões um pouco mais complexas.

Existem quatro tipos de operadores, que serão detalhados a seguir.

Operadores Aritméticos

Os **operadores aritméticos** são aqueles utilizados nos cálculos matemáticos. Desde os anos iniciais na escola, nós já aprendemos a utilizar estes operadores.

Na lógica de programação, estes operadores são utilizados apenas com variáveis numéricas (dos tipos Inteiro e Real), gerando também uma saída numérica.

A tabela abaixo lista os operadores aritméticos por ordem de precedência.

Precedência	Operador	Descrição
1	$^$	Potenciação
2	$*$	Multiplicação
2	$/$	Divisão
3	$+$	Adição
3	$-$	Subtração

A ordem de precedência nada mais é do que uma preferência de execução. Por exemplo, na expressão existente na figura abaixo, a multiplicação é realizada antes da subtração, devido à ordem de precedência dos operadores:

$$7 - \underbrace{2 * 3}_{12} \\ 29$$

Vamos aproveitar para relembrar um item estudado por você anteriormente, nas disciplinas de matemática: o uso dos parênteses.

Os parênteses servem para definir a ordem em que uma expressão deve ser resolvida, sendo que a resolução dela se dá sempre do parêntese mais interno para o mais externo, independente de sua posição na expressão. Por exemplo:

$$3 + (\underbrace{(1 - 4)}_{1^{\text{a}}}) \underbrace{* 5}_{2^{\text{a}}} \underbrace{\phantom{1^{\text{a}} * 5}}_{3^{\text{a}}}$$

A ordem de precedência dos operadores passa a valer no momento em que existem dois ou mais operadores em um mesmo nível de parênteses (ou fora deles). Exemplo:

$$\underbrace{7 + 1}_{2^{\text{a}}} - \underbrace{5 * 3}_{1^{\text{a}}} \underbrace{}_{3^{\text{a}}}$$

No exemplo anterior, todos os operadores estão no mesmo nível, pois não há parênteses. O que ocorre, então, é a resolução da expressão na sequência definida pela ordem de precedência dos operadores. Contudo, mesmo que desejemos que a expressão seja resolvida nessa sequência, o melhor é empregar o uso dos parênteses para enfatizar a sequência de resolução para quem for ler a expressão, conforme o exemplo a seguir:

$$\underbrace{7 + 1}_{2^{\text{a}}} - \underbrace{(5 * 3)}_{1^{\text{a}}} \underbrace{}_{3^{\text{a}}}$$

Para melhorar a clareza das expressões, torna-se interessante adotarmos a seguinte regra: empregar parênteses quando for utilizar operadores de prioridades diferentes. Contudo, cabe a quem está construindo a expressão utilizar o bom senso para manter a legibilidade da mesma.

Vejamos um exemplo mais prático:

$$\text{qtdeCarne} = (\underbrace{(\text{consumoMedioM} * \text{qtdeM})}_{1^{\text{a}}} + \underbrace{(\text{consumoMedioH} * \text{qtdeH})}_{2^{\text{a}}}) * 1,2 \underbrace{\phantom{(\text{consumoMedioM} * \text{qtdeM}) + (\text{consumoMedioH} * \text{qtdeH})}}_{3^{\text{a}}} \underbrace{\phantom{(\text{consumoMedioM} * \text{qtdeM}) + (\text{consumoMedioH} * \text{qtdeH}) * 1,2}}_{4^{\text{a}}}$$

Os cálculos realizados na expressão anterior poderiam ser decompostos em várias linhas, como no algoritmo abaixo:

```
{Calcula a quantidade de carne necessária para um churrasco}
Algoritmo CalculaCarneChurrasco
Constantes
    MARGEM_SEGURANCA := 1,2; {para garantir que não faltará carne!}
Variáveis
    consumoMedioM, consumoMedioH : Real;
    qtdeM, qtdeH : Inteiro;
    consumoTotalM, consumoTotalH, total : Real;
Início
    Escreva("CÁLCULO DE CARNE PARA CHURRASCO");
    Escreva("Digite o consumo médio feminino:");
    Leia(consumoMedioM);
    Escreva("Digite a quantidade de mulheres:");
    Leia(qtdeM);
    Escreva("Digite o consumo médio masculino:");
    Leia(consumoMedioH);
    Escreva("Digite a quantidade de homens:");
    Leia("qtdeH");

    consumoTotalM := consumoMedioM * qtdeM;
    consumoTotalH := consumoMedioH * qtdeH;
    total := (consumoTotalM + consumoTotalH) * MARGEM_SEGURANCA;

    Escreva("A quantidade total de carne para o churrasco é:");
    Escreva(total);
Fim
```

Em muitos casos, porém, iremos preferir realizar todos os cálculos em uma única expressão, para deixar o algoritmo mais compacto. Veja:

```
{Calcula a quantidade de carne necessária para um churrasco}
Algoritmo CalculaCarneChurrasco
Constantes
    MARGEM_SEGURANCA := 1,2; {para garantir que não faltará carne!}
Variáveis
    consumoMedioM, consumoMedioH, qtdeCarne : Real;
    qtdeM, qtdeH : Inteiro;
Início
    Escreva("CÁLCULO DE CARNE PARA CHURRASCO");
    Escreva("Digite o consumo médio feminino:");
    Leia(consumoMedioM);
```

```

Escreva("Digite a quantidade de mulheres:");
Leia(qtdeM);
Escreva("Digite o consumo médio masculino:");
Leia(consumoMedioH)
Escreva("Digite a quantidade de homens:");
Leia("qtdeH");

    qtdeCarne := ((consumoMedioM * qtdeM) + (consumoMedioH * qtdeH))
    * MARGEM_SEGURANCA;

    Escreva("A quantidade total de carne para o churrasco é:");
    Escreva(qtdeCarne);

Fim

```

Operadores Relacionais

Os **operadores relacionais** permitem realizar comparações entre valores, gerando como resultado um valor lógico (Verdadeiro ou Falso). Você deve se lembrar desses operadores das suas aulas de matemática, porém, no contexto da lógica de programação, estes operadores permitem comparar quaisquer tipos de dados, desde que os valores a serem comparados sejam do mesmo tipo.

A tabela abaixo lista os operadores relacionais:

Operador	Descrição
=	Igual
<>	Diferente
<	Menor
<=	Menor ou Igual
>	Maior
>=	Maior ou Igual

Veja alguns exemplos:

```

A >= B
(3 + 7) < (2 * 4)
4 >= ((3 - 1) * 2)
(consumoMedio * qtdePessoas) > qtdeDisponivel

```

Perceba que a maioria dos exemplos anteriores utiliza os operadores relacionais em conjunto com operadores aritméticos.

Exemplo de uso em algoritmo:

```
Algoritmo ExemploOperadoresRelacionais
Variáveis
    pedidoMinimo, qtdePedido, qtdeEstoque : Inteiro;
    respeitaMinimo, podeAtender : Lógico;
Início
    pedidoMinimo := 10;
    qtdePedido := 20;
    qtdeEstoque := 18;

    respeitaMinimo := qtdePedido >= pedidoMinimo;
    {após a linha acima, a variável respeitaMinimo terá o valor lógico
    VERDADEIRO (V), indicando que o pedido
    respeita a quantidade mínima exigida}

    podeAtender = qtdePedido < qtdeEstoque;
    {após a linha acima, a variável podeAtender terá o
    valor lógico FALSO (F), indicando que não é possível
    atender o pedido, pois não há estoque suficiente}
Fim
```

28

Como o objetivo desta categoria de operadores é estabelecer comparações, não existe uma ordem de precedência estabelecida, pois estas comparações são realizadas sempre entre dois valores.

Operadores Lógicos

Assim como os relacionais, estes operadores também geram valores lógicos. Porém, ao contrário dos operadores relacionais que podem comparar qualquer tipo de dado, os operadores lógicos trabalham apenas com valores lógicos.

Os operadores lógicos são:

Precedência	Operador	Operação
1	.NÃO.	Negação
2	.E.	Conjunção
3	.OU.	Disjunção

O uso destes operadores geram as chamadas **operações lógicas**, nomeadas na tabela acima.

Negação

Negação é uma operação lógica que gera como saída um valor inverso ao valor lógico de entrada. O operador **NÃO** também é usualmente representado por **!** ou **NOT**.

Exemplos:

```
.NÃO. varA
```

```
.NÃO. (5 > 3)
```

O operador **NÃO** sempre deve ser inserido à esquerda da variável ou expressão cujo valor deve modificar.

Exemplo em algoritmo:

```
Algoritmo ExemploNao
Variáveis
    X, Y : Inteiro;
    L, M : Lógico;
Início
    X := 1;
    Y := 2;
    L := (X > Y); {A variável L receberá o valor F (falso)}
    M := .NÃO.(X > Y); {A variável M receberá o valor V(verdadeiro)}
Fim
```

Existe uma técnica chamada de **tabela verdade** ou **tabela de verificação**, que serve para ilustrar o resultado de associações lógicas. A tabela verdade, do operador **NÃO**, é bastante simples:

A	.NÃO. A
V	F
F	V

Conjunção

A **conjunção** é uma operação lógica que relaciona dois valores lógicos por meio do operador **E**. Este operador também é representado por **&&** e **AND**.

A operação de conjunção relaciona dois valores lógicos, gerando um valor lógico de saída, que será verdadeiro somente se os dois valores de entrada forem verdadeiros.

Por exemplo:

```
varA .E. varB
```

```
(altura <= alturaMax) .E. (largura <= larguraMax)
```

Rpare que o segundo exemplo utiliza também operadores relacionais.

Agora em um algoritmo:

```
Algoritmo ExemploE
Variáveis
    X, Y, Z : Inteiro;
    L, M, N, O, P : Lógico;
Início
    X := 1;
    Y := 2;
    Z := 3;

    L := (X > Y); {A variável L receberá o valor F (falso)}
    M := (Z > Y); {A variável M receberá o valor V (verdadeiro)}
    N := V; {A variável N receberá o valor V (verdadeiro)}

    O := L .E. M {A variável O receberá o valor F (falso)
                   porque L é falso}
    P := M .E. N {A variável O receberá o valor V (verdadeiro)
                   porque M e N são verdadeiros}

Fim
```

30

Para aprofundar o entendimento destas associações, analise a tabela verdade do operador **E**:

A	B	A .E. B
F	F	F
V	F	F
F	V	F
V	V	V

Disjunção

A **disjunção** é uma operação lógica que relaciona dois valores lógicos por meio do operador **OU**, também representado por **||** e **OR**.

Esta operação relaciona dois valores lógicos, gerando um valor lógico de saída que será verdadeiro se pelo menos um dos valores de entrada for verdadeiro.

Exemplos:

```
varA .OU. varB  
(altura > alturaMax) .OU. (largura > larguraMax)
```

Vejamos o uso em um algoritmo:

```
Algoritmo ExemploOU
Variáveis
    X, Y, Z : Inteiro;
    L, M, N, O, P : Lógico;
Início
    X := 1;
    Y := 2;
    Z := 3;

    L := (X > Y); {A variável L receberá o valor F (falso)}
    M := (Z > Y); {A variável M receberá o valor V (verdadeiro)}
    N := F; {A variável N receberá o valor F (falso)}

    O := L .OU. M {A variável O receberá o valor V (verdadeiro)
                    porque M é verdadeiro}
    P := L .OU. N {A variável O receberá o valor F (falso)
                    porque L e N são falsos}

Fim
```

31

Confira a tabela verdade do operador **OU**:

A	B	A .OU. B
F	F	F
V	F	V
F	V	V
V	V	V

Disjunção Exclusiva

Existe um outro operador lógico, representado por **XOU** ou **XOR**, que dá origem a uma operação chamada **disjunção exclusiva**. Este operador é menos importante para nossos estudos sobre lógica de programação e não será abordado neste momento.

Operador Literal

O **operador literal** tem o objetivo de unir duas informações do tipo texto. Este processo é chamado de concatenação.

Operador	Descrição
+	Concatenação

Exemplos:

```
"Conca" + "teração"
```

```
"Prezado(a)" + nomeCliente
```

Algoritmo de exemplo:

```
Algoritmo ExemploConcat
Variáveis
    nome, sobrenome, nomeCompleto : Caractere;
Início
    Escreva("Digite seu nome:");
    Leia(nome);
    Escreva("Digite seu sobrenome:");
    Leia(sobrenome);

    nomeCompleto := nome + " " + sobrenome; {concatena o nome,
        um espaço e o sobrenome, atribuindo o resultado à
        variável nomeCompleto }

    Escreva("O primeiro nome é: ");
    Escreva(nome);

    Escreva("O nome completo é: " + nomeCompleto); {aqui estamos
        concatenando diretamente no comando de saída}
Fim
```

32

Você já deve ter percebido que o símbolo que representa o operador de concatenação é o mesmo que representa o operador aritmético de soma. O que irá definir se será realizada uma soma ou uma concatenação é o tipo de informação que estiver sendo empregada em conjunto com o operador. Veja:

{Exemplos de concatenação e diferenças com relação à adição}

Algoritmo TestaConcatenar

Variáveis

```
a, b : Caractere;  
c, d : Inteiro;
```

Inicio

```
a := "1";  
b := "2";  
c := 3;  
d := 4;
```

```
Escreva(a + b); {o resultado será o valor caractere "12"}
```

```
Escreva(c + d); {o resultado será o valor inteiro 7}
```

```
Escreva(a + c); {o resultado será o valor caractere "13"}
```

```
Escreva(c + a); {está incorreto, geraria um erro}
```

Fim

33

Na grande maioria das linguagens de programação, a concatenação converte o valor à direita do operador “+” para o tipo Caractere. Por isso, se tentarmos concatenar um texto com um valor numérico e colocarmos o valor numérico à direita, estamos indicando que o valor deve ser convertido para o tipo Caractere e deve ser realizada uma concatenação com o valor à esquerda. Porém, se o valor numérico estiver à esquerda do operador e o texto à direita, ocorrerá um erro, pois estaremos indicando que queremos fazer uma soma, o que não pode ser feito com uma informação do tipo Caractere. Ou seja:

{Ilustra a tentativa de concatenar Caractere com Inteiro}

Algoritmo DetalheConcat

Variáveis

```
idadeAluno : Inteiro;
```

Inicio

```
idadeAluno := 18;
```

```
Imprima("A idade do aluno é " + idadeAluno); {está correto}
```

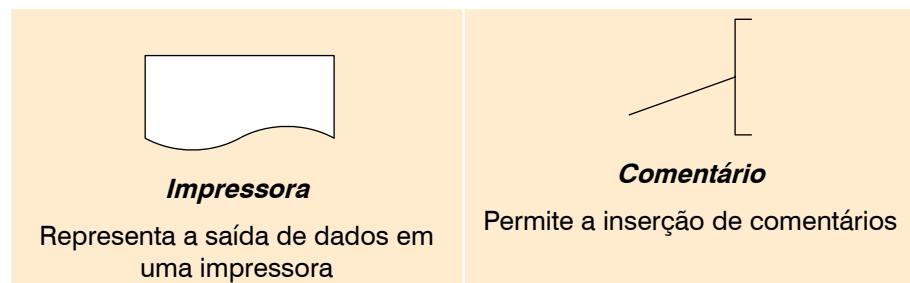
```
Imprima(idadeAluno + " é a idade do aluno."); {está incorreto}
```

Fim

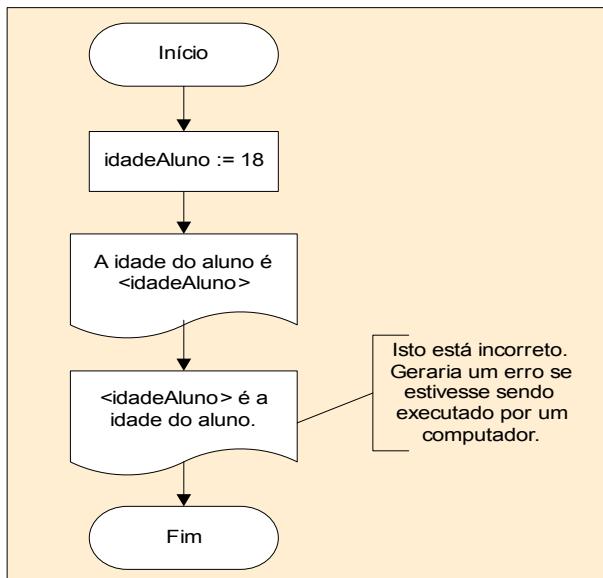
No exemplo acima, existe um novo comando: **Imprima**. Em pseudocódigo, este comando serve para representar o envio de informações para uma impressora.

A representação de impressão nos fluxogramas também é feita por uma figura específica, que está representada na tabela a seguir. Além disso, também é possível inserir comentários nos fluxogramas, mas há um detalhe importante: sempre procurar apontar o comentário para o passo do algoritmo a que ele se refere.

Veja as duas novas figuras:



Confira um exemplo de uso no fluxograma abaixo:



34

Teste de Mesa

O **teste de mesa**, também conhecido como teste do chinês ou chinesinho, é um processo de teste manual que permite verificar a eficiência dos algoritmos.

Os testes são realizados por meio de simulações. Nelas, são atribuídos valores de entrada fictícios, sendo possível acompanhar o valor das variáveis do sistema ao longo da execução do algoritmo, o que permite encontrar erros e identificar suas causas.

Funcionamento

Veja um passo a passo de como este processo funciona:

1. Numere cada uma das linhas do algoritmo que deseja testar (somente entre os demarcadores **Início** e **Fim**);
2. Crie uma tabela contendo:
 - a. Uma linha para cada passo do algoritmo;
 - b. Uma coluna para cada variável declarada;

- c. Uma coluna para representar saídas em tela;
 - d. Uma coluna para representar saídas em impressora;
3. Percorra o algoritmo, linha a linha, anotando todas as mudanças. A cada linha do algoritmo percorrida, deve-se preencher uma nova linha no teste de mesa, com os valores de todas as variáveis e as saídas realizadas.

Vamos a um exemplo prático. O algoritmo com linhas numeradas:

```

Algoritmo IlustraTesteDeMesa
Variáveis
    idade, idadeMin : Inteiro;
    idadeOK, autorizadoPais, liberado : Lógico;
Início
01    idadeMin := 18;
02    Leia(idade); {para teste vamos simular que foi informado 16}
03    Leia(autorizadoPais); {para teste, V}
04    idadeOK := idade >= idadeMin;
05    liberado := idadeOK .OU. autorizadoPais;
06    Escreva("Resultado: " + liberado);
Fim

```

A tabela com os dados da execução:

Nº Linha	idade	idadeMin	autorizadoPais	idadeOK	liberado	Tela	Impressora
1		18					
2	16	18					
3	16	18	V				
4	16	18	V	F			
5	16	18	V	F	V		
6	16	18	V	F	V	Resultado: V	

As constantes declaradas no algoritmo não precisam ser representadas na tabela, já que seus valores não se alteram ao longo do tempo.

Por meio do teste de mesa, é possível realizar as mais variadas simulações, porém, quando o algoritmo é grande, o processo acaba tornando-se um pouco trabalhoso. Nesses casos, pode-se optar por testar apenas a parte do algoritmo mais propensa a erros.



Atividades

- 1) Crie um algoritmo que solicite o nome e a idade do usuário e, em seguida, imprima estas informações de forma concatenada em uma frase de boas vindas. Represente seu algoritmo em pseudocódigo e fluxograma.
- 2) Considerando os operadores relacionais, crie um algoritmo em pseudocódigo que leia dois números e escreva na tela se o primeiro é maior do que o segundo. A saída deverá ser do tipo V ou F.
- 3) Crie um algoritmo em pseudocódigo que leia o valor de X e escreva na tela o valor de X^2 .
- 4) Crie um algoritmo em pseudocódigo para calcular a taxa de serviço do garçom, a partir da entrada do valor da conta. A taxa de serviço é fixa em 10%. O sistema deverá escrever na tela o valor da taxa de serviço e depois o valor total a ser pago.

Em seguida, você deverá fazer um teste de mesa para o algoritmo.

- 5) Faça o teste de mesa para o seguinte algoritmo:

```
Algoritmo IlustraTesteDeMesa
Variáveis
    valA, valB, valC : Inteiro;
    X, Y, Z : Lógico;
Início
01    valA := 5;
02    Leia(valB); {simular o valor 7}
03    valC := valA + (valB - valA);
04    Escreva(valC);
05    X := valC > valA;
06    Y := valA >= valB;
07    Escreva(valB <> valC);
08    Imprima(X .OU. Y);
09    Z := X .E. .NÃO.Y;
10    Escreva(Z);
Fim
```

Estruturas de Controle

Seleção

Até agora, nossos algoritmos eram totalmente sequenciais, ou seja, todos os passos eram executados em sequência, sem nenhum tipo de modificação no fluxo do programa. Na prática, este tipo de algoritmo não é muito útil, porque mesmo os problemas minimamente complexos já exigem algum tipo de controle sobre os passos que serão executados.

A partir deste ponto, veremos que é possível adicionar estruturas de modificação de fluxo nos algoritmos, as quais são denominadas **estruturas de controle**.

A primeira estrutura de controle que iremos conhecer é a **estrutura de seleção**, que permite selecionar os passos que devem ser executados pelo algoritmo em um determinado ponto. Esta estrutura também é chamada de **estrutura de decisão** ou **estrutura condicional**.

A seleção dos passos, que devem ou não ser executados, é feita a partir do resultado de uma expressão lógica ou relacional. Na prática, isto representa dotar o algoritmo de um mecanismo que lhe permita tomar decisões em tempo real, buscando atender a critérios preestabelecidos.

Então, sempre que precisarmos tomar uma decisão em algum ponto do algoritmo, iremos utilizar uma estrutura de seleção.

Estrutura de Seleção Simples

A **estrutura de seleção simples** permite definir um bloco de instruções que serão executadas apenas se forem atendidos os critérios definidos. Esta estrutura também é conhecida como **desvio condicional simples**.

No pseudocódigo, a estrutura de seleção simples é representada pelo comando **Se**, que utiliza a seguinte estrutura:

```
Se (<condição>) Então  
...  
Fim_Se
```

Todos os comandos existentes entre **Então** e **Fim_Se** só serão executados se a condição for atendida. Por exemplo:

```
{Exemplo de uso do comando Se}
Algoritmo ExemploSe
Variáveis
    idadeAluno : Inteiro;
Início
01    Escreva("Digite a idade do aluno:");
02    Leia(idadeAluno);

03    Se(idadeAluno < 18) Então
04        Escreva("O aluno é menor de idade.");
05    Fim_Se

06    Escreva("A idade do aluno é: " + idadeAluno);
Fim
```

No algoritmo acima, a frase “O aluno é menor de idade.” só será impressa na tela se o resultado da expressão (**idadeAluno < 18**) for verdadeiro.

Para verificar o funcionamento do algoritmo, vamos realizar dois testes de mesa, simulando diferentes valores de entrada.

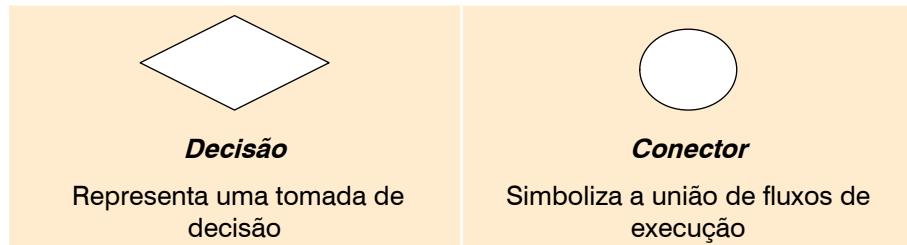
Primeiro teste: simulando que a idade digitada seja 19:

Nº Linha	idadeAluno	Tela
1		Digite a idade do aluno:
2	19	
3	19	
5	19	
6		A idade do aluno é: 19

Segundo teste: simulando que a idade digitada seja 15:

Nº Linha	idadeAluno	Tela
1		Digite a idade do aluno:
2	15	
3	15	
4	15	O aluno é menor de idade.
5	15	
6		A idade do aluno é: 15

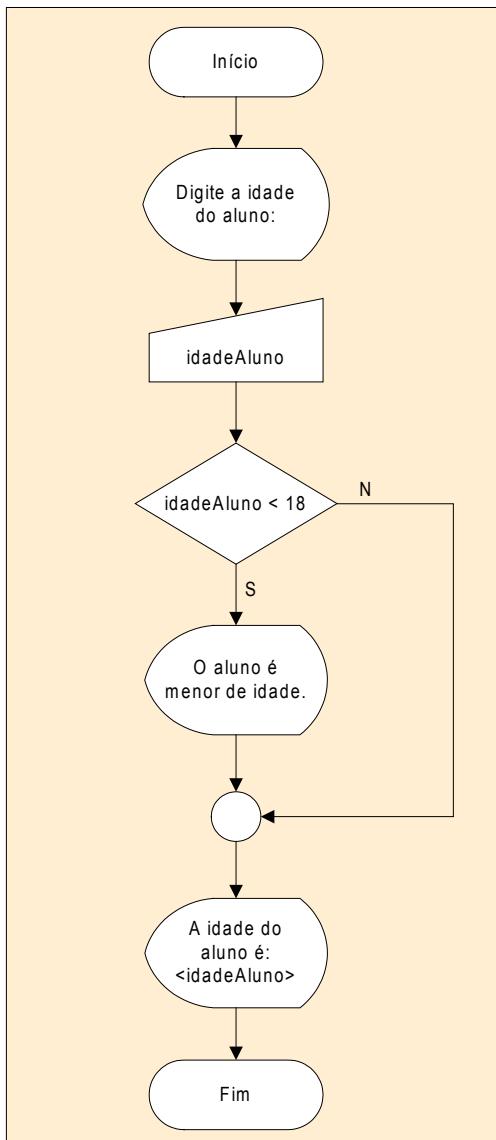
Nos fluxogramas, a representação de estruturas de seleção é feita por meio das seguintes figuras:



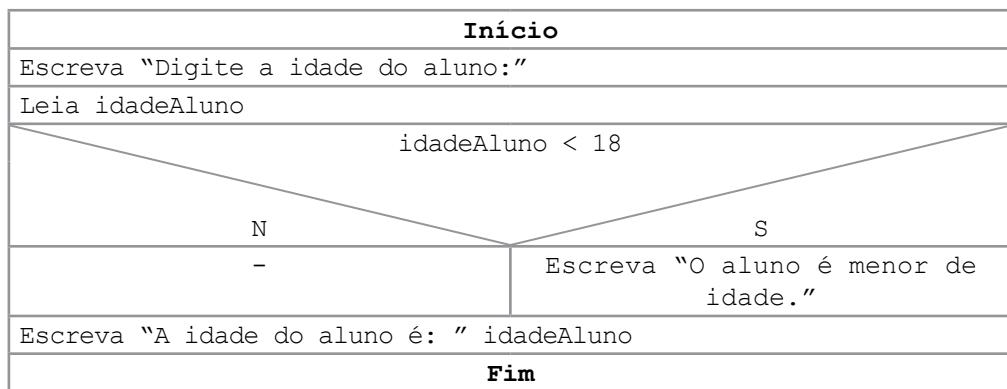
O losango é utilizado para representar decisões. Possui um ou mais fluxos de entrada e sempre dois fluxos de saída (uma vez que a expressão analisada resultará sempre em verdadeiro ou falso).

O conector, já visto anteriormente com a função de representar a continuação do algoritmo em outro ponto da página, também é utilizado nas estruturas de seleção para unir as setas indicativas de fluxo após a passagem pelo bloco condicional.

Veja ambas as figuras em uso, no exemplo abaixo:



O diagrama de Chapin também possui uma representação específica para a estrutura de seleção. Veja:



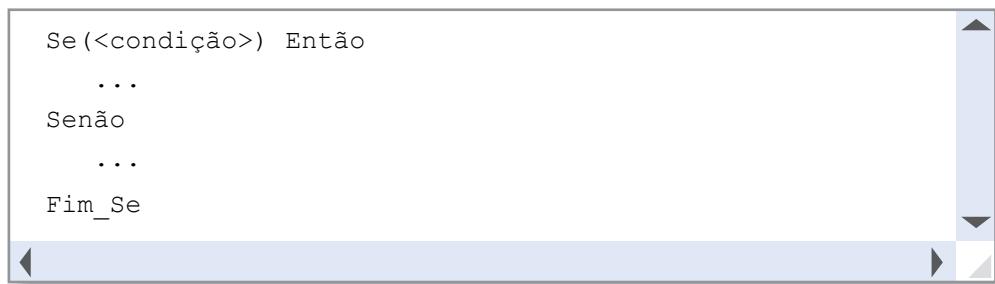
Observe que o bloco referente ao teste condicional está dividido em três partes: uma contém o teste a ser realizado e as outras duas representam os possíveis fluxos.

Logo após o teste, há uma divisão em duas colunas, cada uma representando um fluxo de execução. Quando o trecho do algoritmo relacionado à estrutura de seleção acaba, o fluxo volta a ser representado em apenas uma coluna.

Estrutura de Seleção Composta

A **estrutura de seleção composta** permite definir dois blocos de instruções, sendo que um deles será executado e o outro não, de acordo com o atendimento ou não dos critérios definidos. Esta estrutura também é conhecida como **desvio condicional composto**.

No pseudocódigo, a estrutura de seleção composta é representada pelo comando **Se...Senão**, que utiliza a seguinte estrutura:



Todos os comandos existentes entre **Então** e **Senão** só serão executados se a condição for atendida. E todos os comandos existentes entre **Senão** e **Fim_Se** só serão executados se a condição não for atendida.

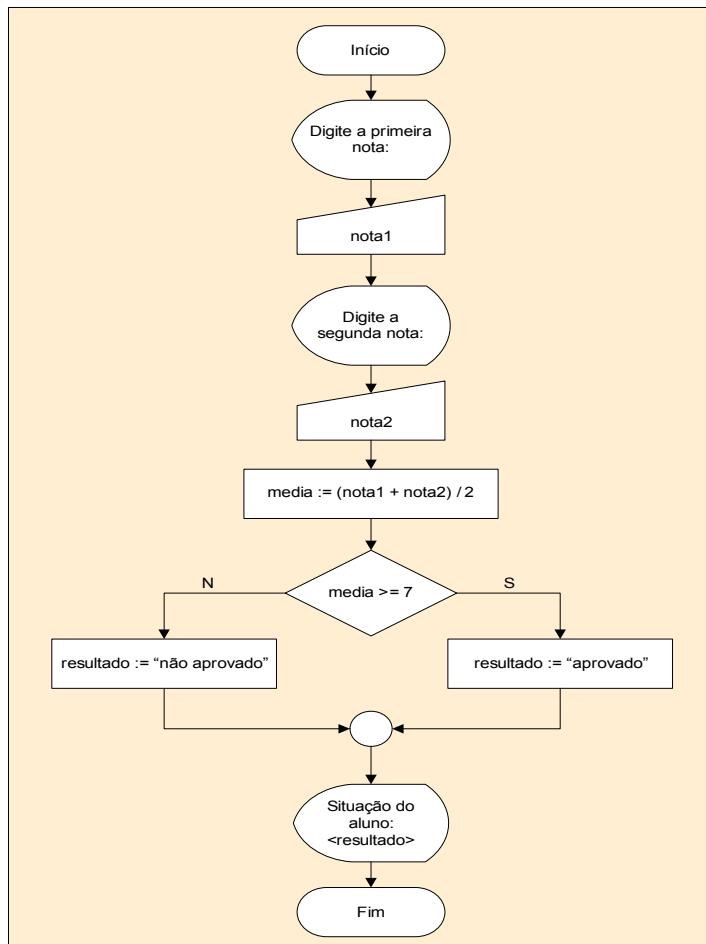
Exemplo de uso:

```
{Exemplo de uso do comando Se...Senão}
Algoritmo ExemploSeSenao
Variáveis
    nota1, nota2, media : Real;
    resultado : Caractere;
Início
01    Escreva("Digite a primeira nota:");
02    Leia(nota1);
03    Escreva("Digite a segunda nota:");
04    Leia(nota2);
05    media := (nota1 + nota2) / 2;

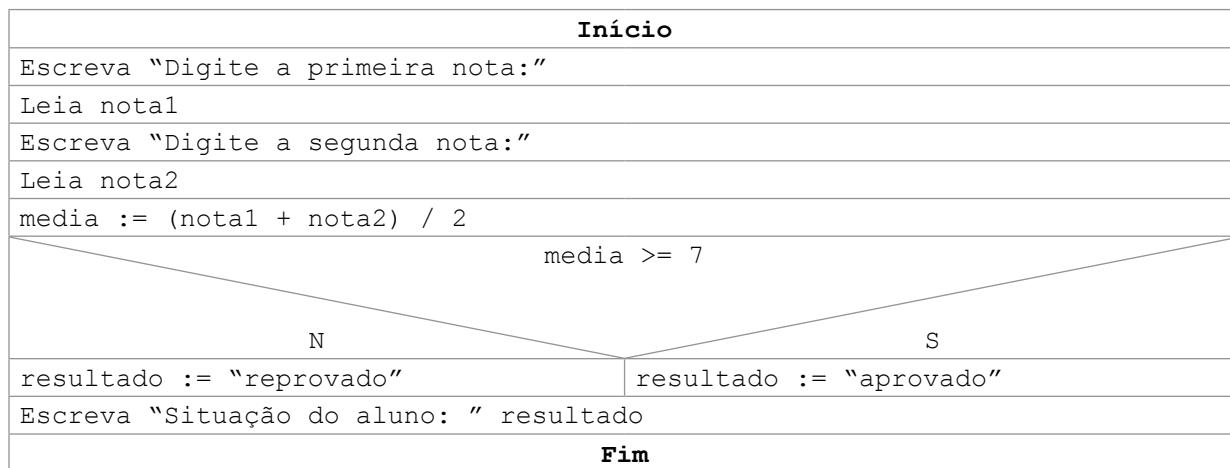
06    Se(media >= 7) Então
07        resultado := "aprovado" ;
08    Senão
09        resultado := "não aprovado";
10    Fim_Se

11    Escreva("Situação do aluno: " + resultado);
Fim
```

Nos fluxogramas, a representação da estrutura de seleção composta é feita com as mesmas figuras que conhecemos anteriormente. Veja o exemplo:



No diagrama de Chapin, a estrutura também é a mesma que já conhecemos, porém com as colunas dos dois fluxos contendo comandos. Veja:



Estruturas de Seleção Aninhadas

Muitas vezes, dentro de um fluxo condicional, será necessário tomar uma nova decisão. Ou pode ser que tenhamos mais de duas opções de fluxo de execução. Em ambos os casos, podemos utilizar **estruturas de seleção aninhadas**, que nada mais são do que uma estrutura de seleção dentro de outra. Veja um exemplo:

42

{Exemplo de uso de Se aninhado}

Algoritmo ExemploSeAninhado

Variáveis

 nota1, nota2, media : Real;

 nome, resultado : Caractere;

Início

01 Escreva("Digite o nome do aluno:");

02 Leia(nome);

03 Escreva("Digite a primeira nota:");

04 Leia(nota1);

05 Escreva("Digite a segunda nota:");

06 Leia(nota2);

07 media := (nota1 + nota2) / 2;

08 Se(media >= 7) Então

09 resultado := "aprovado" ;

10 Senão

11 Se(media >= 4) Então

12 resultado := "em exame";

13 Senão

14 resultado := "reprovado";

15 Fim_Se

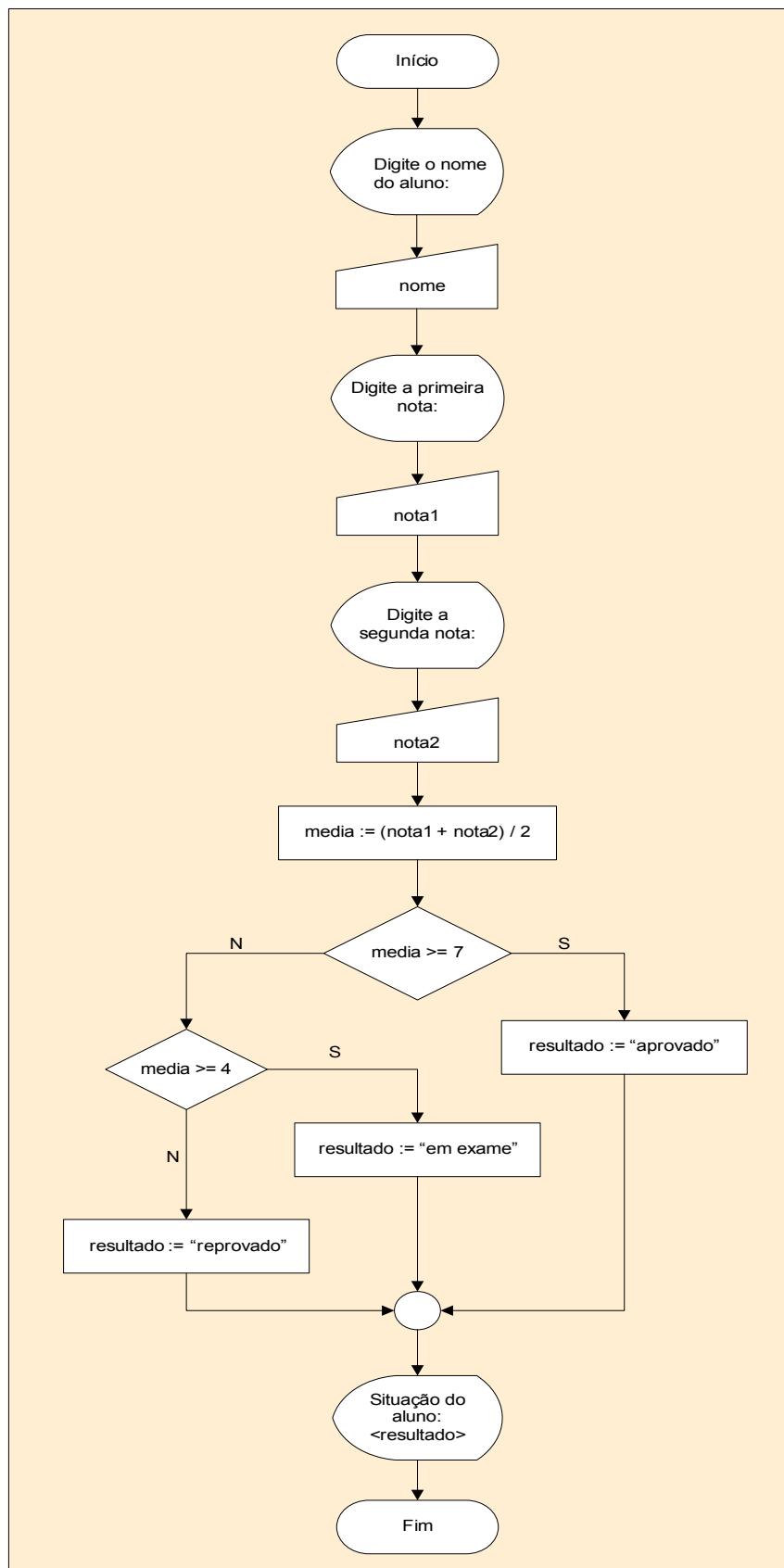
16 Fim_Se

17 Escreva("O aluno " + nome + " está " + resultado);

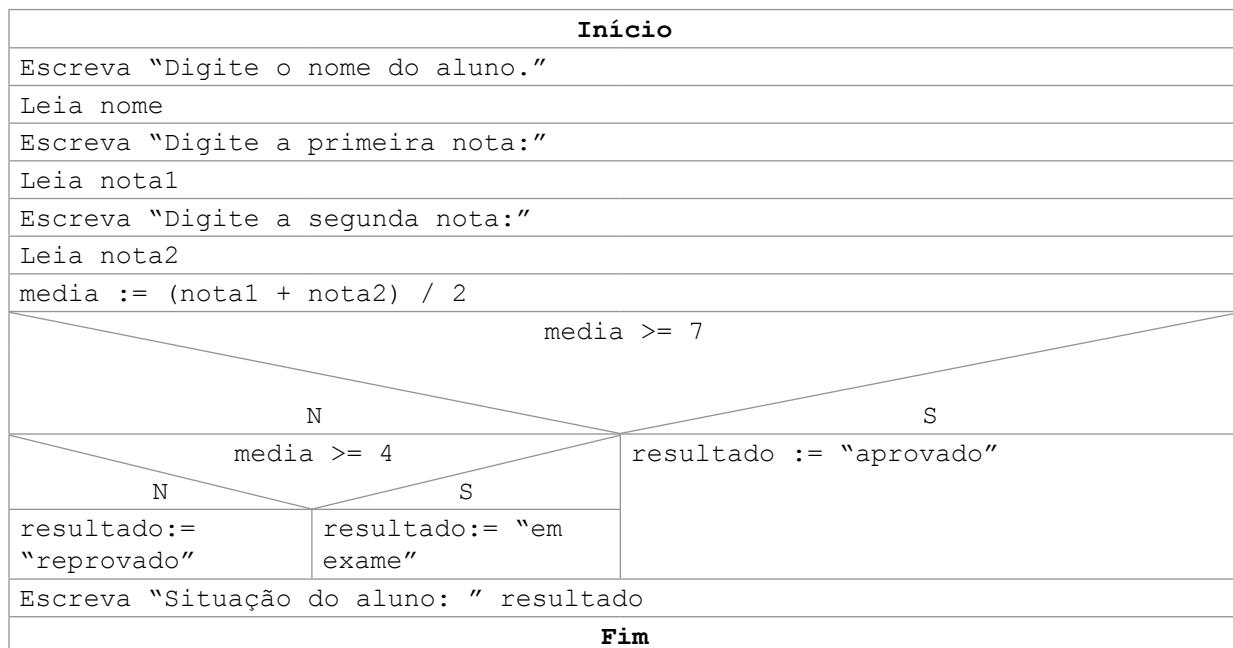
Fim

Perceba que, no exemplo anterior, cada novo nível de instruções recebe um avanço na indentação, a fim de facilitar a leitura e o entendimento do algoritmo.

Veja, a seguir, um exemplo de estrutura de seleção aninhada em um fluxograma:



No diagrama de Chapin, a representação de estruturas aninhadas ocupa bastante espaço, o que dificulta sua adoção para algoritmos maiores. Confira:



Não há um limite para o número de estruturas que podem ser aninhadas em um algoritmo, mas deve-se utilizar o bom senso a fim de se evitar algoritmos excessivamente longos e complexos.

Pode-se dizer que, geralmente, quando as estruturas de seleção de um algoritmo atingem muitos níveis de aninhamento, o algoritmo não está utilizando a melhor opção possível de implementação. Em outras palavras, existe uma maneira mais eficiente de realizar a mesma tarefa.

44

Repetição

O segundo tipo de estrutura de controle que iremos aprender é a **estrutura de repetição**, que permite executar um ou mais passos do algoritmo repetidas vezes. Esta estrutura também é chamada de **estrutura de iteração** ou **estrutura de looping**.

Em alguns casos, nós sabemos com antecedência quantas vezes será preciso executar um bloco de comandos, mas em outros não. As três estruturas de repetição que iremos conhecer podem ser utilizadas nas mais variadas situações, sendo algumas mais propícias para determinados casos.

Repetição com Pré-Teste

A **repetição com Pré-Teste** é uma estrutura de *looping* que repete um bloco de comandos enquanto a expressão avaliada for verdadeira.

No pseudocódigo, este tipo de repetição é representado pelo comando **Enquanto**, que utiliza a seguinte estrutura:

```
Enquanto (<condição>) Faça  
    ...  
Fim_Enquanto
```

Todos os comandos existentes entre **Faça** e **Fim_Enquanto** serão executados repetidamente, até que a condição deixe de ser atendida. Por exemplo:

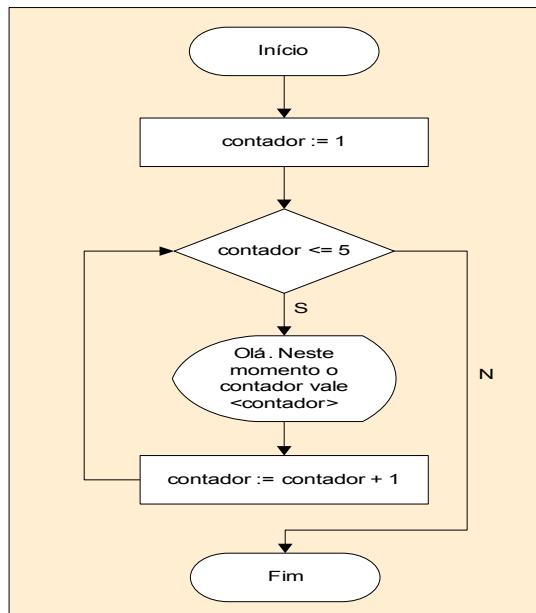
```
{Exemplo de Repetição com Pré-Teste}
Algoritmo ExemploEnquanto
Variáveis
    contador : Inteiro;
Início
01    contador := 1;
02    Enquanto(contador <= 5) Faça
03        Escreva("Olá. Neste momento o contador vale " + contador);
04        contador := contador + 1;
05    Fim_Enquanto
Fim
```

No exemplo acima, adicionamos na linha 2 o comando **Enquanto**, associado à expressão (**contador <= 5**), o que significa que enquanto o contador não for maior do que 5, o bloco de comandos será repetido. Na linha 4, modificamos o valor da variável **contador** em uma operação chamada de incremento, que consiste em atribuir à variável um novo valor baseado no atual, porém acrescido de 1.

Veja como ficaria o teste de mesa para este algoritmo:

Nº Linha	contador	Tela
1	1	
2	1	
3	1	Olá. Neste momento o contador vale 1
4	2	
2	2	
3	2	Olá. Neste momento o contador vale 2
4	3	
2	3	
3	3	Olá. Neste momento o contador vale 3
4	4	
2	4	
3	4	Olá. Neste momento o contador vale 4
4	5	
2	5	
3	5	Olá. Neste momento o contador vale 5
4	6	
2	6	
5	6	

A representação de repetições do tipo Pré-Teste, em fluxogramas, é realizada por meio de figuras já conhecidas por nós. Vejamos:



Na repetição com Pré-Teste, muitas vezes não temos como saber quantas vezes o *loop* será executado. O número de iterações pode ser zero ou inúmeras. O motivo é que a expressão que determina a saída do *loop* pode depender de uma ação do usuário ou, então, de algum outro fator que só será definido ao longo do processamento.

A partir deste ponto, não iremos mais utilizar diagramas de Chapin nos exercícios e exemplos. Vamos nos concentrar no pseudocódigo e no fluxograma que são, nesta ordem, as formas de representação mais utilizadas.

Vamos analisar outro exemplo, agora utilizando repetição com Pré-Teste e Seleção em um mesmo algoritmo:

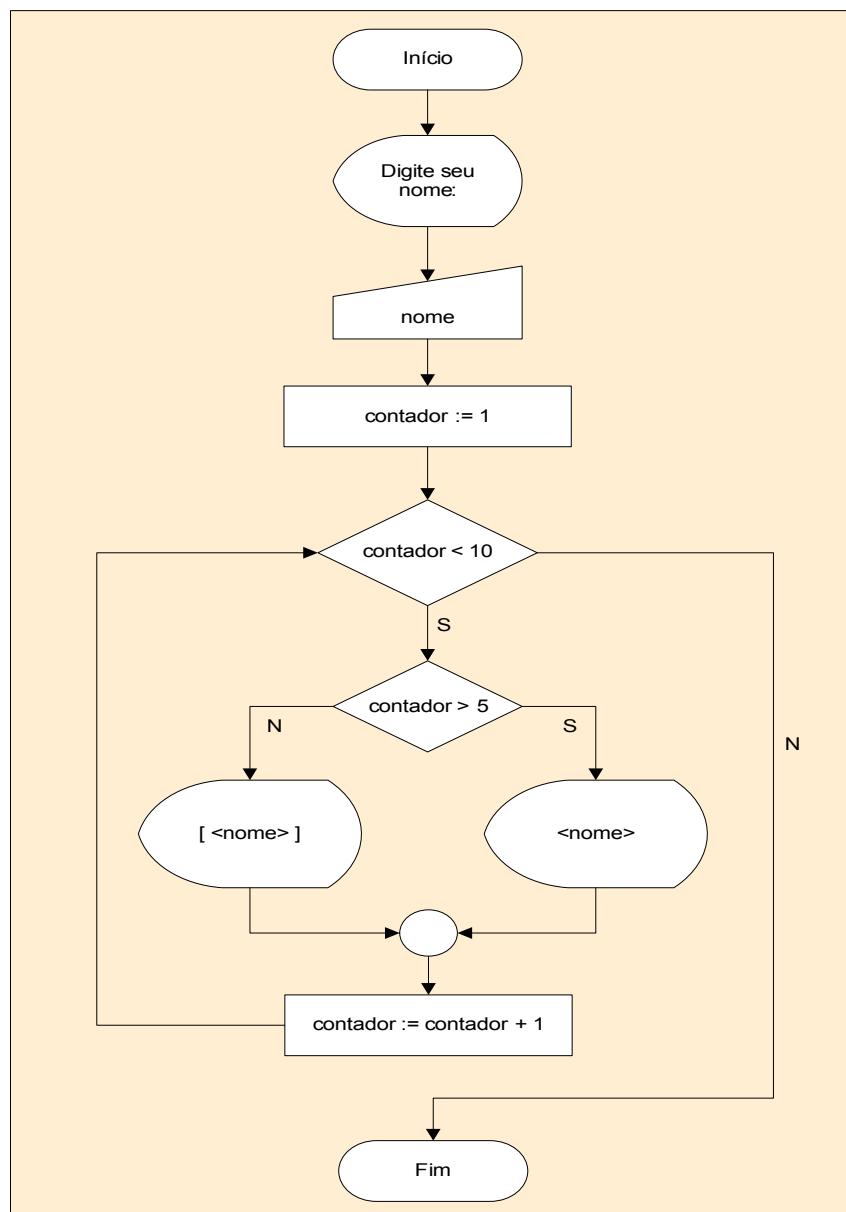
{Exemplo de Repetição Pré-Teste e Seleção}
Algoritmo ExemploEnquanto2
Variáveis
 contador : Inteiro;
 nome : caractere;
Início
01 Escreva("Digite seu nome:");
02 Leia(nome);
03 contador := 1;
04 Enquanto(contador <= 10) Faça

05 SE (contador <= 5) Então
06 Escreva(nome);
07 Senão
08 Escreva("[" + nome + "]");
09 Fim_se

10 contador := contador + 1;
11 Fim_Enquanto
Fim

No exemplo anterior, o algoritmo pergunta um nome e o escreve 10 vezes na tela, porém, nas últimas 5 vezes, o nome é escrito entre colchetes.

O fluxograma deste algoritmo ficaria da seguinte forma:



Repetição com Pós-Teste

A **repetição com Pós-Teste** é similar à repetição com Pré-Teste, porém com duas diferenças cruciais:

1. Na repetição com Pós-Teste, o teste que define a execução ou não do bloco de comandos é realizado ao final do bloco. A consequência disso é que o bloco de instruções é executado ao menos uma vez.

2. Ao contrário da repetição com Pré-Teste, que executa o bloco de comandos enquanto o resultado do teste for verdadeiro, a repetição com Pós-Teste executa o bloco de comandos enquanto o resultado do teste for falso. Ou seja, a repetição com Pós-Teste executa até que o resultado do teste se torne verdadeiro.

No pseudocódigo, a repetição com Pós-Teste é representada pelo comando **Repita...Até_Que**, utilizando a seguinte estrutura:

```
Repita
    ...
Até_Que (<condição>)
```

Todos os comandos existentes entre **Repita** e **Até_Que** serão executados repetidamente, até que a condição seja atendida. Por exemplo:

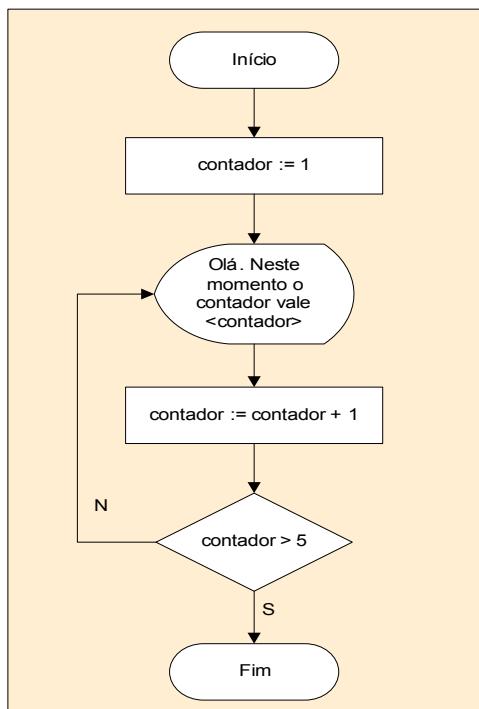
```
{Exemplo de Repetição com Pós-Teste}
Algoritmo ExemploRepita
Variáveis
    contador : Inteiro;
Início
01    contador := 1;
02    Repita
03        Escreva("Olá. Neste momento o contador vale " + contador);
04        contador := contador + 1;
05    Até_Que(contador > 5)
Fim
```

O algoritmo acima possui a mesma funcionalidade do exemplo apresentado para a repetição com Pré-Teste. Aqui, ele foi reescrito para passar a utilizar a repetição do tipo Pós-Teste. Observe que as únicas mudanças aconteceram na estrutura de repetição. Para conseguirmos o mesmo resultado, foi necessário modificar o teste que determina a saída do fluxo. O teste era **contador <= 5**, passou a ser **contador > 5**.

Na repetição com Pós-Teste, em muitos casos, também não é possível saber quantas vezes o *loop* será executado, mas sabemos que será ao menos uma vez. O número de iterações pode ser de uma ou de inúmeras, porque, assim como na repetição com Pré-Teste, a expressão que determina a saída do *loop* pode depender de fatores definidos somente durante o processamento.

Para representar uma repetição do tipo Pós-Teste em um fluxograma, utilizam-se as mesmas figuras usadas no Pré-Teste, porém com o teste que define a saída do *loop* sendo realizado ao final do bloco de instruções.

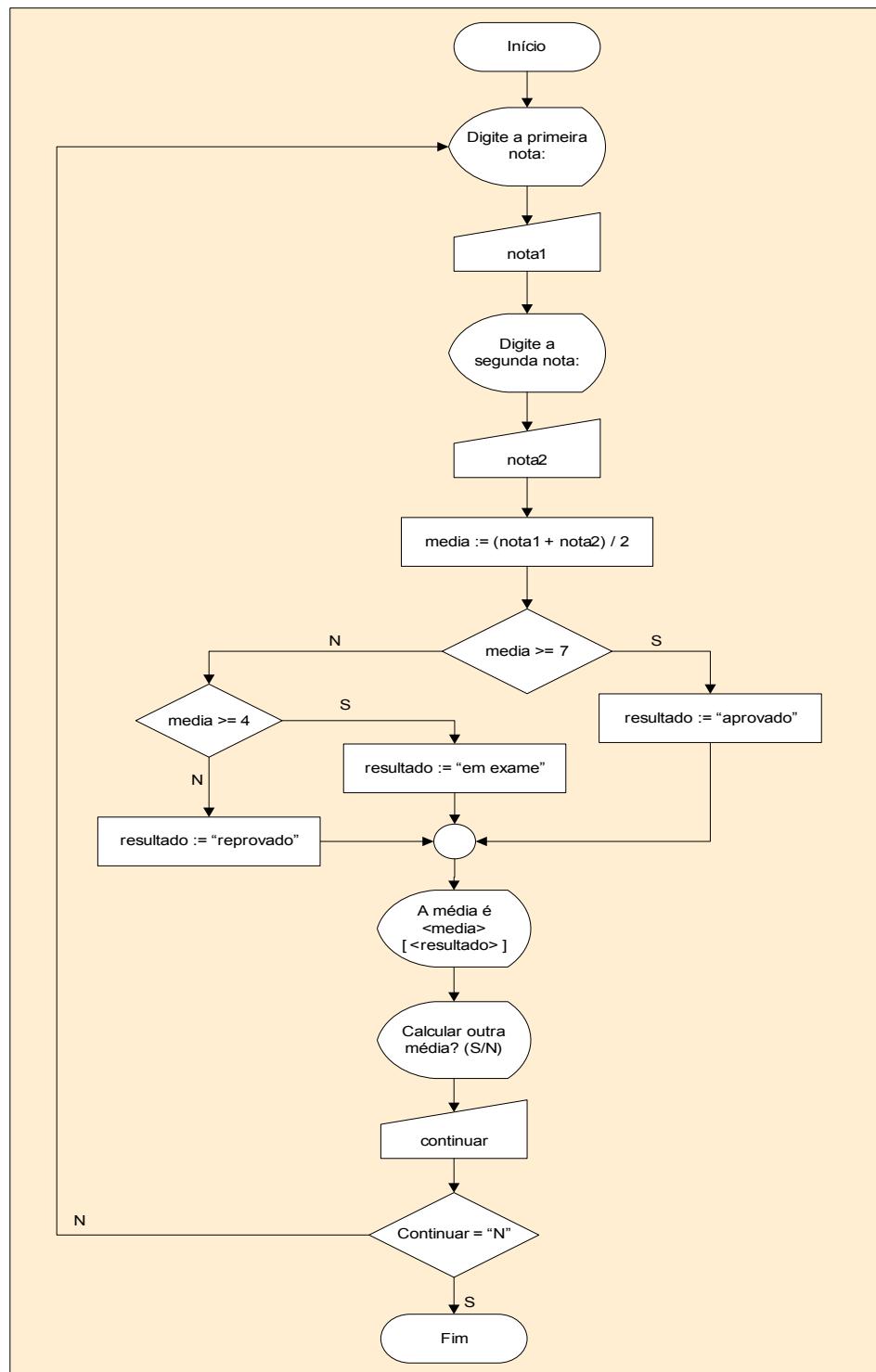
A representação do algoritmo anterior em fluxograma é feita da seguinte forma:



Vejamos um exemplo de repetição com Pós-Teste e seleção no mesmo algoritmo:

{Exemplo de Repetição Pós-Teste e Seleção}
Algoritmo ExemploRepita2
Variáveis
nota1, nota2, media : Real;
resultado, continuar : Caractere;
Início
01 Repita
02 Escreva("Digite a primeira nota:");
03 Leia(nota1);
04 Escreva("Digite a segunda nota:");
05 Leia(nota2);
06 media := (nota1 + nota2) / 2;
07 Se(media >= 7) Então
08 resultado := "aprovado";
09 Senão
10 Se(media >= 4) Então
11 resultado := "em exame";
12 Senão
13 resultado := "reprovado";
14 Fim_Se
15 Fim_Se
16 Escreva("A média é " + media + "[" + resultado + "]");
17 Escreva("Calcular outra média? (S/N)");
18 Leia(continuar);
19 Até_Que(continuar = "N")
Fim

O fluxograma deste algoritmo ficaria da seguinte forma:



Repetição com Variável de Controle

A **repetição com variável de controle** é baseada em uma variável numérica, cujo valor é controlado dentro da própria estrutura de repetição, com base nos critérios definidos na sua criação.

Este tipo de repetição é ideal quando sabemos com antecedência quantas vezes o bloco de instruções deverá ser executado.

No pseudocódigo, a repetição com variável de controle é representada pelo comando **Para**, utilizando a seguinte estrutura:

```
Para <variável_controle> De <val_inicial> Até <val_final> Passo  
<incremento> Faça  
    ...  
Fim_Para
```

Todos os comandos existentes entre **Para** e **Fim_Para** serão executados repetidamente, até que a variável de controle atinja seu valor final. Por exemplo:

```
{Exemplo de Repetição com Variável de Controle}  
Algoritmo ExemploPara  
    Variáveis  
        contador : Inteiro;  
    Início  
        01    Para contador De 1 Até 5 Passo 1 Faça  
        02        Escreva("Olá. Neste momento o contador vale " + contador);  
        03    Fim_Para  
    Fim
```

51

No algoritmo acima, o comando da linha 2 será executado 5 vezes, uma vez que foi definido na linha 1 que a variável **contador** terá seu valor incrementado com 1 unidade de cada vez (passo), iniciando em 1 e indo até 5.

Como se pode perceber, o gerenciamento deste tipo de repetição torna-se mais simples e menos propenso a erros, uma vez que o controle do *loop* é realizado em uma única linha.

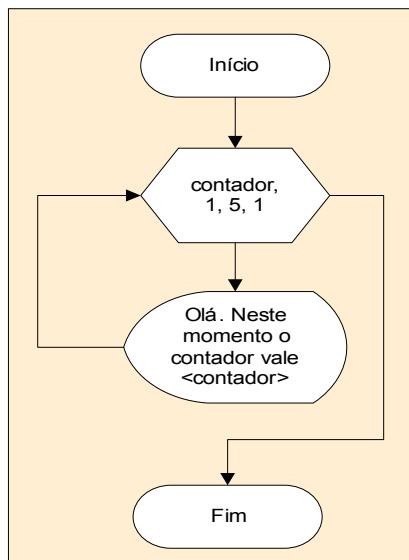
Para representar nos fluxogramas as repetições do tipo variável de controle, utiliza-se a seguinte figura:



Preparação

Representa a preparação de instruções para processamento em estruturas de repetição do tipo Variável de Controle.

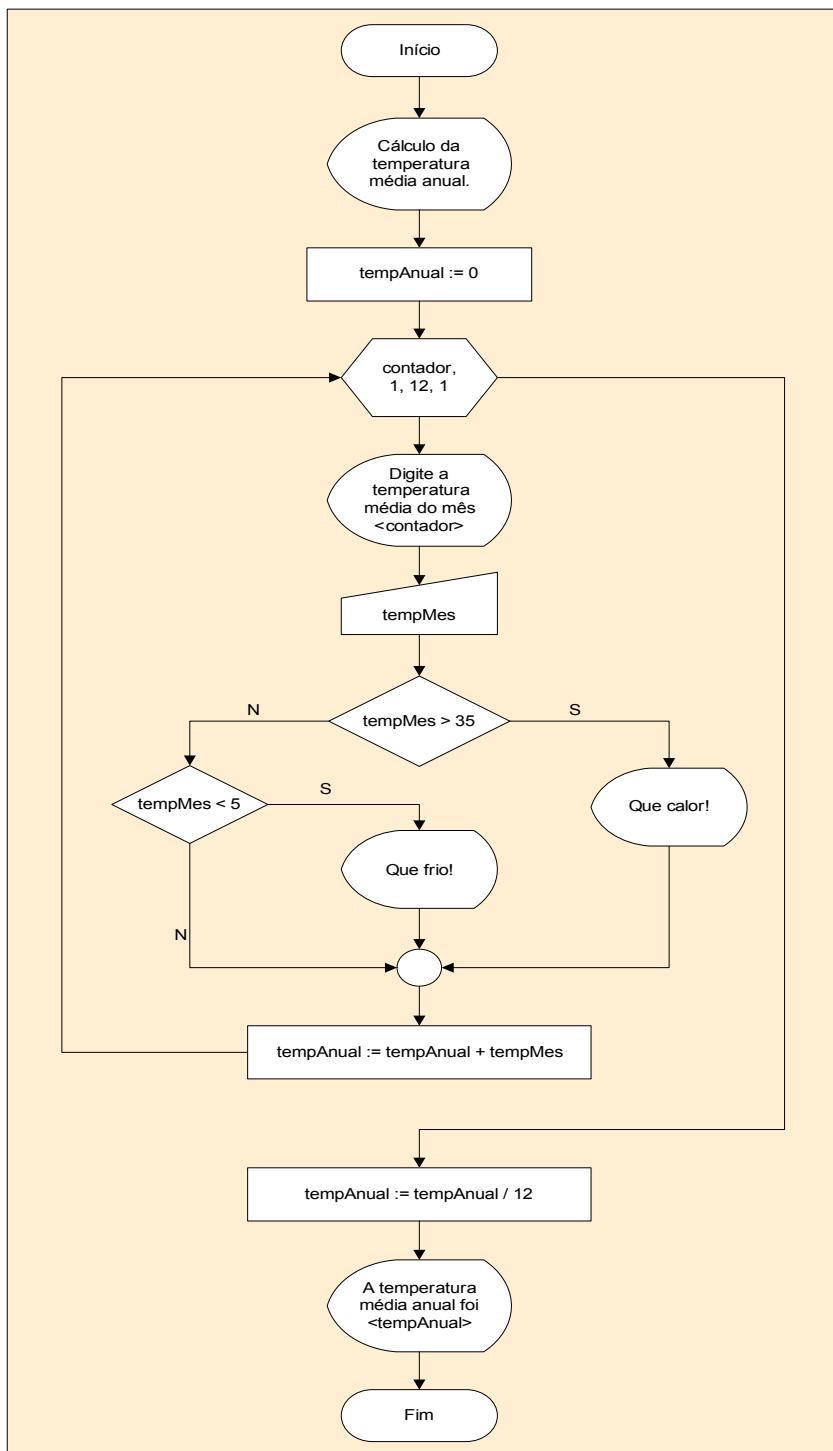
Veja o algoritmo de exemplo, representado em fluxograma:



Vejamos um exemplo de repetição com variável de controle e estrutura de seleção no mesmo algoritmo:

52 {Exemplo de Repetição com Variável de Controle e Seleção}
Algoritmo ExemploPara2
Variáveis
 contador : Inteiro;
 tempMes, tempAnual : Real;
Início
01 Escreva("Cálculo de temperatura média anual.");
02 tempAnual := 0;
03 Para contador De 1 Até 12 Passo 1 Faça
04 Escreva("Digite a temperatura média do mês " + contador);
05 Leia(tempMes);
06 Se(tempMes > 35) Então
07 Escreva("Que calor!");
08 Senão
09 Se(tempMes < 5) Então
10 Escreva("Que frio!");
11 Fim_Se
12 Fim_Se
13
14 tempAnual := tempAnual + tempMes;
15 Fim_Para
16 tempAnual := tempAnual / 12;
17 Escreva("A temperatura média anual foi " + tempAnual);
Fim

O fluxograma do exemplo ficaria assim:



Considerações Sobre os Tipos de Repetição

Pode-se dizer que é possível construir a maioria dos algoritmos utilizando qualquer um dos três tipos de repetição, sendo possível substituir um tipo de repetição por outro, mantendo a mesma funcionalidade, com poucas mudanças no algoritmo. A única exceção é que não se pode utilizar uma repetição do tipo variável de controle para os casos em que o número de iterações é desconhecido.

Atenção: Quando se diz que “o número de iterações é desconhecido” não se está fazendo referência a um valor inserido pelo usuário, pois quando o usuário informa um valor, então o algoritmo sabe quantas vezes terá que executar o bloco de comandos. Não saber quantas vezes o *loop* será executado significa que o número de iterações só será descoberto após o início da execução do *loop*. Por exemplo, imagine que estamos criando um algoritmo que leia arquivos gravados no disco e escreva na tela o conteúdo lido. A cada linha lida de um arquivo, o programa escreve uma linha na tela – isso exige uma estrutura de *looping*. Não sabemos, porém, quantas linhas de informação existirão em cada arquivo a ser processado, pode ser 1 linha ou 500, e esta informação só será conhecida quando chegarmos ao fim do arquivo. Este é um caso típico em que a repetição com variável de controle não pode ser utilizada.

Para os casos em que não sabemos quantas vezes iremos executar o bloco de comandos, devem-se escrever *loops* com Pré e Pós-Teste. Porém, há um alerta: o algoritmo deve ser bem testado para garantir que não ocorrerá um erro de repetição infinita, o chamado **loop infinito** (também conhecido como **loop eterno**), que é uma condição em que a expressão responsável pela saída do *loop* nunca é atendida.

Veja um exemplo de *loop* infinito em uma repetição do tipo Pré-Teste:

54

Estruturas de Controle

```
{Exemplo de loop infinito}
Algoritmo ExemploLoopInfinito
Variáveis
    contador : Inteiro;
Início
01    contador := 1;
02    Enquanto(contador <= 5) Faça
03        Escreva("Olá. Neste momento o contador vale " + contador);
04    Fim_Enquanto
Fim
```

Perceba que no exemplo acima o **contador** não é incrementado, o que faz com que seu valor seja sempre 1 e, consequentemente, que o bloco de comandos seja executado infinitamente.

Pela sua estrutura, a repetição baseada em variável de controle é menos propensa a este tipo de erro, pois as regras de execução são definidas em uma única linha e o controle das repetições é feito pela própria estrutura.

Na prática, a decisão sobre qual tipo de repetição utilizar vai depender em muito das preferências pessoais de quem está criando o algoritmo. Conforme vamos avançando no conteúdo, vai ficando evidente que é extremamente difícil duas pessoas escreverem um algoritmo exatamente igual, pois além da lógica de resolução de problemas ser diferente, existem múltiplas opções de comandos para realizar uma mesma implementação.

Assim como acontece com as estruturas de seleção, as estruturas de repetição também podem trabalhar de forma aninhada, ou seja, é possível inserir uma estrutura de repetição dentro de outra em um mesmo algoritmo.

Veja um exemplo:

```
{Exemplo de loop aninhado}
Algoritmo ExemploLoopAninhado
Variáveis
    numLido, resultado, contador : Inteiro;
Início
01    Escreva("Digite um número para ver sua tabuada
            ou 0 para sair:");
02    Leia(numLido);
03    Enquanto(numLido > 0) Faça

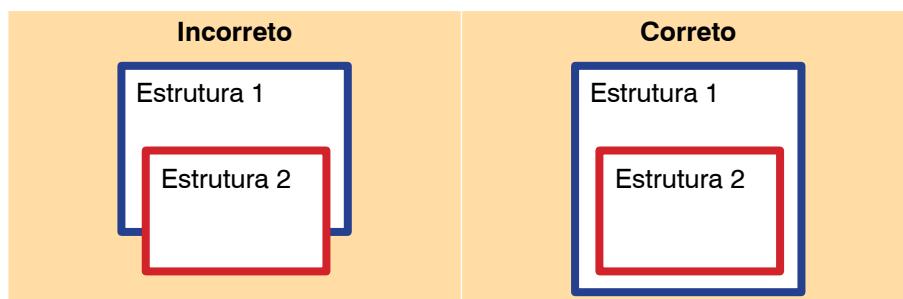
04        Para contador De 1 Até 10 Passo 1 Faça
            resultado := contador * numLido;
            Escreva(contador + " x " + numLido + " = " + resultado);
07        Fim_Para

08        Escreva("Digite um número para ver sua tabuada
            ou 0 para sair:");
09        Leia(numLido);
10        Fim_Enquanto
11    Escreva("Fim do programa.");
Fim
```

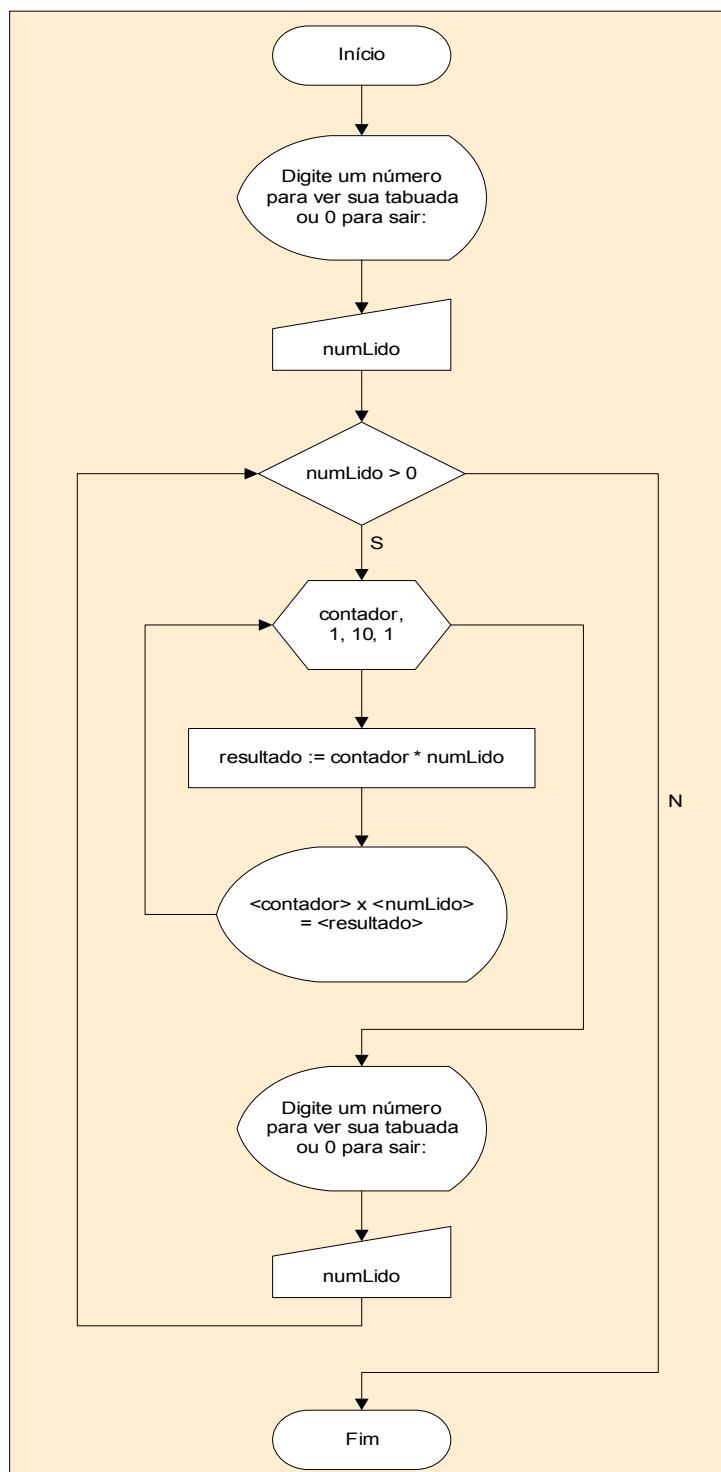
55

No exemplo acima, temos uma repetição com variável de controle inserida dentro de uma repetição com Pré-Teste. Note que quando uma estrutura é inserida dentro de outra, ela deve iniciar e encerrar dentro desta. No nosso exemplo, seria incorreto colocar o **Fim_Para** depois do **Fim_Enquanto**, uma vez que o **Para** foi declarado dentro do **Enquanto**.

O correto é que uma estrutura de repetição ou seleção sempre encerre no mesmo nível em que foi declarada. Veja:



Representado em fluxograma, o algoritmo anterior ficaria da seguinte forma:



56

Estruturas de Controle

Na resolução de problemas mais complexos, é comum termos o aninhamento (ou encadeamento) de estruturas de repetição e estruturas de seleção em um mesmo algoritmo.



Atividades

- 1) Crie um algoritmo que leia dois números informados pelo usuário e, em seguida, exiba na tela uma mensagem dizendo se o maior deles é o primeiro, o segundo, ou se são iguais.
Represente seu algoritmo em pseudocódigo, fluxograma e diagrama de Chapin. Também aplique o teste de mesa.

- 2) Crie um algoritmo que realize as seguintes atividades:
- Solicite ao usuário três valores inteiros.
 - Multiplique o menor valor lido pelo maior e some o resultado com o valor do meio.
 - Imprima na tela o resultado.

Seu algoritmo deverá ser representado em pseudocódigo e fluxograma. É necessário realizar o teste de mesa para garantir o funcionamento do algoritmo.

- 3) Desenvolva um algoritmo em pseudocódigo para aplicar um percentual de desconto sobre o valor de uma compra informado pelo usuário. Os percentuais de desconto são:

- 15% para compras acima de R\$ 500,00;
- 10% para compras entre R\$ 200,00 e R\$ 499,99;
- 5% para compras abaixo de R\$ 200,00.

O algoritmo deverá mandar para a impressora as seguintes informações:

- Valor antes do desconto;
- Valor do desconto;
- Valor a ser pago.

- 4) Crie um algoritmo (pseudocódigo e fluxograma) que leia um valor inteiro para X e escreva na tela X^3 . O algoritmo deve continuar pedindo o valor de X até que o usuário informe 0 (zero), então o programa encerra.

O algoritmo deve ser criado utilizando a estrutura de repetição com Pré-Teste.

- 5) Desenvolva um algoritmo (pseudocódigo e fluxograma) capaz de apresentar na tela o fatorial de um número inteiro informado pelo usuário.

O algoritmo deve ser criado utilizando a estrutura de repetição com Pré-Teste.

- 6) Crie um algoritmo (pseudocódigo e fluxograma) que solicite ao usuário um nome e um número inteiro, que representará a quantidade de vezes que o nome informado deverá ser escrito na tela.

O algoritmo deve ser criado utilizando a estrutura de repetição com Pós-Teste.

- 7) Construa um algoritmo (pseudocódigo e fluxograma) que seja capaz de calcular o valor total de uma compra, somando o preço de cada um dos produtos.

O algoritmo deverá solicitar o preço de cada produto e ir somando ao montante total, e deve entender que os produtos acabaram quando o preço informado for 0 (zero), então mostrará o número de itens comprados e o total da compra, encerrando a execução.

Caso seja informado algum valor menor do que zero, o programa deve desconsiderá-lo e exibir uma mensagem de erro solicitando que o valor correto do produto seja digitado.

O algoritmo deve ser criado utilizando a estrutura de repetição com Pós-Teste.

- 8) Crie um algoritmo (pseudocódigo e fluxograma) que realize as seguintes atividades:

- Pergunte a quantidade de alunos da turma.
- Solicite ao usuário o nome de cada um dos X alunos.
- Envie cada nome lido para a impressora.

O algoritmo deve ser criado utilizando a estrutura de repetição variável de controle.

- 9) Desenvolva um algoritmo (pseudocódigo e fluxograma) que solicite ao usuário a entrada de 5 valores inteiros e, a cada valor lido, aplique a seguinte regra: se o número lido for maior que 10, subtraí 5 e escreve o resultado na tela, se não soma 2 e manda o resultado para a impressora.

O algoritmo deve ser criado utilizando-se a estrutura de repetição variável de controle.

Estruturas de Dados Homogêneas

Vetores

Os **vetores**, também chamados de matrizes unidimensionais, são estruturas de dados que permitem armazenar várias informações de um mesmo tipo de dado sob o mesmo rótulo. Em outras palavras, um vetor é composto por um conjunto de elementos do mesmo tipo, que compartilham um nome comum. Cada elemento é como uma variável: pode receber atribuição de valores, ser utilizado em cálculos, etc.

Imagine que temos que armazenar as médias de uma turma com 35 alunos. Com o que vimos até agora, seria necessário criarmos 35 variáveis, uma para a média de cada aluno. Ao invés disso, podemos criar um vetor e armazenar todas as notas nele.

Do ponto de vista dos dados armazenados, um vetor é como uma tabela com uma única linha e várias colunas, assim:

Brasil	Paraguai	Argentina	Uruguai	Chile
--------	----------	-----------	---------	-------

Para entendermos melhor a estrutura de um vetor, vamos representá-lo da seguinte forma:

Índice	1	2	3	4	5
Elemento	Brasil	Paraguai	Argentina	Uruguai	Chile

O índice nada mais é do que uma forma de fazer referência a cada um dos elementos do vetor, ou seja, é uma maneira de indicar a posição do elemento que estamos querendo utilizar naquele momento. No vetor acima, o elemento da posição 1 guarda o valor Brasil, o da posição 2 guarda o valor Paraguai, e assim por diante.

O trabalho com vetores facilita a declaração e o gerenciamento dos dados, pois ao invés de precisar declarar uma variável para cada valor a ser armazenado, declara-se apenas o vetor informando a quantidade de valores que ele será capaz de armazenar.

Outra vantagem dos vetores é que os algoritmos criados para atribuir e ler dados de vetores geralmente são mais compactos do que os que trabalham com variáveis. Além disso, tarefas como ordenação e pesquisa de valores são atividades realizadas de forma mais eficiente por meio de vetores.

Mas atenção! O vetor é útil quando precisamos manipular uma lista de dados de mesmo significado. Vetores não devem ser usados para armazenar informações de significados diferentes, ainda que estas informações sejam do mesmo tipo de dado. Ex.: nota, idade e nº do aluno na turma, mesmo sendo todos do tipo Inteiro, não devem ser armazenados juntos em um vetor, pois são informações de significados totalmente diferentes. Esta é uma prática que ajuda a manter a organização e a consistência dos dados manipulados pelo algoritmo.

Declaração

Conforme já foi citado, todos os elementos de um vetor pertencem necessariamente ao mesmo tipo de dado. Esta é a essência das estruturas de dados homogêneas.

A declaração de vetores em pseudocódigo é realizada utilizando-se a seguinte sintaxe:

```
<nomeVetor> : Vetor[1..<tamanho>] De <tipo_de_dado>
```

60

Exemplo:

```
{Declaração de vetor}
Variáveis
    valores : Vetor[1..10] De Inteiro;
    locais : Vetor[1..5] de Caractere;
```

Tenha em mente que uma vez que um vetor seja declarado não é possível modificar seu tamanho. Contudo, em termos de aproveitamento de memória não é uma boa prática criar um vetor com posições que você não pretende utilizar, pois quando declaramos um vetor, o computador aloca (reserva) a quantidade de memória necessária para armazenar todos os elementos do vetor.

O ideal é só declarar um vetor quando souber a quantidade de itens que se deseja armazenar nele, porém, em alguns casos, é inevitável declararmos um vetor com um tamanho que não será totalmente utilizado. Por exemplo, a quantidade de alunos em uma turma pode variar bastante, então a solução é declarar o vetor com o tamanho máximo que uma turma pode ter, mesmo sabendo que para a maioria das turmas haverá desperdício de espaço.

A nomenclatura utilizada para os vetores segue os mesmos padrões utilizados para as variáveis.

Atribuição e Leitura

Para acessar (atribuir ou ler) um determinado elemento dentro do vetor, é necessário informar sua posição, também chamada de índice, por meio da seguinte sintaxe:

```
{para atribuir}  
<nomeVetor>[<índice>] := <valor>;  
  
{para ler}  
<nomeVariavel> := <nomeVetor>[<índice>];
```

As operações de atribuição e leitura podem ser realizadas diretamente em cada posição do vetor, mas isto se torna desnecessariamente trabalhoso. Veja:

```
{Exemplo de vetor}  
Algoritmo ExemploVetor  
Variáveis  
    valores : Vetor[1..10] De Inteiro;  
Início  
    Escreva("Digite um valor para a posição 1:");  
    Leia(valores[1]);  
    Escreva("Digite um valor para a posição 2:");  
    Leia(valores[2]);  
    {...até o último elemento}  
  
    Escreva("O elemento da posição 1 vale " + valores[1]);  
    Escreva("O elemento da posição 2 vale " + valores[2]);  
    {...até o último elemento}  
Fim
```

Se formos acessar todos os itens de um vetor da forma como está no exemplo acima, não há vantagem alguma em relação ao uso de variáveis. O acesso direto é uma alternativa apenas para quando desejamos acessar poucos elementos do vetor.

Para percorrer um vetor, o melhor é utilizarmos uma estrutura de repetição, e a opção mais interessante é a estrutura de repetição com variável de controle (o **Para**), pois, neste caso, a quantidade de iterações é conhecida com antecedência.

Vejamos um exemplo em pseudocódigo, ilustrando o uso de vetores:

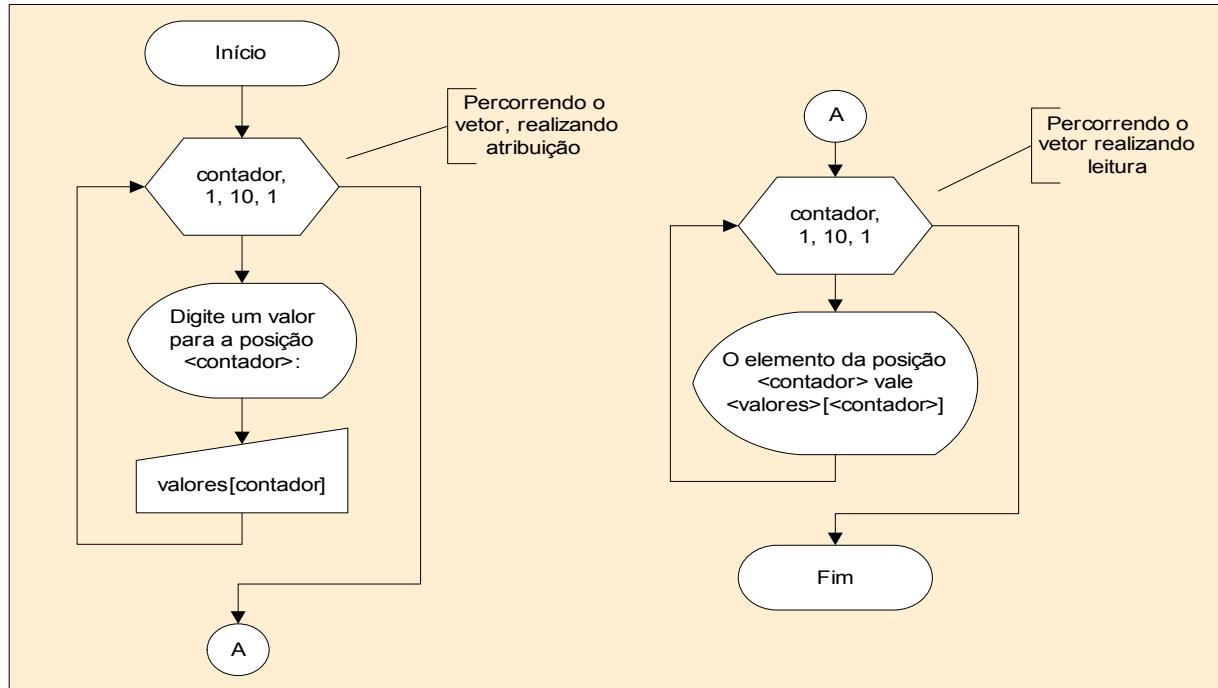
```
{Exemplo de vetor}
Algoritmo ExemploVetor
Variáveis
    valores : Vetor[1..10] De Inteiro;
    contador : Inteiro;
Início
01    Para contador De 1 Até 10 Passo 1 Faça
02        Escreva("Digite um valor para a posição " + contador + ":" );
03        Leia(valores[contador]);
04    Fim_Para

05    Para contador De 1 Até 10 Passo 1 Faça
06        Escreva("O elemento da posição " + contador +
07            " vale " + valores[contador])
08    Fim_Para
Fim
```

Veja o fluxograma do exemplo:

62

Estruturas de Dados Homogêneas



Vejamos um exemplo um pouco mais detalhado, somente em pseudocódigo:

```
{Exemplo de vetor 2}
Algoritmo ExemploVetor2
    Variáveis
        vetA, vetB : Vetor[1..5] De Inteiro;
        vetC : Vetor[1..10] De Inteiro;
        contador, complemento : Inteiro;
    Início
01        Escreva("Populando o primeiro vetor");
02        Para contador De 1 Até 5 Passo 1 Faça
            Escreva("Digite um valor para a posição " + contador + ":");
            Leia(vetA[contador]);
05        Fim_Para

06        Escreva("Populando o segundo vetor automaticamente");
07        Para contador De 1 Até 5 Passo 1 Faça
            vetB[contador] := vetA[contador] * 2;
09        Fim_Para

10        complemento := 0;
11        Para contador De 1 Até 5 Passo 1 Faça
12            vetC[contador + complemento] := vetA[contador];
13            complemento := complemento + 1;
14            vetC[contador + complemento] := vetB[contador];
15        Fim_Para
    Fim
```



Exemplo:

```
{O compartilhamento de contador é incorreto}
Algoritmo ExemploCompartilharContador
Variáveis
    vetA, vetB : Vetor[1..5] De Inteiro;
    contador : Inteiro;
Início
01    Para contador De 1 Até 10 Passo 1 Faça
02        Escreva("Digite um valor para a posição " + contador + ":");
03        Leia(vetA[contador]);

04    Para contador De 1 Até 3 Passo 1 Faça
05        Escreva("Compartilhar contador não dá certo.");
06        Fim_Para
07    Fim_Para
Fim
```

No exemplo anterior, sempre que a estrutura de repetição mais interna encerrar seu *loop*, o contador estará valendo 3; o que fará com que a estrutura externa entre em *loop* infinito, pois o contador nunca chegará ao valor 10, que é o critério de saída desta estrutura. Dependendo dos critérios estabelecidos em cada estrutura, o compartilhamento de variáveis de controle também pode causar outros tipos de problemas.

Para evitar o problema do exemplo, deveriam ter sido declaradas duas variáveis de controle, uma para cada estrutura de repetição. Esta regra vale independentemente do tipo de repetição que estivermos utilizando: pode ser uma repetição com Pós-Teste, dentro de uma com variável de controle; uma com Pré-Teste, dentro de uma com Pós-Teste, etc. Lembre-se de nunca compartilhar variáveis de controle entre estruturas de repetição aninhadas.

64

Matrizes

Matrizes são estruturas de dados que seguem o mesmo princípio dos vetores, porém as matrizes possuem duas ou mais dimensões, ao contrário dos vetores que possuem apenas uma dimensão.

Matrizes Bi e Tridimensionais

Matrizes Bidimensionais

As matrizes bidimensionais possuem duas ou mais linhas e duas ou mais colunas, e costumam ser representadas por meio de tabelas:

11	23	36
7	31	54
13	66	91
44	80	32

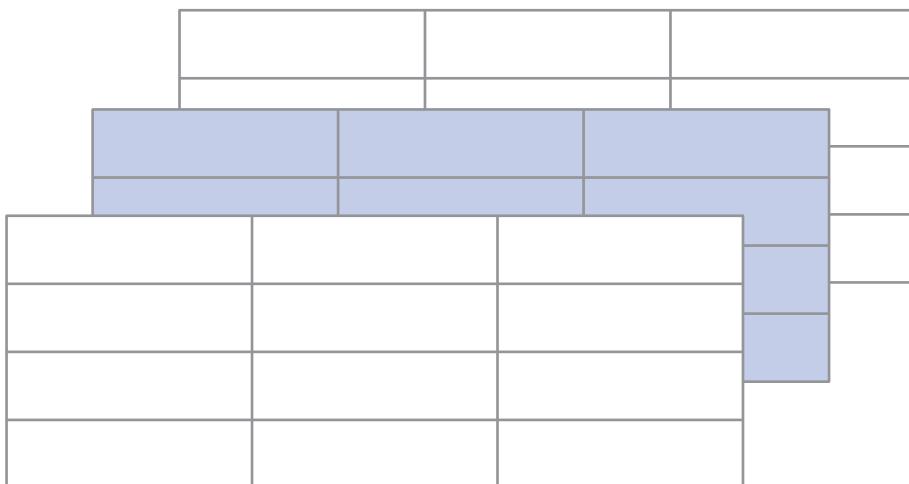
Para entendermos melhor a estrutura das matrizes bidimensionais, vamos representá-las da seguinte forma:

		Índices de coluna		
		1	2	3
Índices de linha	1	11	23	36
	2	37	31	54
	3	13	16	91
	4	44	80	32
	5	61	14	29

A matriz acima possui 15 elementos, distribuídos em 5 linhas e 3 colunas. O elemento da posição linha 1 coluna 1 possui o valor 11; enquanto que o elemento da linha 3, coluna 2, vale 16.

Matrizes Tridimensionais

Uma matriz tridimensional possui largura, altura e profundidade. A estrutura de linhas e colunas é repetida diversas vezes, como se fossem fatias ou camadas. Exemplo:



Cada elemento de uma matriz tridimensional é referenciado por meio da sua posição dada por linha, coluna e profundidade.

Torna-se difícil a representação e, até mesmo, a concepção visual de matrizes com mais de três dimensões, mas, teoricamente, elas também podem ser utilizadas, apesar de tornarem os algoritmos demasiadamente complexos.

Na prática, as matrizes bidimensionais são as mais utilizadas e, em poucos casos, as tridimensionais. Por isso, priorizaremos as matrizes bidimensionais para exemplos e exercícios ao longo do livro.

Declaração

Assim como os vetores, as matrizes também só podem armazenar elementos do mesmo tipo de dado. Além disso, também não é possível modificar o tamanho de uma matriz após a sua declaração.

A declaração de matrizes em pseudocódigo é realizada utilizando-se a seguinte sintaxe:

```
<nomeMatriz> : Matriz[1..<qtdeLinhas>, 1..<qtdeColunas>] De <tipo_de_dado>
```

Exemplo:

```
{Declaração de matriz}
Variáveis
    gradeValores := Matriz[1..10, 1..5] De Real;
    itens : Matriz[1..5, 1..3] De Caractere;
```

Atribuição e Leitura

Para realizar as operações de atribuição e leitura de valor em elementos de uma matriz, é necessário informar a posição do elemento por meio da seguinte sintaxe:

```
{para atribuir}
<nomeMatriz>[<linha>, <coluna>] := <valor>;
{para ler}
<nomeVariavel> := <nomeMatriz>[<linha>, <coluna>];
```

Lembre-se de que para fazer referência a um elemento de uma matriz, é necessário informar sempre o número da linha antes do número da coluna.

Vejamos um exemplo em pseudocódigo, ilustrando o uso de matrizes:

```
{Exemplo de matriz}
Algoritmo ExemploMatriz
Variáveis
    gradeValores : Matriz[1..10, 1..5] De Inteiro;
    i, j : Inteiro;
Início
    Para i De 1 Até 10 Passo 1 Faça
        Escreva("Populando a linha " + i);
        Para j De 1 Até 5 Passo 1 Faça
            Escreva("Digite um valor para a coluna " + j + ":");
            Leia(gradeValores[i, j]);
        Fim_Para
    Fim_Para
```

```

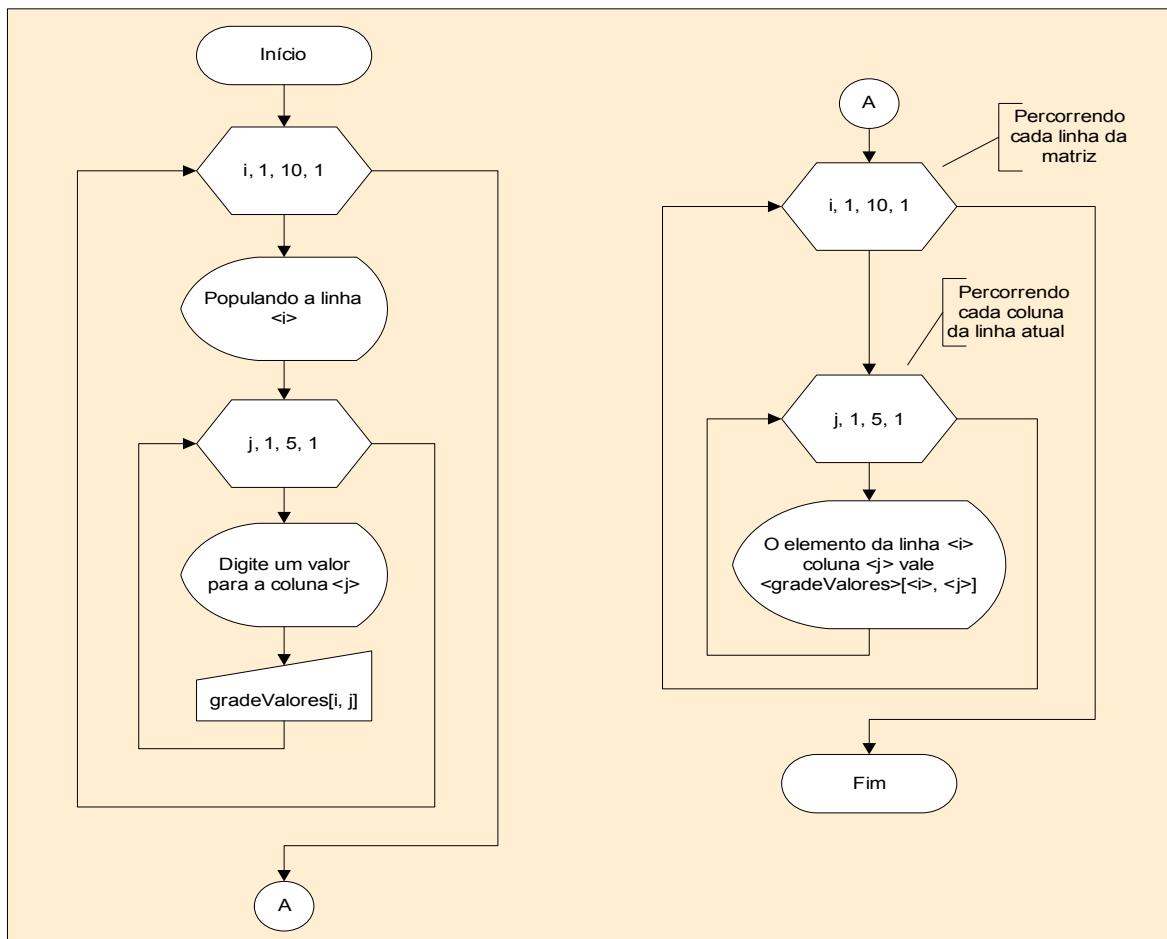
08     Para i De 1 Até 10 Passo 1 Faça
09         Para j De 1 Até 5 Passo 1 Faça
10             Escreva("O elemento da linha " + i + " coluna " + j +
11                 " vale " + gradeValores[i, j])
12         Fim_Para
13     Fim_Para
14 Fim

```

Os comandos entre as linhas 1 e 7 têm a função de popular a matriz. Repare que, para isso, foi necessário utilizar duas estruturas de repetição, uma para percorrer as linhas e outra para percorrer as colunas da matriz. Assim, quando o *loop* externo estiver na linha 1, o *loop* interno fará referência às colunas de 1 a 5, da linha 1; quando o *loop* externo estiver na linha 2, o *loop* interno fará referência às colunas de 1 a 5, da linha 2; e assim sucessivamente, até que toda a matriz seja preenchida.

Entre as linhas 8 e 12, a matriz é percorrida a fim de escrever na tela o valor armazenado em cada uma de suas posições.

Veja o algoritmo representado em um fluxograma:



O exemplo acima é bastante simples. Ele apenas popula uma matriz com informações solicitadas ao usuário e, em seguida, escreve na tela os valores armazenados na matriz.

Vejamos agora um exemplo um pouco mais complexo:

```
{Exemplo de matriz 2}
Algoritmo ExemploMatriz2
Variáveis
    nomeDespesas : Vetor[1..4] de Caractere;
    valorDespesas : Matriz[1..12, 1..4] De Real;
    i, j, ano : Inteiro;
    somatorio, totalAnual : Real;
Início
01    nomeDespesas[1] := "Aluguel";
02    nomeDespesas[2] := "Condomínio";
03    nomeDespesas[3] := "Energia Elétrica";
04    nomeDespesas[4] := "Telefone e Internet";

05    Escreva("Cálculo de despesas anuais.");
06    Escreva("Informe o ano:");
07    Leia(ano);
08    Para i De 1 Até 12 Passo 1 Faça
09        Para j De 1 Até 4 Passo 1 Faça
10            Escreva("Informe o gasto com " + nomeDespesas[j] +
11                " no mês " + i);
12            Leia(valorDespesas[i, j]);
13            Fim_Para
14            Fim_Para

15    Imprima("Relatório de gastos com moradia no ano de " + ano);
16    totalAnual := 0;
17    Para i De 1 Até 4 Passo 1 Faça
18        somatorio := 0;
19        Para j De 1 Até 12 Passo 1 Faça
20            somatorio := somatorio + valorDespesas[j, i];
21            Fim_Para
22            Imprima("O total gasto com " + nomeDespesas[i] +
23                " ao longo do ano foi " + somatorio);
24            totalAnual := totalAnual + somatorio;
25            Fim_Para
26            Imprima("O total gasto no ano foi " + totalAnual);
27            Imprima("A média mensal de gastos foi " + (totalAnual / 12));
28            Fim
```

68

O algoritmo acima tem o objetivo de calcular os gastos com moradia em um determinado ano. Primeiro ele popula um vetor com o nome dos 4 tipos de despesas que irá calcular (linhas 1 a 4). Isto é necessário para identificar cada uma das despesas que serão solicitadas ao usuário.

Entre as linhas 8 e 13, percorre-se a matriz linha a linha; e para cada linha, coluna a coluna. Deste modo, é possível solicitar ao usuário os valores gastos em cada mês com cada tipo de despesa.

Entre as linhas 16 e 23, percorre-se a matriz coluna a coluna; e para cada coluna, linha a linha. Assim é possível somar todos os valores de cada despesa ao longo do ano.

Lembre que esta é apenas uma das várias implementações possíveis para resolver este problema.



Atividades

- 1) Crie um algoritmo em pseudocódigo que solicite ao usuário 10 valores inteiros, armazenando os dados em um vetor. Em seguida, o algoritmo deverá percorrer o vetor escrevendo na tela os valores armazenados nas posições ímpares e mandando para a impressora os valores armazenados nas posições pares. Aplique o teste de mesa.
- 2) Escreva um algoritmo em pseudocódigo que solicite ao usuário 5 valores inteiros e em seguida escreva na tela os valores lidos em ordem inversa (o último valor lido é o primeiro a ser escrito na tela, e assim por diante). Utilize vetor na sua implementação. Você deverá criar 3 versões do algoritmo, cada uma delas utilizando uma das estruturas de repetição estudadas. Aplique o teste de mesa.
- 3) Crie um algoritmo (pseudocódigo) que solicite ao usuário valores reais a fim de popular uma matriz de 3 linhas e 7 colunas. Em seguida os valores lidos devem ser enviados para a impressora, juntamente com a indicação da posição em que ocupavam na matriz.
- 4) Considerando uma turma com 30 alunos, crie um algoritmo em pseudocódigo para calcular a média de todos os alunos, sabendo que cada aluno possui duas notas. Ao final, envie para a impressora os nomes dos alunos com suas respectivas médias.
- 5) Desenvolva um algoritmo capaz de realizar as seguintes atividades:
 - a. Armazenar o nome de 100 produtos informados pelo usuário em um vetor;
 - b. Considerando que existem 3 lojas, solicitar ao usuário o estoque atual de cada produto em cada uma das lojas, armazenando estes dados em uma matriz;
 - c. Gerar o estoque consolidado de cada produto na rede de lojas, armazenando em outra coluna da matriz;
 - d. Considerando que o estoque mínimo de cada produto na rede de lojas deve ser de 30 unidades, enviar para a impressora um relatório com o nome dos produtos que necessitam ser comprados e o estoque atual destes produtos.

Pense na forma mais eficiente de resolver o problema.

Estruturas de Dados Homogêneas: Ordenação e Pesquisa

Com base no que já estudamos, pode-se dizer que a utilização de vetores e matrizes facilita a criação de programas capazes de manipular maiores volumes de dados.

Em muitos casos, porém, os dados precisam ser ordenados antes de serem exibidos ao usuário ou, então, utilizados em algum outro processamento. Em outros casos, será necessário procurar um determinado elemento dentro de um vetor, pois não sabemos sua posição e nem mesmo se ele se encontra armazenado lá. São justamente estas duas demandas que buscamos atender neste capítulo.

Se fosse preciso armazenar uma lista de nomes, você iria utilizar um vetor ou uma matriz? Faz muito mais sentido armazenar em um vetor, certo? É por este motivo que é muito mais frequente termos a necessidade de ordenar e pesquisar os vetores em vez das matrizes. E, partindo dessa premissa, iremos abordar a ordenação e pesquisa do ponto de vista dos vetores. Contudo, também é possível ordenar e pesquisar matrizes, utilizando as técnicas que iremos conhecer, com algumas mudanças nos algoritmos.

Ordenação de Vetores

A ordenação dos elementos de um vetor pode ser realizada independentemente do tipo de dado que o vetor esteja armazenando. Geralmente, a ordenação é feita em ordem crescente, ou seja, com o menor elemento ocupando a primeira posição do vetor e o maior elemento ocupando a última. Depois, caso seja necessário acessar a lista de dados em ordem decrescente, basta percorrer o vetor de trás para frente.

Existem várias técnicas de ordenação de vetores, porém iremos utilizar uma técnica básica, pois a ideia é termos um primeiro contato com este tipo de algoritmo e praticarmos a mudança de posição de elementos em um vetor. Oportunamente você terá contato com técnicas de ordenação mais avançadas, a maioria utilizando recursos que não estudamos até o momento.

Bubble Sort

O método de ordenação **Bubble Sort**, também conhecido como método bolha, é um dos mais simples de todos. Ele realiza a chamada ordenação por flutuação.

Seu funcionamento é bastante simples, consiste em percorrer o vetor várias vezes trazendo para a posição de referência o menor elemento encontrado. Na primeira execução, a posição de referência é a primeira posição do vetor. A cada vez que o vetor é percorrido, a posição de referência avança uma casa. Quando a posição de referência chegar ao final do vetor, este estará ordenado.

Imagine que desejamos ordenar o seguinte vetor:

Brasil	Paraguai	Argentina	Uruguai	Chile
--------	----------	-----------	---------	-------

A seguir, veremos um passo a passo para ordená-lo, utilizando o método *Bubble Sort*. Vamos representá-lo com índice para facilitar a compreensão.

O primeiro passo é entendermos a função de duas peças importantes:

- **Índice de Referência (IR):** Como é necessário percorrer todo o vetor, realizando comparações, o índice de referência é um marco que indica que todos os itens à sua esquerda já estão ordenados. Ele também indica o local onde deve ser armazenado o menor valor encontrado durante a iteração atual. Na primeira iteração, o índice de referência é a primeira posição do vetor.
- **Índice Atual (IA):** Indica a posição atual que deve ser utilizada na comparação com o elemento do índice de referência. Todos os itens à sua esquerda já foram comparados na iteração atual, e os da direita ainda não. A cada nova iteração, o índice atual é iniciado com o valor do índice de referência mais um.

De acordo com a explicação acima, iniciamos o processo sempre com o **IR** apontando para o primeiro elemento do vetor, e o **IA** apontando para **IR + 1**:

Índice	1	2	3	4	5
Elemento	Brasil	Paraguai	Argentina	Uruguai	Chile
	▲	▲			
	IR	IA			

O primeiro passo é compará-los. Se o elemento **IR** for maior do que o elemento **IA**, eles devem trocar de lugar. Como a resposta para este teste foi negativa, avançamos o **IA** para a posição 3:

Índice	1	2	3	4	5
Elemento	Brasil	Paraguai	Argentina	Uruguai	Chile
	▲		▲		
	IR		IA		

Agora, ao realizarmos uma nova comparação, percebemos que os elementos devem ser trocados de lugar, pois na ordem alfabética, **Argentina** vem antes de **Brasil**:

Índice	1	2	3	4	5
Elemento	Argentina	Paraguai	Brasil	Uruguai	Chile
	▲		▲		
	IR		IA		

Avançamos novamente o **IA** (posição 4) e realizamos nova comparação entre eles:

Índice	1	2	3	4	5
Elemento	Argentina	Paraguai	Brasil	Uruguai	Chile
	▲			▲	
	IR			IA	

Como o resultado da comparação acima indicou que não deve haver troca na posição dos elementos, apenas avançamos o **IA** novamente (posição 5):

Índice	1	2	3	4	5
Elemento	Argentina	Paraguai	Brasil	Uruguai	Chile
	▲				▲
	IR				IA

72

Comparando **IA** e **IR** verificamos que não é necessária a troca. **IA** chegou ao final do vetor, indicando que o elemento que agora se encontra no **IR** já está ordenado, por isso o **IR** deve avançar para a próxima posição (2). Além disso, como está iniciando uma nova iteração, o **IA** é inicializado com o valor **IR + 1** (posição 3):

Índice	1	2	3	4	5
Elemento	Argentina	Paraguai	Brasil	Uruguai	Chile
		▲	▲		
		IR	IA		

Seguindo o mesmo processo, percebemos que é necessário trocar **Paraguai** e **Brasil** de lugar:

Índice	1	2	3	4	5
Elemento	Argentina	Brasil	Paraguai	Uruguai	Chile
		▲	▲		
		IR	IA		

Avançamos **IA** para o índice 4. A troca entre os índices 2 e 4 não é necessária:

Índice	1	2	3	4	5
Elemento	Argentina	Brasil	Paraguai	Uruguai	Chile
		▲		▲	
		IR		IA	

Avançamos **IA** mais uma vez, e constatamos que a troca também não é necessária:

Índice	1	2	3	4	5
Elemento	Argentina	Brasil	Paraguai	Uruguai	Chile
		▲			▲
		IR			IA

Como **IA** está novamente no final do vetor, a iteração chegou ao fim, por isso avançamos **IR** novamente e inicializamos **IA** com **IR + 1**:

Índice	1	2	3	4	5
Elemento	Argentina	Brasil	Paraguai	Uruguai	Chile
			▲	▲	
			IR	IA	

Comparamos **IR** a **IA** e percebemos que a troca não é necessária. Então, avançamos **IA** para a posição 5:

Índice	1	2	3	4	5
Elemento	Argentina	Brasil	Paraguai	Uruguai	Chile
			▲		▲
			IR		IA

Desta vez, a troca se faz necessária:

Índice	1	2	3	4	5
Elemento	Argentina	Brasil	Chile	Uruguai	Paraguai
			▲		▲
			IR		IA

Como **IA** chegou novamente ao final do vetor, avançamos **IR** para a posição 4 e inicializamos **IA** com **IR + 1**:

Índice	1	2	3	4	5
Elemento	Argentina	Brasil	Chile	Uruguai	Paraguai
				▲	▲
				IR	IA

A posição máxima de **IR** sempre será **<tamanho_vetor> - 1**, o que indica que esta é a última iteração. Como a inicialização de **IA** já o coloca no último elemento do vetor (5), esta iteração terá apenas uma comparação. Comparando os dois elementos, percebemos a necessidade de trocá-los de lugar:

Índice	1	2	3	4	5
Elemento	Argentina	Brasil	Chile	Paraguai	Uruguai
				▲	▲
				IR	IA

Como **IR** não pode avançar além do penúltimo elemento (neste caso a posição 4) e **IA** já está no fim do vetor, a ordenação está encerrada.

Você percebeu que o tamanho das iterações foi se reduzindo ao longo do processo de ordenação? Isso ocorre porque a cada nova iteração um elemento é ordenado, tomando sua posição definitiva no vetor, o que faz com que a nova iteração tenha um item a menos para percorrer.

Agora veja o algoritmo *Bubble Sort* implementado em pseudocódigo:

```

{Ordenação com o método bolha}
Algoritmo ExemploBubbleSort
Variáveis
    países : Vetor[1..5] de Caractere;
    aux : Caractere;
    ir, ia : Inteiro;
Início
01    países[1] := "Brasil";
02    países[2] := "Paraguai";
03    países[3] := "Argentina";
04    países[4] := "Uruguai";
05    países[5] := "Chile";

06    Para ir De 1 Até 4 Passo 1 Faça
07        Para ia De ir+1 Até 5 Passo 1 Faça
08            Se(paises[ir] > países[ia]) Então
09                aux := países[ir];
10                países[ir] := países[ia];
11                países[ia] := aux;
12            Fim_Se
13            Fim_Para
14        Fim_Para
Fim

```

Perceba que é necessário o uso de uma variável auxiliar na hora de inverter a posição dos elementos no vetor, pois sem o seu uso o valor original de **IR** seria perdido quando este recebesse o valor de **IA**.

O teste de mesa para o algoritmo acima ficaria assim:

Linha	ir	ia	aux	paises[1]	paises[2]	paises[3]	paises[4]	paises[5]
1				Brasil				
2				Brasil	Paraguai			
3				Brasil	Paraguai	Argentina		
4				Brasil	Paraguai	Argentina	Uruguai	
5				Brasil	Paraguai	Argentina	Uruguai	Chile
6	1			Brasil	Paraguai	Argentina	Uruguai	Chile
7	1	2		Brasil	Paraguai	Argentina	Uruguai	Chile
8	1	2		Brasil	Paraguai	Argentina	Uruguai	Chile
7	1	3		Brasil	Paraguai	Argentina	Uruguai	Chile
8	1	3		Brasil	Paraguai	Argentina	Uruguai	Chile
9	1	3	Brasil	Brasil	Paraguai	Argentina	Uruguai	Chile
10	1	3	Brasil	Argentina	Paraguai	Argentina	Uruguai	Chile
11	1	3	Brasil	Argentina	Paraguai	Brasil	Uruguai	Chile
7	1	4	Brasil	Argentina	Paraguai	Brasil	Uruguai	Chile
8	1	4	Brasil	Argentina	Paraguai	Brasil	Uruguai	Chile
7	1	5	Brasil	Argentina	Paraguai	Brasil	Uruguai	Chile
8	1	5	Brasil	Argentina	Paraguai	Brasil	Uruguai	Chile
6	2	5	Brasil	Argentina	Paraguai	Brasil	Uruguai	Chile
7	2	3	Brasil	Argentina	Paraguai	Brasil	Uruguai	Chile
8	2	3	Brasil	Argentina	Paraguai	Brasil	Uruguai	Chile
9	2	3	Paraguai	Argentina	Paraguai	Brasil	Uruguai	Chile
10	2	3	Paraguai	Argentina	Brasil	Brasil	Uruguai	Chile
11	2	3	Paraguai	Argentina	Brasil	Paraguai	Uruguai	Chile
7	2	4	Paraguai	Argentina	Brasil	Paraguai	Uruguai	Chile
8	2	4	Paraguai	Argentina	Brasil	Paraguai	Uruguai	Chile
7	2	5	Paraguai	Argentina	Brasil	Paraguai	Uruguai	Chile
8	2	5	Paraguai	Argentina	Brasil	Paraguai	Uruguai	Chile
6	3	5	Paraguai	Argentina	Brasil	Paraguai	Uruguai	Chile
7	3	4	Paraguai	Argentina	Brasil	Paraguai	Uruguai	Chile
8	3	4	Paraguai	Argentina	Brasil	Paraguai	Uruguai	Chile
7	3	5	Paraguai	Argentina	Brasil	Paraguai	Uruguai	Chile
8	3	5	Paraguai	Argentina	Brasil	Paraguai	Uruguai	Chile
9	3	5	Paraguai	Argentina	Brasil	Paraguai	Uruguai	Chile
10	3	5	Paraguai	Argentina	Brasil	Chile	Uruguai	Chile
11	3	5	Paraguai	Argentina	Brasil	Chile	Uruguai	Paraguai
6	4	5	Paraguai	Argentina	Brasil	Chile	Uruguai	Paraguai
7	4	5	Paraguai	Argentina	Brasil	Chile	Uruguai	Paraguai
8	4	5	Paraguai	Argentina	Brasil	Chile	Uruguai	Paraguai
9	4	5	Uruguai	Argentina	Brasil	Chile	Uruguai	Paraguai
10	4	5	Uruguai	Argentina	Brasil	Chile	Paraguai	Paraguai
11	4	5	Uruguai	Argentina	Brasil	Chile	Paraguai	Uruguai

Obs.: Para facilitar a leitura do teste de mesa, em cada passo foi colocado em azul o valor da variável que sofreu alteração.

Apesar de atingir satisfatoriamente seu objetivo na ordenação de vetores, o método *Bubble Sort* não é aconselhado para grandes volumes de dados, pois o fato de percorrer o vetor diversas vezes para realizar a ordenação compromete sua *performance*.

Pesquisa Sequencial

A **pesquisa sequencial** é o tipo de pesquisa mais simples que existe. A ideia é percorrer o vetor item a item até encontrar o elemento desejado ou descobrir que ele não se encontra no vetor. Existem duas variantes da busca sequencial, uma aplicável a vetores não ordenados e outra a vetores ordenados.

Vetores Não Ordenados

A pesquisa sequencial em vetores não ordenados também é conhecida como pesquisa exaustiva, pois consiste em comparar o valor pesquisado com cada um dos elementos do vetor, até que haja uma correspondência ou que se chegue ao final do vetor (o que indica que o elemento pesquisado não existe no conjunto).

Quando um vetor não está ordenado, a única forma de pesquisa efetiva é a pesquisa exaustiva.

Vejamos o algoritmo de pesquisa exaustiva implementado em pseudocódigo:

```
{Pesquisa exaustiva}
Algoritmo ExemploPesqExaustiva
Variáveis
    países : Vetor[1..5] de Caractere;
    posAtual : Inteiro;
    itemPesquisa : Caractere;
    achou : Lógico;
Início
    Para posAtual De 1 Até 5 Passo 1 Faça
        Escreva("Digite o nome do país para a posição " + posAtual);
        Leia(países[posAtual]);
        Fim_Para

    Escreva("Digite o nome do país que deseja pesquisar:");
    Leia(itemPesquisa);

    posAtual := 1;
    achou := F;
    Enquanto((posAtual <= 5) .E. (achou = F)) Faça
        Se(países[posAtual] = itemPesquisa) Então
            achou := V;
        Senão
            posAtual := posAtual + 1;
        Fim_Se
    Fim_Enquanto

    Se(achou = V) Então
        Escreva("O item " + itemPesquisa +
               " foi encontrado na posição " + posAtual);
    Senão
        Escreva("O item " + itemPesquisa +
               " não foi encontrado no vetor");
    Fim_Se
Fim
```

O algoritmo anterior realiza a leitura de 5 elementos, armazenando-os em um vetor (linhas 1 a 4). Em seguida, solicita a entrada de um valor a ser pesquisado no vetor (linhas 5 e 6). O que mais nos interessa no momento é o trecho que vai da linha 7 até a linha 20, onde são realizadas as pesquisas no vetor e a apresentação do resultado ao usuário.

Vejamos um teste de mesa a fim de comprovar o funcionamento do algoritmo. Para fins de exemplo, iremos simular que o usuário digitou os seguintes valores:

Brasil	Paraguai	Argentina	Uruguai	Chile
--------	----------	-----------	---------	-------

Vamos ainda considerar que o valor a ser pesquisado é **Chile**.

Com isso, iremos executar o teste de mesa apenas do trecho que nos interessa, que é a parte de pesquisa e de retorno ao usuário (linhas 7 a 20):

Linha	posAtual	Achou	itemPesquisa	paises [1]	paises [2]	paises [3]	paises [4]	paises [5]
7	1		Chile	Brasil	Paraguai	Argentina	Uruguai	Chile
8	1	F	Chile	Brasil	Paraguai	Argentina	Uruguai	Chile
9	1	F	Chile	Brasil	Paraguai	Argentina	Uruguai	Chile
10	1	F	Chile	Brasil	Paraguai	Argentina	Uruguai	Chile
12	1	F	Chile	Brasil	Paraguai	Argentina	Uruguai	Chile
13	2	F	Chile	Brasil	Paraguai	Argentina	Uruguai	Chile
9	2	F	Chile	Brasil	Paraguai	Argentina	Uruguai	Chile
10	2	F	Chile	Brasil	Paraguai	Argentina	Uruguai	Chile
12	2	F	Chile	Brasil	Paraguai	Argentina	Uruguai	Chile
13	3	F	Chile	Brasil	Paraguai	Argentina	Uruguai	Chile
9	3	F	Chile	Brasil	Paraguai	Argentina	Uruguai	Chile
10	3	F	Chile	Brasil	Paraguai	Argentina	Uruguai	Chile
12	3	F	Chile	Brasil	Paraguai	Argentina	Uruguai	Chile
13	4	F	Chile	Brasil	Paraguai	Argentina	Uruguai	Chile
9	4	F	Chile	Brasil	Paraguai	Argentina	Uruguai	Chile
10	4	F	Chile	Brasil	Paraguai	Argentina	Uruguai	Chile
12	4	F	Chile	Brasil	Paraguai	Argentina	Uruguai	Chile
13	5	F	Chile	Brasil	Paraguai	Argentina	Uruguai	Chile
9	5	F	Chile	Brasil	Paraguai	Argentina	Uruguai	Chile
10	5	F	Chile	Brasil	Paraguai	Argentina	Uruguai	Chile
11	5	V	Chile	Brasil	Paraguai	Argentina	Uruguai	Chile
16	5	F	Chile	Brasil	Paraguai	Argentina	Uruguai	Chile
17			Saída em tela: "O item Chile foi encontrado na posição 5"					

Vetores Ordenados

A pesquisa sequencial em vetores ordenados segue o mesmo princípio da versão anterior, porém como o vetor está ordenado, não é necessário percorrê-lo até o fim para saber que o elemento pesquisado não se encontra no vetor. Basta percorrer o vetor até encontrar um elemento maior do que o pesquisado (isto se o vetor estiver em ordem crescente).

Vejamos um exemplo em pseudocódigo:

```
{Pesquisa sequencial em vetor ordenado}
Algoritmo ExemploPesqSequencial
Variáveis
    países : Vetor[1..5] de Caractere;
    posAtual, ir, ia : Inteiro;
    itemPesq, aux : Caractere;
Início
01    Para posAtual De 1 Até 5 Passo 1 Faça
02        Escreva("Digite o nome do país para a posição " + posAtual);
03        Leia(paises[posAtual]);
04    Fim_Para

05    Escreva("Digite o nome do país que deseja pesquisar:");
06    Leia(itemPesq);

07    Para ir De 1 Até 4 Passo 1 Faça
08        Para ia De ir+1 Até 5 Passo 1 Faça
09            Se(paises[ir] > paises[ia]) Então
10                aux := paises[ir];
11                paises[ir] := paises[ia];
12                paises[ia] := aux;
13            Fim_Se
14        Fim_Para
15    Fim_Para

16    posAtual := 1;
17    Enquanto((posAtual <= 5) .E. (paises[posAtual] < itemPesq)) Faça
18        posAtual := posAtual + 1;
19    Fim_Enquanto

20    Se(paises[posAtual] = itemPesq) Então
21        Escreva("O item " + itemPesq +
22            " foi encontrado na posição " + posAtual);
23    Senão
24        Escreva("O item " + itemPesq +
25            " não foi encontrado no vetor");
26    Fim_Se
Fim
```

78

O algoritmo acima realiza a leitura de 5 elementos, armazenando-os em um vetor (linhas 1 a 4) e solicita a entrada de um valor a ser pesquisado (linhas 5 e 6). Entre as linhas 7 e 15 é realizada a ordenação do vetor. Novamente, o que mais nos interessa é o trecho onde é realizada a pesquisa, que vai da linha 16 até a linha 24, onde também é dado um retorno ao usuário sobre o resultado da pesquisa.

Vejamos um teste de mesa, a fim de comprovar o funcionamento do algoritmo. Para fins de exemplo, iremos simular que o usuário digitou os mesmos valores utilizados no exemplo da pesquisa exaustiva:

Vamos considerar que o valor a ser pesquisado também é **Chile**.

Novamente iremos executar o teste de mesa apenas no trecho que nos interessa, que é a parte da pesquisa e do retorno ao usuário (linhas 16 a 24). Note que, neste ponto do algoritmo, o vetor já está ordenado:

Linha	posAtual	itemPesquisa	paises[1]	paises[2]	paises[3]	paises[4]	paises[5]
16	1	Chile	Argentina	Brasil	Chile	Paraguai	Uruguai
17	1	Chile	Argentina	Brasil	Chile	Paraguai	Uruguai
18	2	Chile	Argentina	Brasil	Chile	Paraguai	Uruguai
17	2	Chile	Argentina	Brasil	Chile	Paraguai	Uruguai
18	3	Chile	Argentina	Brasil	Chile	Paraguai	Uruguai
17	3	Chile	Argentina	Brasil	Chile	Paraguai	Uruguai
20	3	Chile	Argentina	Brasil	Chile	Paraguai	Uruguai
21	Saída em tela: "O item Chile foi encontrado na posição 3"						

A grande diferença desta versão em relação à pesquisa em vetores não ordenados ocorre quando o item pesquisado não existe no vetor. Se estivéssemos procurando por **Bolívia**, seria possível concluir que o elemento não existe no vetor assim que chegássemos no item **Brasil**. No método exaustivo, teríamos que ir até o final do vetor para ter esta certeza. Veja:

Linha	posAtual	itemPesquisa	paises[1]	paises[2]	paises[3]	paises[4]	paises[5]
16	1	Bolívia	Argentina	Brasil	Chile	Paraguai	Uruguai
17	1	Bolívia	Argentina	Brasil	Chile	Paraguai	Uruguai
18	2	Bolívia	Argentina	Brasil	Chile	Paraguai	Uruguai
17	2	Bolívia	Argentina	Brasil	Chile	Paraguai	Uruguai
20	2	Bolívia	Argentina	Brasil	Chile	Paraguai	Uruguai
22	2	Bolívia	Argentina	Brasil	Chile	Paraguai	Uruguai
23	Saída em tela: "O item Bolívia não foi encontrado no vetor"						

Pesquisa Binária

Quando um vetor está ordenado, podem-se utilizar outros métodos de pesquisa mais eficientes do que a pesquisa sequencial. Um método bastante eficiente é a pesquisa binária, que, a cada comparação, elimina uma série de elementos do universo de pesquisa, o que a torna bem mais rápida do que a pesquisa sequencial, especialmente em vetores grandes.

O principal motivo da eficiência deste método é que são necessárias poucas comparações até encontrarmos o elemento desejado ou descobrir que ele não existe no vetor.

A pesquisa binária funciona basicamente da seguinte forma:

- Compara-se o valor pesquisado com o elemento da posição central do vetor.
- Se o item da posição central é igual ao item pesquisado, a busca acabou.
- Se o item da posição central é maior do que o pesquisado, então o item pesquisado pode estar na primeira metade do conjunto e, com certeza, não estará na segunda metade, o que faz com que esta seja descartada do universo de pesquisa.

- Se o item da posição central é menor do que o pesquisado, então o item pesquisado pode estar na segunda metade do conjunto e, com certeza, não estará na primeira metade, o que faz com que esta seja descartada do universo de pesquisa.
- Repete-se o processo, considerando apenas o novo universo de pesquisa, agora reduzido pela metade com relação à última comparação.

Para que seja possível realizarmos as operações citadas acima, é necessário que tenhamos três índices definidos:

- Índice Inicial (II):** Indica o primeiro elemento do universo de pesquisa. No começo da pesquisa, o **II** está posicionado no primeiro elemento do vetor.
- Índice Final (IF):** Indica o último elemento do universo de pesquisa. No começo da pesquisa, o **IF** está posicionado no último elemento do vetor.
- Índice de Teste (IT):** Indica a próxima posição a ser comparada. No começo da pesquisa, o **IT** está posicionado no elemento mais próximo ao centro do vetor, o que é feito por meio da seguinte fórmula: $IT := (II + IF) / 2$. Quando o resultado da divisão for um número fracionário, considera-se apenas a parte inteira do valor.

Vamos a um exemplo prático. Imagine que devemos pesquisar o valor 89 no vetor abaixo:

11	18	31	50	72	89	91	99
----	----	----	----	----	----	----	----

O primeiro passo é posicionarmos os índices. Como o vetor possui 8 elementos, o índice de teste fica na posição 4.

Índice	1	2	3	4	5	6	7	8
Elemento	11	18	31	50	72	89	91	99
▲				▲				▲
II				IT				IF

Comparamos, então, o elemento da posição 4 com o valor pesquisado. Como 89 é maior que 50, constatamos que se o valor pesquisado estiver no vetor, estará entre **IT+1** e **IF**, o que nos leva a descartar os elementos menores que **IT** (e o próprio **IT**) do universo de pesquisa. Reiniciamos, então, os índices considerando o novo conjunto a ser pesquisado:

Índice	1	2	3	4	5	6	7	8
Elemento	11	18	31	50	72	89	91	99
▲				▲	▲			▲
II				IT		IT		IF

Então, **II** passou para o primeiro elemento do novo conjunto e **IT** passou para o segundo, pois agora existem quatro elementos apenas. Comparamos novamente o elemento referenciado por **IT** com o valor pesquisado e constatamos que o elemento pesquisado está armazenado na posição 6 do vetor.

Perceba que, neste caso, foram necessárias apenas duas comparações para encontrar o valor pesquisado, sendo que com o método sequencial seriam necessárias seis. A *performance* do método binário tende a variar pouco, independente da posição que o elemento pesquisado

ocupe no vetor. A *performance* do método sequencial, por outro lado, cai drasticamente quando o elemento pesquisado ocupa uma posição mais avançada no vetor. Além disso, a diferença de *performance* entre os métodos binário e sequencial aumenta conforme utilizam-se vetores cada vez maiores.

Veja o algoritmo de pesquisa binária representado em pseudocódigo:

```
{Pesquisa binária}
Algoritmo ExemploPesaqBinaria
Variáveis
    valores : Vetor[1..8] de Inteiro;
    ir, ia, aux : Inteiro; {usados na ordenação}
    ii, if, it, itemPesq : Inteiro; {usados na pesquisa}
    achou : Lógico;
Início
01   Para posAtual De 1 Até 8 Passo 1 Faça
02       Escreva("Digite um valor inteiro para a posição " + posAtual);
03       Leia(valores[posAtual]);
04   Fim_Para

05   Escreva("Digite o valor que deseja pesquisar:");
06   Leia(itemPesq);

07   Para ir De 1 Até 7 Passo 1 Faça
08       Para ia De ir+1 Até 8 Passo 1 Faça
09           Se(valores[ir] > valores[ia]) Então
10               aux := valores[ir];
11               valores[ir] := valores[ia];
12               valores[ia] := aux;
13           Fim_Se
14       Fim_Para
15   Fim_Para

16   achou := F;
17   ii := 1; {índice inicial}
18   if := 8; {índice final}
19   Enquanto((ii <= if) .E. (achou = F)) Faça
20       it := (ii + if) / 2; {índice de teste}
21       Se(valores[it] = itemPesq) Então
22           achou := V;
23       Senão
24           Se(valores[it] > itemPesq) Então
25               {descarta da pesquisa os elementos da direita, até it}
26               if := it - 1;
27           Senão
28               {descarta da pesquisa os elementos da esquerda, até it}
29               ii := it + 1;
30       Fim_Se
31   Fim_Enquanto
```

```

31  Se(achou = V) Então
32      Escreva("O valor " + itemPesquisa +
            " foi encontrado na posição " + it);
33  Senão
34      Escreva("O valor " + itemPesquisa +
            " não foi encontrado no vetor");
35  Fim_Se
Fim

```

O algoritmo acima realiza a leitura de 8 valores inteiros, armazenando-os em um vetor (linhas 1 a 4) e solicita a entrada de um valor a ser pesquisado (linhas 5 e 6). Entre as linhas 7 e 15, é realizada a ordenação do vetor. A pesquisa binária é realizada entre as linhas 16 e 30, e entre as linhas 31 e 35 é apresentado o resultado da pesquisa ao usuário.

Veja um teste de mesa realizado com os dados do exemplo anterior:

11	18	31	50	72	89	91	99
----	----	----	----	----	----	----	----

O valor a ser pesquisado é 89.

Vamos executar o teste de mesa apenas no trecho onde ocorre a pesquisa binária e apresentação do resultado (linhas 16 a 35). Como os elementos do vetor não sofrem alteração de valor ao longo da execução do algoritmo, podemos representá-los em uma tabela à parte:

Valores							
valores [1]	valores [2]	valores [3]	valores [4]	valores [5]	valores [6]	valores [7]	valores [8]
11	18	31	50	72	89	91	99

A execução do teste de mesa do algoritmo com estes valores ficaria da seguinte forma:

Linha	ii	if	it	itemPesquisa	Achou
16				89	F
17	1			89	F
18	1	8		89	F
19	1	8		89	F
20	1	8	4	89	F
21	1	8	4	89	F
23	1	8	4	89	F
24	1	8	4	89	F
26	1	8	4	89	F
27	5	8	4	89	F
19	5	8	4	89	F
20	5	8	6	89	F
21	5	8	6	89	F
22	5	8	6	89	V
19	5	8	6	89	V
31	5	8	6	89	V
32	Saída em tela: "O valor 89 foi encontrado na posição 6 "				



Atividades

Os algoritmos criados nos exercícios a seguir devem ser todos representados em pseudocódigo.

- 1) Crie um algoritmo que solicite ao usuário 10 valores inteiros, armazenando-os em um vetor. Em seguida, o vetor deve ser ordenado e os valores exibidos na tela em ordem crescente.
- 2) Construa um algoritmo que leia o nome de 20 pessoas e depois os imprima em ordem decrescente.
- 3) Crie um algoritmo que leia 15 valores reais, guardando-os em um vetor e, em seguida, realize uma pesquisa exaustiva no vetor com um novo valor informado pelo usuário.
- 4) Crie um algoritmo que solicite o nome de 10 alunos, armazenando-os em um vetor. Em seguida deve ser realizada uma pesquisa sequencial com um nome informado pelo usuário.
- 5) Crie um algoritmo que solicite o nome de 20 produtos. Em seguida, o algoritmo deve solicitar o nome do produto a ser pesquisado e escrever na tela a posição em que ele se encontra no vetor. A pesquisa deve ser feita utilizando o método binário.
- 6) Considerando a necessidade de comparar o consumo de combustível de 10 veículos, crie um algoritmo que realize as seguintes atividades:
 - a. Solicite o modelo do veículo e a quantidade de quilômetros que ele é capaz de rodar com um litro de gasolina;
 - b. Gere uma média de consumo entre todos os carros e exiba na tela;
 - c. Contabilize quantos veículos consomem menos do que a média geral e exiba esta informação na tela;
 - d. Gere um *ranking* de consumo, começando pelo veículo mais econômico até o que mais consome combustível;
 - e. Envie para a impressora o *ranking* gerado;
 - f. Permita que o usuário informe o nome de um veículo para saber que posição ele ocupa no *ranking*, e qual sua diferença de consumo para o carro mais econômico. Em seguida pergunte se o usuário deseja consultar outro veículo (considerar 0 para sair e 1 para nova consulta).

Estruturas de Dados Heterogêneas

Ao contrário das estruturas homogêneas, que permitem armazenar apenas elementos de um mesmo tipo de dado, as **estruturas de dados heterogêneas** são capazes de guardar diferentes tipos de dados em uma mesma estrutura.

Imagine que tenhamos de ler e manipular os seguintes dados:

Nome Produto (tipo Caractere)	Preço Custo (tipo Real)	Preço Venda (tipo Real)	Estoque (tipo Inteiro)
Mouse óptico	27,00	36,25	8
Teclado ABNT	22,79	29,00	14
Modem ADSL	53,50	71,00	6
Speaker	14,90	19,90	4
Estabilizador	37,20	52,50	15

Com os recursos que vimos até agora, a melhor opção seria armazenar estas informações em quatro vetores ou dois vetores e uma matriz, porque tanto vetores quanto matrizes só podem armazenar elementos que sejam do mesmo tipo de dado.

Para atender a este tipo de demanda, foram criadas as estruturas de dados heterogêneas, que são representadas por um tipo de estrutura chamada “**Registro**”.

Registros

Os **registros** permitem agrupar as informações referentes a uma mesma entidade (um aluno, um produto, etc.) em uma mesma estrutura. Com os registros, ao invés de termos as informações referentes a um determinado elemento espalhadas em vários vetores/matrizes, podemos ter todas elas agrupadas em um só lugar. Seria como se cada produto acima fosse representado assim:

Nome	Mouse óptico
Preço Custo	27,00
Preço Venda	36,25
Estoque	8

Podemos entender um registro como sendo um conjunto de variáveis que armazenam informações relacionadas. É importante você lembrar que um registro deve ser utilizado para representar uma entidade ou parte dela. Embora seja possível a criação de registros para armazenar dados não relacionados, isso é totalmente desaconselhável, pois vai contra o seu propósito e, ao invés de melhorar, irá piorar a organização dos dados.

As variáveis que compõem os registros são chamadas de campos. Deste modo, é correto afirmar que o registro representado acima possui quatro campos.

Em termos conceituais, os registros são tipos de dados compostos definidos pelo programador. Isto significa que, antes que possamos utilizar um registro, este precisa ser definido.

Pense no registro como um novo tipo de dado que você está criando. A primeira coisa a fazer é definir o tipo, para que, posteriormente, seja possível declarar elementos deste novo tipo.

Definição

A definição de um registro é a etapa em que iremos montar sua estrutura, indicando quais informações o registro deverá ser capaz de armazenar. Pense na definição do registro como um molde para criar registros daquele tipo. Esta definição é realizada em um bloco próprio, chamado **Tipos**, que deve estar antes do bloco de declaração de variáveis.

No pseudocódigo, a definição de registros segue a seguinte sintaxe:

```
<identificadorRegistro> : Registro
    <declaração dos campos do registro>
        Fim_Registro
```

85

Exemplo:

```
{Definição de registro}
Tipos
    RegAluno : Registro
        nome : Caractere;
        nota1 : Real;
        nota2 : Real;
    Fim_Registro
```

Para facilitar a identificação dos registros, iremos adotar como padrão definir todos os registros com o prefixo **Reg**.

Declaração

Declarar um registro é reservar uma área de memória capaz de armazenar um conjunto de dados, cujos nomes e tipos estão especificados na definição do registro.

A declaração de registros ocorre basicamente pelo mesmo processo que usamos para as declaração das variáveis. Por exemplo, quando declaramos uma variável do tipo Real, estamos criando uma área de armazenamento de informações com base em um tipo de dado definido anteriormente. A diferença é que, até o momento, nós utilizávamos apenas os tipos de dados primitivos, ou seja, aqueles já existentes. Agora, com a definição de registros, nós começamos a criar nossos próprios tipos de dados, tomando como base os tipos primitivos.

Para declarar um registro, utilizamos o mesmo padrão de declaração das variáveis:

```
<nome do registro> : <tipo do registro>;
```

Exemplo:

```
{Declaração de registro}
Variáveis
    alunol : RegAluno;
```

Atribuição e Leitura

Para realizar as operações de atribuição e de leitura de valor nos campos de um registro, é necessário informar o campo ao qual desejamos acessar, por meio da seguinte sintaxe:

```
{para atribuir}
<nomeRegistro>.<nomeCampo> := <valor>

{para ler}
<nomeVariavel> := <nomeRegistro>.<nomeCampo>;
```

Vejamos um algoritmo de exemplo, ilustrando o uso de registros:

```
{Exemplo de utilização de registros}
Algoritmo ExemploRegistros1
    Tipos
        RegAluno : Registro
            nome : Caractere;
            nota1 : Real;
            nota2 : Real;
        Fim_Registro
```

```
Variáveis  
    aluno1 : RegAluno;  
    aluno2 : RegAluno;  
    mediaTemp : Real;  
  
Início  
01    Escreva("Informe o nome do aluno 1");  
02    Leia(aluno1.nome);  
03    Escreva("Informe a nota 1 do aluno 1");  
04    Leia(aluno1.nota1);  
05    Escreva("Informe a nota 2 do aluno 1");  
06    Leia(aluno1.nota2);  
  
07    Escreva("Informe o nome do aluno 2");  
08    Leia(aluno2.nome);  
09    Escreva("Informe a nota 1 do aluno 2");  
10    Leia(aluno2.nota1);  
11    Escreva("Informe a nota 2 do aluno 2");  
12    Leia(aluno2.nota2);  
  
13    Imprima("O nome do aluno 1 é " + aluno1.nome);  
14    mediaTemp := (aluno1.nota1 + aluno1.nota2) /2;  
15    Imprima("A média do aluno 1 é " + mediaTemp);  
  
16    Imprima("O nome do aluno 2 é " + aluno2.nome);  
17    mediaTemp := (aluno2.nota1 + aluno2.nota2) /2;  
18    Imprima("A média do aluno 2 é " + mediaTemp);  
  
Fim
```

O algoritmo acima define um registro capaz de armazenar os dados de um aluno (bloco **Tipos**). Em seguida, no bloco **Variáveis**, declara dois registros do tipo definido. No corpo do algoritmo, é solicitado ao usuário a entrada dos dados de cada aluno (linhas 1 a 12), e as informações são armazenadas nos registros. Por fim, os dados dos alunos são enviados para a impressora (linhas 13 a 18).

Este é um exemplo bastante simples, que serve para ilustrar apenas um uso básico dos registros. Em aplicações “do mundo real”, os registros muitas vezes são utilizados para trabalhar com volumes maiores de dados.

Mas, por exemplo, se a turma tiver 40 alunos, teríamos que declarar 40 registros? Não. E você já conhece a resposta para atender a esta demanda. Anteriormente, nós estudamos uma estrutura de dados capaz de armazenar um conjunto de elementos de mesmo tipo, lembra? Estamos falando dos vetores. Como um registro é um tipo de dado, ele pode ser armazenado em vetores. Então, para atender à demanda dos 40 alunos, bastaria criarmos um vetor contendo 40 registros do tipo **RegAluno**. Veja:

```
{Exemplo com vetor de registros}
Algoritmo ExemploRegistros2
    Tipos
        RegAluno : Registro
            nome : Caractere;
            nota1 : Real;
            nota2 : Real;
        Fim_Registro

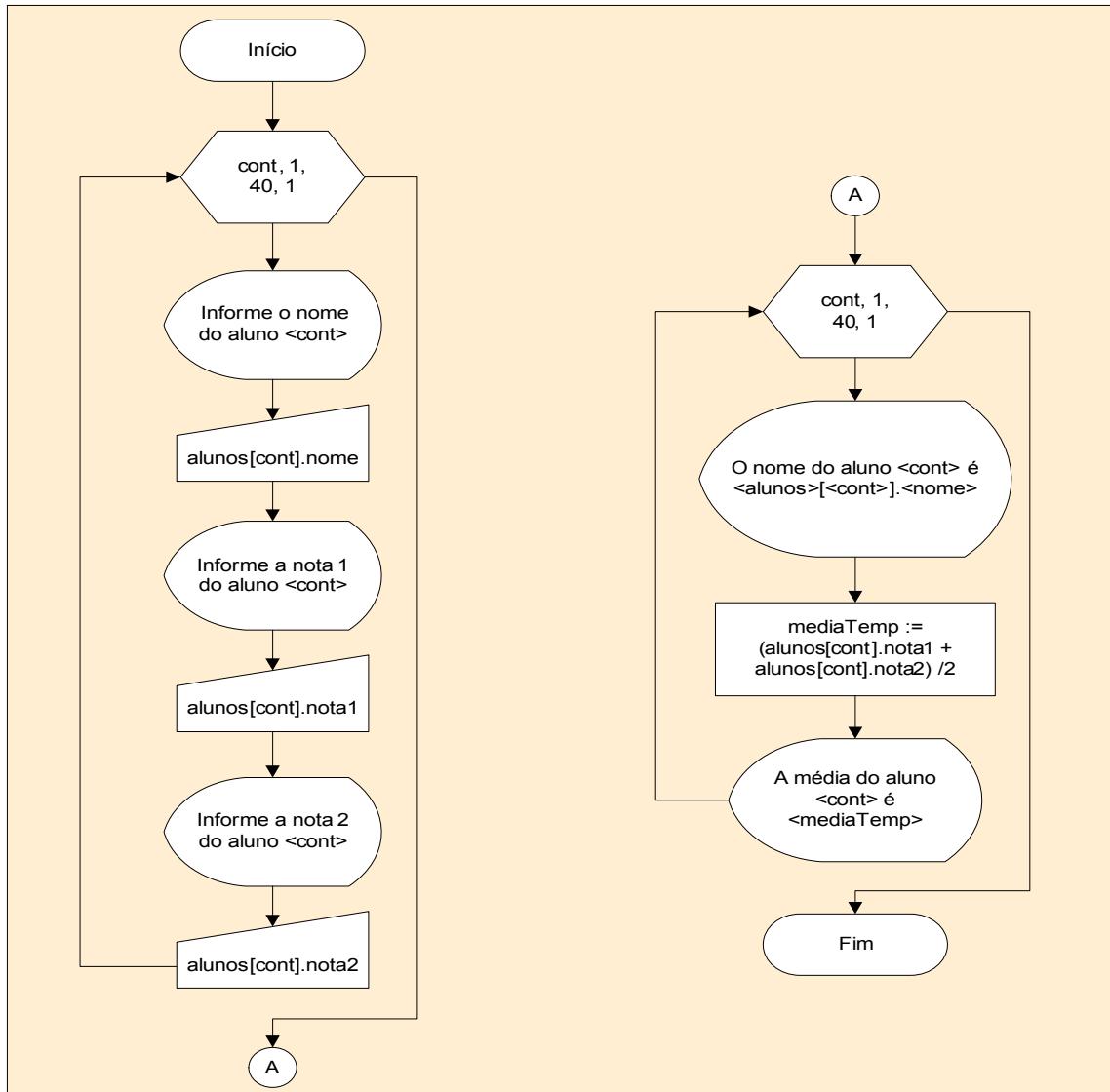
    Variáveis
        alunos : Vetor[1..40] De RegAluno;
        mediaTemp : Real;
        cont : Inteiro;

    Início
    01    Para cont De 1 Até 40 Passo 1 Faça
        02        Escreva("Informe o nome do aluno " + cont);
        03        Leia(alunos[cont].nome);
        04        Escreva("Informe a nota 1 do aluno " + cont);
        05        Leia(alunos[cont].nota1);
        06        Escreva("Informe a nota 2 do aluno " + cont);
        07        Leia(alunos[cont].nota2);
        08    Fim_Para

    09    Para cont De 1 Até 40 Passo 1 Faça
        10        Imprima("O nome do aluno " + cont + " é " +
                        alunos[cont].nome);
        11        mediaTemp := (alunos[cont].nota1 + alunos[cont].nota2) /2;
        12        Imprima("A média do aluno " + cont + " é " + mediaTemp);
        13    Fim_Para
    Fim
```

O algoritmo acima declara um vetor com 40 registros e, entre as linhas 1 e 8, popula o vetor com dados solicitados ao usuário. Depois, entre as linhas 9 e 13, imprimem-se os nomes e as médias de todos os alunos. A utilização de vetores de registros permite a criação de algoritmos mais enxutos, simples e funcionais.

Veja este algoritmo representado em fluxograma:



A definição de registros não é representada nos fluxogramas. E assim como acontece em relação às variáveis e estruturas homogêneas, a declaração de registros também não deve ser representada.

Com relação à estrutura dos registros, existe outra possibilidade que ainda não exploramos: a criação de registros contendo vetores. Esta é uma opção interessante para determinados tipos de implementação, pois mantém a estrutura dos dados bem organizada, ao mesmo tempo em que facilita a atribuição/leitura de valores por meio de *loops*.

Imagine que tenhamos de armazenar os números sorteados em cada um dos últimos dez concursos da Mega-Sena. Teremos, então, um vetor de registros em que cada registro terá o número do concurso, a data do sorteio e um vetor com os seis números sorteados.

Veja como ficaria um algoritmo para atender a esta demanda:

```
{Exemplo com vetor de registros, em que cada registro tem um vetor}

Algoritmo ExemploRegistros1
    Tipos
        RegConcurso : Registro
            numero : Inteiro;
            data : Caractere;
            numeros : Vetor[1..6] De Inteiro;
        Fim_Registro

    Variáveis
        concursos : Vetor[1..10] De RegConcurso;
        cont, conNum : Inteiro;

    Início
        01    Escreva("Informe os dados dos últimos 10 concursos
                  da megasena");
        02    Para cont De 1 Até 10 Passo 1 Faça
            03        Escreva("Informe o numero do concurso " + cont);
            04        Leia(concursos[cont].numero);
            05        Escreva("Informe a data do concurso " + cont);
            06        Leia(concursos[cont].data);

            07        Para contNum de 1 até 6 Passo 1 Faça
                08            Escreva("Informe o número " + contNum);
                09            Leia(concursos[cont].numeros[contNum]);
            10        Fim_Para
        11    Fim_Para

        12    Imprima("Relação dos números sorteados nos últimos
                  10 concursos da megasena");
        13    Para cont De 1 Até 10 Passo 1 Faça
            14        Imprima("Numero do concurso: " + concursos[cont].numero);
            15        Imprima("Data do concurso " + concursos[cont].data);
            16        Imprima("Números sorteados: ");
            17        Para contNum de 1 até 6 Passo 1 Faça
                18            Escreva("[ " + concursos[cont].numeros[contNum] + " ]");
            19        Fim_Para
        20    Fim_Para
    Fim
```

90

Estruturas de Dados Heterogêneas

Assim, criamos um algoritmo que manipula um vetor de registros, onde cada registro possui outro vetor dentro de si. Preste atenção nas linhas 9 e 18, cujos dados dos vetores internos de cada registro são atribuídos e lidos, respectivamente.

Como estamos acessando nosso registro dentro de um vetor, precisamos informar o índice em que o registro se encontra no vetor (**concursos[cont]**), e como dentro do nosso registro existe um outro vetor, também temos que indicar o índice para acessar um elemento deste vetor (**numeros[contNum]**). Para que seja possível este acesso, utilizamos duas estruturas de repetição, uma para percorrer o vetor de registros, e outra para percorrer o vetor interno de cada registro.

Como você pode ver, a utilização conjunta de estruturas de dados homogêneas e heterogêneas permite manipular maiores volumes de dados de forma mais organizada e eficiente, com algoritmos bem estruturados e sem grande complexidade.



Atividades

Todos os exercícios a seguir devem ser implementados em pseudocódigo.

- 1) Crie um algoritmo que utilize um registro para armazenar os dados de uma ficha cadastral informados pelo usuário (no mínimo 5 campos). Os dados do registro lido deverão ser enviados para a impressora.
- 2) Crie um programa capaz de armazenar o resultado dos 10 jogos realizados em uma rodada do campeonato brasileiro de futebol. Para cada jogo, será necessário armazenar as seguintes informações:
 - a. Data do jogo;
 - b. Local;
 - c. Time mandante;
 - d. Time visitante;
 - e. Escore do mandante;
 - f. Escore do visitante.

91

Depois dos dados informados, o algoritmo deve escrever na tela os dados de cada jogo, indicando quem foi o vencedor do confronto. Utilize um vetor de registros na sua implementação.

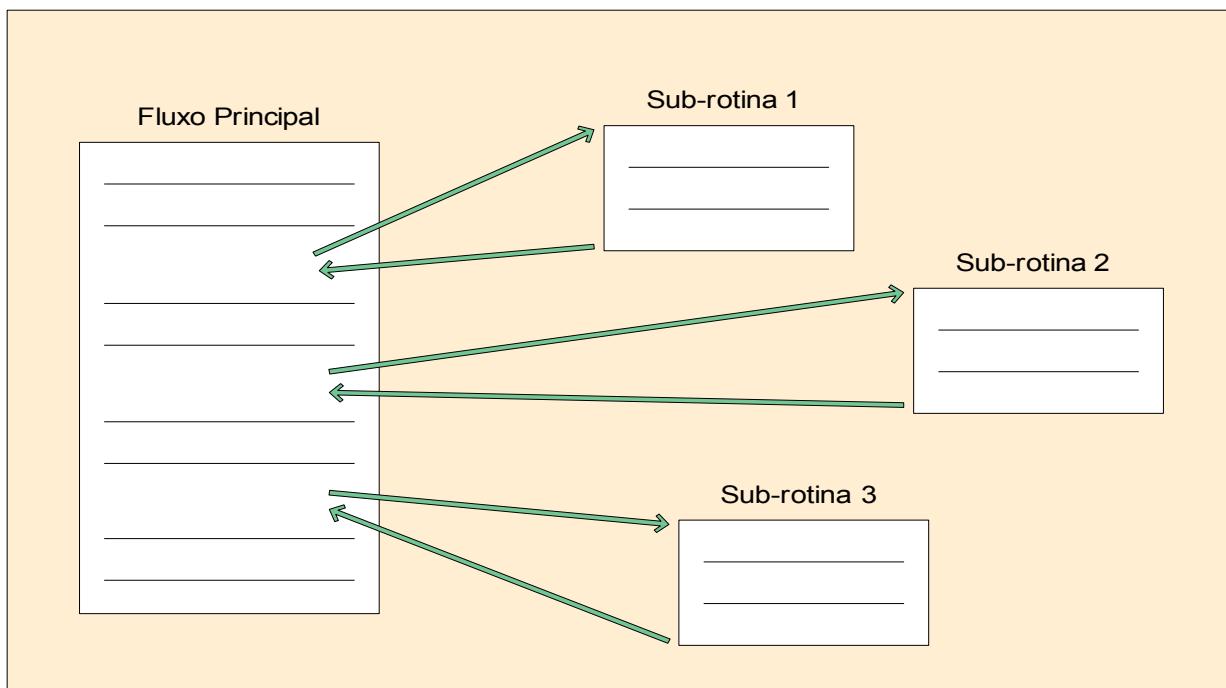
- 3) Crie um algoritmo para realizar empréstimos de DVDs em uma locadora, sabendo que o cliente pode retirar até 5 DVDs de cada vez. É necessário imprimir um relatório com os dados do cliente e os códigos dos filmes que ele está levando emprestado. Na sua implementação, utilize um vetor dentro do registro para armazenar os códigos dos filmes que estão sendo retirados.

Sub-rotinas

Você deve lembrar que as estruturas de controle são mecanismos que permitem modificar o fluxo de execução dos algoritmos. Agora, iremos conhecer outro tipo de estrutura de controle, chamada de sub-rotina.

De forma bastante simples, uma **sub-rotina** é um conjunto de instruções que não faz parte do corpo principal do algoritmo, mas que pode ser executado n (zero ou muitas) vezes ao longo da execução do algoritmo, por meio de chamadas ao seu nome.

Quando uma sub-rotina é chamada, o fluxo de execução é desviado para o corpo da sub-rotina. Ao final da execução, o fluxo de execução retorna para o ponto de onde partiu a chamada, continuando a execução a partir da próxima instrução abaixo da chamada.



Um dos princípios do paradigma científico cartesiano diz que um problema grande pode ser resolvido de forma mais fácil se for subdividido em vários problemas pequenos. Esta é a ideia por trás das sub-rotinas. Ao invés de termos um algoritmo enorme, cheio de estruturas de seleção e repetição, podemos quebrá-lo em vários pedaços, gerando sub-rotinas responsáveis por determinadas tarefas que fazem parte da resolução do problema. Assim, podemos pensar na lógica necessária para resolver cada um dos pequenos problemas, esquecendo-nos temporariamente do todo, enquanto nos focamos na tarefa que uma determinada sub-rotina deverá executar.

O uso de sub-rotinas permite modularizar os algoritmos, de modo que os conjuntos de instruções responsáveis por determinadas tarefas estejam estruturados de forma mais organizada, facilitando seu entendimento e manutenção. Além disso, aumentamos a possibilidade de reutilização das soluções implementadas.

Existem dois tipos de sub-rotinas: os **Procedimentos** e as **Funções**.

Procedimentos

O **procedimento**, também chamado de **Proc**, é um tipo de sub-rotina comumente utilizada para agrupar instruções relacionadas a uma determinada atividade. Um dos usos frequentes dos procedimentos consiste em realizar a interação com o usuário – os comandos **Leia**, **Imprima** e **Escreva** podem ser implementados dentro de procedimentos. Isso torna a leitura do algoritmo mais agradável e, ao mesmo tempo, encapsula cada uma das atividades em blocos de instruções especializadas em resolver uma parte do problema. Isto facilita o reaproveitamento de soluções entre os algoritmos, pois uma vez que já existe uma sub-rotina capaz de resolver uma determinada parte de um problema, basta copiá-la para outro algoritmo e utilizá-la.

Um procedimento é declarado em um bloco próprio, com início e fim demarcados. Dentro deste bloco insere-se o conjunto de comandos relacionados à atividade para a qual o procedimento foi criado. A declaração de procedimentos deve ser realizada após o bloco de variáveis e antes do demarcador de início do corpo principal do algoritmo. O corpo principal do algoritmo é o último segmento a ser declarado e é por onde se inicia a execução do algoritmo.

O padrão de declaração de procedimentos em pseudocódigo segue a seguinte sintaxe:

```
Procedimento <nome do procedimento>()
    Início
        <comandos>
    Fim_Procedimento
```

Depois de sua declaração, um procedimento poderá ser executado em qualquer ponto do algoritmo. A chamada a um procedimento é realizada por meio de seu nome, por isso é interessante que este seja condizente com as atividades que o procedimento executa. Além disso, é uma boa prática nomear os procedimentos iniciando com um verbo. Exemplos: **LerNotas()**, **ImprimirRelatorio()**, **CalcularMedias()**, **ListarAlunosAprovados()**.

Vejamos um exemplo simples, utilizando procedimento:

```
{Exemplo de procedimento}
Algoritmo ExemploProcedimento
Variáveis
    valores : Vetor[1..10] De Inteiro;
    contador : Inteiro;

Procedimento LerValores()
Início
    Para contador De 1 Até 10 Passo 1 Faça
        Escreva("Digite um valor para a posição " + contador + ":");
        Leia(valores[contador]);
    Fim_Para
Fim_Procedimento

Procedimento EscreverValores()
Início
    Para contador De 1 Até 10 Passo 1 Faça
        Escreva("O elemento da posição " + contador +
            " vale " + valores[contador])
    Fim_Para
Fim_Procedimento

Início
    LerValores();
    EscreverValores();
Fim
```

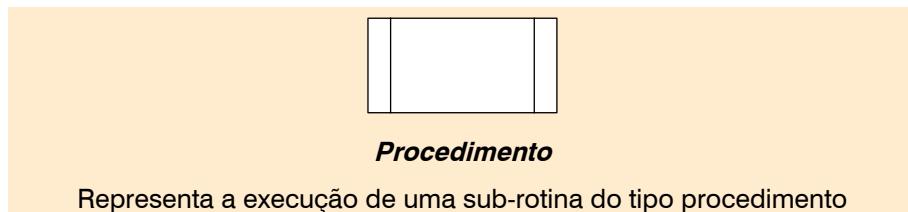
O algoritmo acima apenas popula um vetor e depois escreve na tela os valores lidos. Perceba que foram declarados dois procedimentos, um para ler e outro para escrever os dados. No corpo principal do algoritmo, foram inseridas as chamadas aos procedimentos.

Procure pensar nas possibilidades que este método de trabalho nos abre. Por exemplo, se fosse necessário escrever os valores diversas vezes, bastaria inserir no corpo do algoritmo diversas chamadas ao procedimento, ao invés de repetir várias vezes o mesmo bloco de instruções.

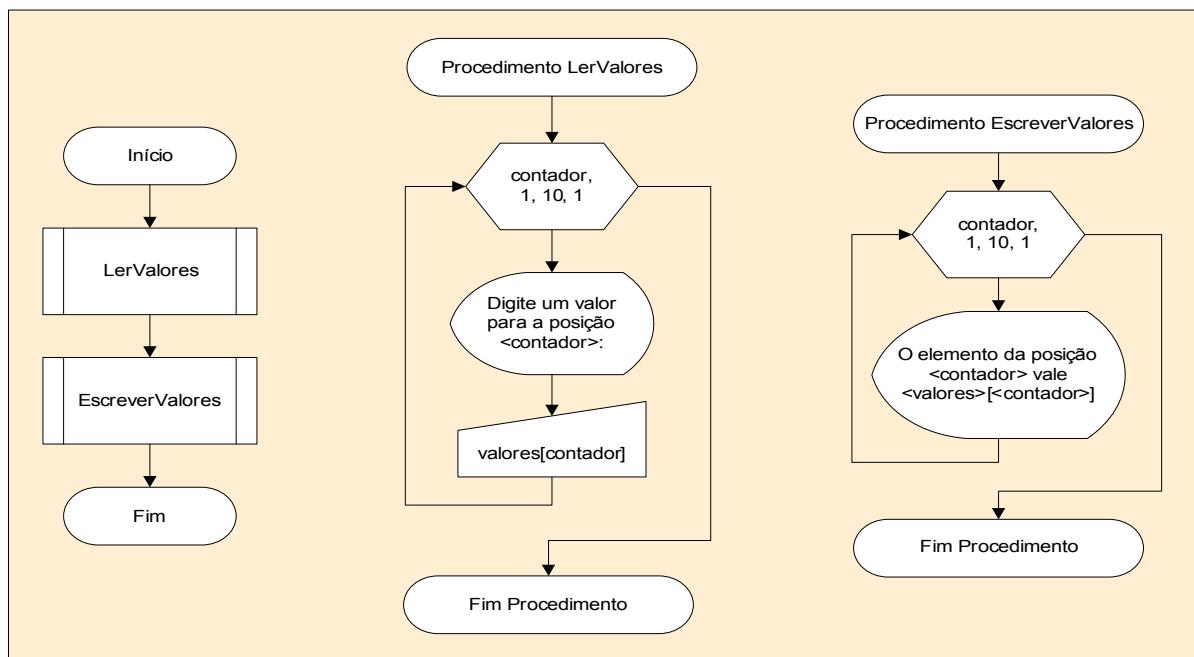
Nos fluxogramas, os procedimentos são representados utilizando-se os seguintes padrões:

- A declaração do procedimento é representada em outro fluxograma, que deve estar anexo ao principal.
- Nos fluxogramas que representam procedimentos, no lugar das instruções **Início** e **Fim**, utiliza-se **Procedimento <Nome do Procedimento>** e **Fim Procedimento**, mantendo-se o uso da figura **terminador** em ambos os casos.

- A chamada de um procedimento é representada pela figura “procedimento”, apresentada abaixo:



Veja o algoritmo do exemplo anterior, representado em fluxograma:



95

Sub-rotinas

Agora vejamos o uso de procedimentos em um exemplo um pouco mais elaborado:

```

{Exemplo_Procedimento_2}
01 Algoritmo ExemploProcedimento2
02 Tipos
03     RegFuncionario : Registro
04         nome : Caractere;
05         cargo : Caractere;
06         salario : Real;
07         bonificacao : Real;
08     Fim_Registro
09 Variáveis
10     funcionarios : Vetor[1..50] De RegFuncionario;
11     cont, ir, ia : Inteiro;
12     aux : RegFuncionario;
13     totalTemp : Real;

```

```
14 Procedimento LerFuncionarios()
15 Início
16     Escreva("Informe os dados dos funcionários");
17     Para cont De 1 Até 50 Passo 1 Faça
18         Escreva("Informe o nome do funcionário " + cont);
19         Leia(funcionarios[cont].nome);
20         Escreva("Informe o cargo do funcionario " + cont);
21         Leia(funcionarios[cont].cargo);
22         Escreva("Informe o salário do funcionário " + cont);
23         Leia(funcionarios[cont].salario);
24         funcionarios[cont].bonificacao := 0;
25     Fim_Para
26 Fim_Procedimento

{Ordena o vetor de funcionários com base no cargo, para que saia ordenado no relatório}
Procedimento OrdenarFuncionariosPorCargo()
27 Início
28     Para ir De 1 Até 49 Passo 1 Faça
29         Para ia De ir+1 Até 50 Passo 1 Faça
30             Se(funcionarios[ir].cargo > funcionarios[ia].cargo) Então
31                 aux := funcionários[ir];
32                 funcionários[ir] := funcionários[ia];
33                 funcionários[ia] := aux;
34             Fim_Se
35         Fim_Para
36     Fim_Para
37 Fim_Procedimento

{Gera bonificação para os funcionários, seguindo as seguintes regras:
 - Vendedor: 10% sobre o salário
 - Caixa: 5% sobre o salário
 - Demais cargos: Não há bonificação}
Procedimento GerarBonificacao()
39 Início
40     Para cont De 1 Até 50 Passo 1 Faça
41         Se(Funcionarios[cont].cargo = "Vendedor") Então
42             Funcionarios[cont].bonificacao :=
43                 Funcionarios[cont].salario * 0,1;
44         Senão
45             Se(Funcionarios[cont].cargo = "Caixa") Então
46                 Funcionarios[cont].bonificacao :=
47                     Funcionarios[cont].salario * 0,05;
48             Fim_Se
49         Fim_Se
50     Fim_Para
51 Fim_Procedimento
```

```

51 Procedimento GerarRelatorioHonorarios()
52 Início
53 OrdenarFuncionariosPorCargo(); {solicita a ordenação do vetor}
54 Para cont De 1 Até 50 Passo 1 Faça
55   Imprima("-----");
56   Imprima("Funcionário: " + funcionario[cont].nome);
57   Imprima("Cargo: " + funcionario[cont].cargo);
58   Imprima("Salário: " + funcionario[cont].salario);
59   Imprima("Bonificação: " + funcionario[cont].bonificacao);
60   totalTemp := funcionario[cont].salario +
61     funcionario[cont].bonificacao;
62   Imprima("Total a Receber: " + totalTemp);
63 Fim_Para
64   Imprima("-----");
65 Escreva("O Relatório de Honorários foi enviado
66 para a impressora");
67 Fim_Procedimento
68
69 Início
70   LerFuncionarios();
71   GerarBonificacao();
72   GerarRelatorioHonorarios();
73   Escreva("Fim de execução");
74 Fim

```

- Entre as linhas 51 e 65, encontra-se a declaração do procedimento responsável por gerar um relatório de honorários dos funcionários. É este procedimento que chama o procedimento de ordenação, antes de imprimir os dados. A chamada ao procedimento de ordenação poderia ser feita no corpo principal do algoritmo, antes da chamada ao procedimento que gera o relatório, mas como a ordenação interessa ao relatório (podemos considerar uma sub-tarefa do relatório) fica mais lógico solicitar a ordenação do relatório a partir deste ponto.

Funções

Assim como os procedimentos, as **funções** são sub-rotinas comumente utilizadas para encapsular a execução de uma determinada atividade. A diferença básica entre as funções e os procedimentos é que as funções sempre retornam um valor ao final de sua execução. Este valor retornado pode ser utilizado em uma atribuição ou, então, enviado para a tela ou para a impressora.

O padrão de declaração de funções em pseudocódigo segue a seguinte sintaxe:

```
Função <nome da função>() : <tipo de dado retornado>
Início
    <comandos>
    <nome da função> := <valor de retorno>;
Fim_Função
```

98

Sub-rotinas

No pseudocódigo, as convenções de nomenclatura adotadas para as funções são as mesmas utilizadas para os procedimentos.

Vejamos um exemplo de algoritmo utilizando função:

```
{Exemplo de função}
01 Algoritmo ExemploFuncao
02 Variáveis
03     valorVenda, valorComissao : Real;

{Retorna um valor de comissão referente ao valor da venda.
Regras: 5% para vendas >= 500 e 7,5% para vendas < 500}
04 Função CalcularComissao() : Real
05 Início
06     Se(valorVenda >= 500) Então
07         CalcularComissao := valorVenda * 0,05;
08     Senão
09         CalcularComissao := valorVenda * 0,075;
10     Fim_Se
11 Fim_Função
```

```

12 Início
13     valorVenda := 0;
14     Escreva("Cálculo de comissão sobre venda");
15     Repita
16         Escreva("Digite o valor da venda ou 0 para sair: ");
17         Leia(valorVenda);
18         valorComissao := CalcularComissao();
19         Escreva("O valor da comissão é: " + valorComissao);
20     Até_Que(valorVenda = 0)
21     Escreva("Fim de execução");
22 Fim

```

Entre as linhas 4 e 11, é declarada uma função responsável por calcular o valor da comissão sobre a venda informada. A função é chamada na linha 18, sendo o seu valor de retorno atribuído à variável **valorComissao**.

Ao contrário dos procedimentos, cuja chamada só pode ser realizada isoladamente (em uma linha exclusiva), as funções também podem ser chamadas dentro de expressões, comandos ou chamadas a outras funções.

As funções também trazem a vantagem de reduzir o número de variáveis necessárias ao longo do algoritmo. Por exemplo, no algoritmo acima, as linhas 18 e 19 poderiam ser substituídas pela linha abaixo, reduzindo o tamanho do algoritmo e evitando a criação da variável **valorComissao**:

```
Escreva("O valor da comissão é: " + CalcularComissao());
```

99

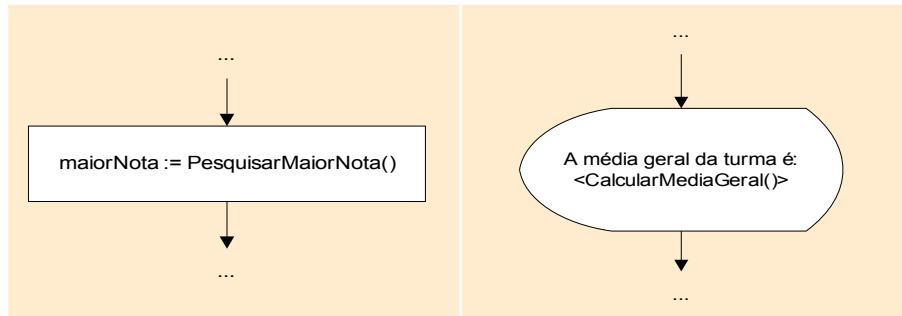
Mas atenção! Nem sempre economizar no número de variáveis vale a pena. Caso seja necessário utilizar o valor retornado pela função em outros pontos do algoritmo, provavelmente será mais vantajoso, em termos de economia de recursos, manter a variável na memória, do que executar várias vezes a função.

Na hora de definir a implementação de um algoritmo, procure levar em consideração outras possíveis soluções para o problema. Utilize sempre o bom senso, pense sobre as consequências resultantes de cada uma das suas opções de implementação.

Nos fluxogramas, as funções são representadas utilizando-se os seguintes padrões:

- Do mesmo modo que ocorre com um procedimento, a declaração de uma função é representada em outro fluxograma, que deve estar anexo ao principal.
- Nos fluxogramas que representam funções, no lugar das instruções **Início** e **Fim**, utiliza-se **Função <Nome do Procedimento>** e **Fim Função**, mantendo-se o uso da figura **Terminador** em ambos os casos.

- A representação de chamada a uma função não é realizada por meio de uma figura específica, sendo que a representação depende da forma como o retorno da função é utilizado. Quando o retorno é atribuído a uma variável, utiliza-se a figura **Processamento**; já quando o resultado é enviado para a tela, utiliza-se a figura **Vídeo**; e assim por diante, sempre com a chamada à função aparecendo dentro da figura. Exemplo:



Vejamos outro exemplo de algoritmo utilizando funções:

```

100
Sub-rotinas
{Exemplo funções 2}
01 Algoritmo ExemploFuncoes2
02 Tipos
03     RegAluno : Registro
04             nome : Caractere;
05             nota1, nota2, media : Real;
06             Fim_Registro
07 Variáveis
08     alunos : Vetor[1..40] De RegAluno;
09     notasTemp : Real;
10     cont, qtdeAprovados : Inteiro;

11 Procedimento LerAlunos()
12 Início
13     Escreva("Informe os dados dos alunos");
14     Para cont De 1 Até 40 Passo 1 Faça
15         Escreva("Informe o nome do aluno " + cont);
16         Leia(alunos[cont].nome);
17         Escreva("Informe a nota 1:");
18         Leia(alunos[cont].nota1);
19         Escreva("Informe a nota 2:");
20         Leia(alunos[cont].nota2);
21         alunos[cont].media :=
22             (alunos[cont].nota1 + alunos[cont].nota2) / 2;
23     Fim_Para
24 Fim_Procedimento
  
```

```

24 Função CalcularMediaGeral() : Real
25 Início
26     notasTemp := 0;
27     Para cont De 1 Até 40 Passo 1 Faça
28         notasTemp := notasTemp + alunos[cont].media
29     Fim_Para
30     CalcularMediaGeral := notasTemp / 40;
31 Fim_Função

32 Função ContarAprovados() : Real
33 Início
34     qtdeAprovados := 0;
35     Para cont De 1 Até 40 Passo 1 Faça
36         Se(alunos[cont].media >= 7) Então
37             qtdeAprovados := qtdeAprovados + 1
38         Fim_Se
39     Fim_Para
40     ContarAprovados := qtdeAprovados;
41 Fim_Função

42 Início
43     LerAlunos();
44     Escreva("A média geral da turma é: " CalcularMediaGeral());
45     Escreva("A quantidade de alunos aprovados na turma é: "
46         + ContarAprovados());
47 Fim

```

O exemplo acima utiliza um vetor de registros para armazenar os dados dos alunos de uma turma. Os dados são lidos por um procedimento (declarado entre as linhas 11 e 23), pois não há necessidade de um valor de retorno. Entre as linhas 24 e 31, foi declarada uma função responsável por calcular a média geral da turma, sendo esta função chamada na linha 44, de modo a enviar o valor de retorno diretamente para a tela. Entre as linhas 32 e 41, foi declarada uma função capaz de contar a quantidade de alunos aprovados. Esta função, por sua vez, é chamada na linha 45, onde o valor de retorno também é enviado para a tela.

Escopo de Variáveis

O **escopo** de uma variável diz respeito à área do programa cuja variável irá existir. É a definição de sua criação e visibilidade dentro do programa.

Quanto ao escopo, as variáveis se dividem em: Variáveis Globais e Variáveis Locais.

Variáveis Globais

As variáveis utilizadas nos exemplos anteriores foram declaradas no nível do algoritmo (ou no nível do programa) e, por isso, são chamadas de variáveis globais. Elas estão acessíveis em qualquer ponto do algoritmo, e dentro de todas as sub-rotinas. Além disso, são criadas no início da execução do algoritmo e só serão destruídas quando o algoritmo for finalizado.

Perceba que no exemplo anterior as funções **CalcularMedia** e **ContarAprovados** utilizaram as variáveis **notasTemp** e **qtdeAprovados**, ambas declaradas no escopo do algoritmo, ou seja, variáveis globais.

Em muitos casos, porém, não é interessante que isso ocorra. Quando uma variável interessa apenas à execução de uma determinada sub-rotina, o melhor é declará-la como uma variável local.

Variáveis Locais

Como as sub-rotinas possuem uma estrutura própria, independente do corpo principal do algoritmo, é possível declarar variáveis pertencentes apenas a esta estrutura. Estas variáveis recebem o nome de **Variáveis Locais**, pelo fato de só existirem dentro da sub-rotina onde forem declaradas.

Ao contrário do que acontece com uma variável global, uma variável local só é acessível dentro da sub-rotina onde ela foi declarada.

Veja o exemplo:



```
{Exemplo de variáveis locais x globais}
01 Algoritmo ExemploEscopoVar
02 Variáveis
03     valorA, valorB : Inteiro;
04 Função EscreverVarLocal() : Inteiro
05 Variáveis
06     valorB, valorC : Inteiro;
07 Início
08     valorB := ValorA * 2;
09     Escreva(valorB);
10     valorC := 7;
11 Fim_Função
12 Início
13     valorA := 5;
14     valorB := 3;
15     Escreva(valorB) {escreve "3"}
16     Escreva(EscreverVarLocal); {escreve "10"}
17     Escreva(valorB) {escreve "3"}
18     Escreva(valorC) {gerará um erro, por valorC não existe aqui}
19 Fim
```

O bloco padrão de declaração de variáveis a que estamos acostumados é aquele em que são declaradas as variáveis globais. A declaração de variáveis locais é feita dentro da sub-rotina a que elas estarão subordinadas, conforme as linhas 5 e 6 do exemplo acima.

Veja que, no bloco de declaração de variáveis globais, são declaradas as variáveis **valorA** e **valorB**, enquanto que, dentro da função **EscreverVarLocal**, são declaradas as variáveis **valorB** e **valorC**. Então:

- **valorA** está disponível em qualquer lugar do algoritmo;
- **valorB** global estará disponível em qualquer lugar do algoritmo, exceto na função **EscreverVarLocal**, pois nesta existe uma variável local com o mesmo nome;
- **valorB** local estará disponível apenas dentro da função **EscreverVarLocal**;
- **valorC** estará disponível apenas dentro da função **EscreverVarLocal**;

Quando no corpo principal do algoritmo, escrevemos na tela o valor da variável **valorB**, acessamos a variável **valorB** global (linhas 15 e 17). Entretanto, quando dentro da função **EscreveVarLocal**, atribuímos e escrevemos a variável **valorB**, estamos acessando a variável **valorB** local (linhas 8 e 9). Além disso, o comando da linha 18 gerará um erro, pois a variável **valorC** só existe dentro do escopo da função **EscreveVarLocal**.

A utilização de variáveis locais pode trazer vários benefícios, entre eles o de reduzir o número de variáveis globais, tornando o programa mais limpo. Além disso, em termos de manutenção é bastante desagradável ter que ficar pesquisando onde uma variável global é utilizada antes de ter certeza de que se pode alterá-la sem comprometer o funcionamento do programa. Quando as variáveis que são utilizadas de forma temporária para a execução de uma sub-rotina são declaradas dentro de sua estrutura, tudo fica mais fácil de entender e de fazer manutenção.

É uma boa prática evitar a declaração de variáveis globais que são utilizadas apenas por uma sub-rotina. O ideal é definir como globais apenas as variáveis que sejam úteis ao longo da execução do algoritmo.

Outra vantagem relacionada ao uso de variáveis locais é a melhoria do gerenciamento de memória, pois as variáveis locais são criadas apenas quando a execução da sub-rotina é iniciada, e são destruídas ao término de sua execução.

Vejamos uma nova versão do algoritmo apresentado anteriormente como exemplo, agora fazendo uso de variáveis locais dentro das sub-rotinas:

```
{Exemplo funções com variáveis locais}
01 Algoritmo ExemploVarLocais
02 Tipos
03     RegAluno : Registro
04             nome : Caractere;
05             nota1, nota2, media : Real;
06             Fim_Registro
07 Variáveis
08     alunos : Vetor[1..40] De RegAluno;

09 Procedimento LerAlunos()
10 Variáveis
11     cont : Inteiro;
12 Início
13     Escreva("Informe os dados dos alunos");
14     Para cont De 1 Até 40 Passo 1 Faça
15         Escreva("Informe o nome do aluno " + cont);
16         Leia(alunos [cont].nome);
17         Escreva("Informe a nota 1:");
18         Leia(alunos [cont].nota1);
19         Escreva("Informe a nota 2:");
20         Leia(alunos [cont].nota2);
21         alunos [cont].media :=
22             (alunos [cont].nota1 + alunos [cont].nota2) / 2;
23     Fim_Para
24 Fim_Procedimento

25 Função CalcularMediaGeral() : Real
26 Variáveis
27     notasTemp : Real;
28     cont : Inteiro;
29 Início
30     notasTemp := 0;
31     Para cont De 1 Até 40 Passo 1 Faça
32         notasTemp := notasTemp + alunos [cont].media
33     Fim_Para
34     CalcularMediaGeral := notasTemp / 40;
35 Fim_Função
```

```

36 Função ContarAprovados() : Real
37 Variáveis
38     cont, qtdeAprovados : Inteiro;
39 Início
40     qtdeAprovados := 0;
41     Para cont De 1 Até 40 Passo 1 Faça
42         Se(alunos[cont].media >= 7) Então
43             qtdeAprovados := qtdeAprovados + 1
44         Fim_Se
45     Fim_Para
46     ContarAprovados := qtdeAprovados;
47 Fim_Função

48 Início
49     LerAlunos();
50     Escreva("A média geral da turma é: " CalcularMediaGeral());
51     Escreva("A quantidade de alunos aprovados na turma é: "
52                     + ContarAprovados());
53 Fim

```

Note que em cada sub-rotina, antes do comando **Início**, foi adicionado o bloco **Variáveis**, contendo as variáveis utilizadas pela sub-rotina. O vetor **alunos** continua sendo declarado com o escopo global, pois é utilizado ao longo do algoritmo.

Perceba que as três sub-rotinas do algoritmo utilizam uma variável contador (**cont**) para percorrer o vetor, porém, ainda assim, optou-se por declarar uma variável **cont** local dentro de cada sub-rotina, a declarar a variável como global. Isso parece fazer mais sentido, porque torna a sub-rotina mais independente, ao mesmo tempo em que previne possíveis erros.

Considerações sobre o Uso de Sub-rotinas

A utilização de sub-rotinas ajuda a diminuir a complexidade do algoritmo, pois com a criação de procedimentos/funções responsáveis por determinadas atividades, o corpo principal do algoritmo fica mais limpo e organizado.

As sub-rotinas permitem evitar a duplicação desnecessária de código, visto que são implementadas apenas uma vez e podem ser utilizadas em diversos locais do algoritmo. Imagine que em vários pontos do algoritmo seja necessário imprimir informações sobre um produto. Sem as sub-rotinas, seria necessário reescrever o mesmo bloco de comandos em vários lugares. Mas se criarmos uma sub-rotina, podemos simplesmente invocá-la em qualquer local do algoritmo quando a impressão do relatório se fizer necessária.

Outro aspecto importante está ligado à manutenção do programa. É comum os programas sofrerem mudanças ao longo do tempo, devido a vários fatores, mas um dos mais comuns é a modificação das necessidades dos usuários do sistema. Imaginando que nosso suposto relatório tivesse de ser modificado, se a implementação tivesse sido feita em uma sub-rotina, a mudança para o novo formato de relatório seria necessária em apenas um local. Por outro lado, se a implementação fosse repetida várias vezes ao longo do código, seria necessário alterar todos estes lugares para o novo formato de relatório, sem falar na possibilidade de esquecermos de modificar algum deles.

Uma sub-rotina não precisa necessariamente ser invocada a partir do bloco principal do algoritmo, ela também pode ser chamada a partir de outra sub-rotina. Não existe um limite sobre quantos níveis de chamadas podem ser realizados, ou seja, dentro da sub-rotina 1 podemos chamar a sub-rotina 2, que chama a sub-rotina 3, que chama a sub-rotina 4, e assim por diante. Todavia, cabe ao bom senso do programador evitar a construção de algoritmos demasiadamente complexos. Lembre-se de que as sub-rotinas existem para facilitar a nossa vida, e não para complicá-la.

A criação de algoritmos utilizando estruturas de controle (seleção, repetição e sub-rotinas), dá origem à chamada programação estruturada.

O Uso de Parâmetros

Muitas vezes é necessário que uma sub-rotina conheça um ou mais valores para poder realizar o seu processamento. Com o que foi visto até agora, a forma de se fazer isto seria criar uma variável global, atribuir-lhe o valor desejado e acessá-la de dentro da sub-rotina.

Embora isto funcione, geralmente não é a melhor forma de resolver o problema, pois traz uma série de desvantagens para o algoritmo.

Quando uma sub-rotina está acessando uma variável global, ela passa a ser dependente desta variável. Isso significa que, se a variável for renomeada ou removida, a sub-rotina deixará de funcionar. Então, cada vez que uma variável global for modificada, poderá ser necessário modificar várias sub-rotinas.

Outra desvantagem é que os algoritmos tornam-se mais complexos de se entender, de se encontrar problemas e de se incluir novas funcionalidades, pois, potencialmente, qualquer sub-rotina pode estar dependente de uma variável global, ao mesmo tempo em que qualquer sub-rotina pode alterar o valor de uma variável global. Assim, não é difícil imaginar que o algoritmo tende a se tornar cada vez mais desorganizado conforme cresce de tamanho.

Além disso, com o uso de variáveis globais dentro de sub-rotinas qualquer alteração no algoritmo tende a se tornar mais trabalhosa e complexa, o que se reflete em maior tempo e dinheiro gastos com manutenção.

Esperamos que neste ponto você já esteja convencido de que o acesso à variáveis globais dentro de sub-rotinas não é um bom negócio. Mas qual seria, então, a melhor solução para o problema? O uso de parâmetros.

Um **parâmetro** é uma informação passada para a sub-rotina a partir de sua chamada.

O uso de parâmetros é uma forma elegante e organizada de passar informações para uma sub-rotina, ao mesmo tempo em que a torna independente do restante do algoritmo. Com a passagem de parâmetros, fica explícito a todos os que resolverem utilizar a sub-rotina quais são os dados de que ela necessita para funcionar. Além disso, se uma variável global que estiver sendo passada como parâmetro a uma sub-rotina tiver seu nome modificado, bastará informar o novo nome na chamada à sub-rotina.

De um modo geral, o ideal é que as sub-rotinas recebam por parâmetro os valores de que necessitam para realizar seu processamento, e que as variáveis globais sejam prioritariamente acessadas no corpo principal do algoritmo.

Existem dois modos de passar parâmetros para uma sub-rotina: **Por Valor** ou **Por Referência**.

Passagem de Parâmetro por Valor

A passagem de parâmetro por valor entrega à sub-rotina uma cópia da variável passada como parâmetro, de modo que qualquer alteração no parâmetro não gerará impacto na variável original. A passagem por valor gera uma variável local, que recebe uma cópia do valor da variável passada por parâmetro, mas que é totalmente desvinculada desta.

Esta é a forma de passagem de parâmetro frequentemente utilizada.

A declaração de sub-rotinas utilizando passagem de parâmetros por valor é feita da seguinte forma:

```
<tipo_sub-rotina> <nome_sub-rotina> (<nome_parâmetro> : <tipo_parâmetro>)
```

Por exemplo:

```
Procedimento imprimeQuadrado(valor : Inteiro)
    Início
        Imprime(valor * valor);
    Fim_Função
```

Sendo que a chamada do procedimento seria feita assim:

```
imprimeQuadrado(3);
```

Veja outro exemplo:

```
{Exemplo de passagem de parâmetro por valor}
01 Algoritmo ConversorMoeda
02 Variáveis
03     cotDolar, cotEuro, cotPeso : Real;
04     valorReal, valorDolar, valorEuro, valorPeso : Real;

05 Função Converte(cotacao, valor : Real) : Real
06 Início
07     Converte := valor / cotacao;
08 Fim_Função

09 Início
10     Escreva("Digite a cotação do dia para... ");
11     Escreva("Dólar:");
12     Leia(cotDolar);
13     Escreva("Euro:");
14     Leia(cotEuro);
15     Escreva("Peso:");
16     Leia(cotPeso);

17 Repita
18     Escreva("-----");
19     Escreva("Digite o valor da compra em Real: ");
20     Leia(valorReal);

21     Se(valorReal > 0) Então
22         valorDolar := Converte(cotDolar, valorReal);
23         Escreva("Valor em Dólar: " + valorDolar);
24         valorEuro := Converte(cotEuro, valorReal);
25         Escreva("Valor em Euro: " + valor Euro);
26         valorPeso := Converte(cotPeso, valorReal);
27         Escreva("Valor em Peso: " + valorPeso);
28     Fim_Se
29     Até_Que(valorReal = 0)
30 Fim
```

108

Sub-rotinas

O algoritmo anterior serve para converter o valor de uma compra paga em real para dólar, euro e peso. Veja que a função responsável pela conversão, declarada entre as linhas 5 e 8, recebe por valor dois parâmetros do tipo real. Perceba que, como os dois parâmetros são do mesmo tipo, basta separá-los por vírgula e informar o tipo apenas uma vez.

Passagem de Parâmetro por Referência

A passagem de parâmetro por referência dá à sub-rotina acesso à variável original, de modo que qualquer alteração realizada no parâmetro estará sendo realizada diretamente na variável original, e as alterações persistirão mesmo após o término da execução da sub-rotina.

Neste caso, o nome do parâmetro definido dentro da sub-rotina atua como um apelido ou um segundo nome que dá acesso à variável.

A declaração de sub-rotinas, utilizando passagem de parâmetros por referência, é feita da seguinte forma:

```
<tipo_sub-rotina> <nome_sub-rotina> (Var <nome_parâmetro> : <tipo_parâmetro>)
```

Perceba que em termos de declaração, a única diferença da passagem por referência em relação à passagem por valor é a inclusão do comando **Var** antes do nome do parâmetro.

Por exemplo:

```
Procedimento adicionaTaxaServico(Var valor : Inteiro)
  Início
    valor := valor * 1,1;
  Fim_Procedimento
```

Sendo que a chamada da sub-rotina seria feita da mesma forma que chamaríamos uma sub-rotina que estivesse recebendo o parâmetro por valor:

```
adicionaTaxaServico(valorConta);
```

Podemos dizer que a passagem por referência é uma forma bem estruturada de permitir a uma sub-rotina alterar valores de variáveis globais, evitando os desagradáveis problemas que o acesso direto a estas variáveis iria trazer.

Veja outro exemplo:

```
{Exemplo de passagem de parâmetro por referência}
01 Algoritmo OrdenaVetor
02 Variáveis
03   vetNomes : Vetor[1..50] De Caractere;

04 Procedimento LerNomes(Var nomes : Vetor De Caractere)
05 Variáveis
06   cont: inteiro;
07 Início
08   Para cont De 1 Até 50 Passo 1 Faça
09     Escreva("Informe o valor " + cont);
10    Leia(nomes[cont]);
```

```

11     Fim_Procedimento
12
13 Procedimento OrdenarNomes (Var nomes : Vetor De Caractere)
14 Var
15     ia, ir : Inteiro;
16     aux : Caractere;
17 Início
18     Para ir De 1 Até 49 Passo 1 Faça
19         Para ia De ir+1 Até 50 Passo 1 Faça
20             Se(nomes[ir] > nomes[ia]) Então
21                 aux := nomes[ir];
22                 nomes[ir] := nomes[ia];
23                 nomes[ia] := aux;
24             Fim_Se
25         Fim_Para
26     Fim_Para
27 Fim_Procedimento
28
29 Procedimento ImprimirNomes (nomes : Vetor De Caractere)
30 Variáveis
31     cont: Inteiro;
32 Início
33     Para cont De 1 Até 50 Passo 1 Faça
34         Imprima(nomes[cont]);
35     Fim_Para
36 Fim_Procedimento
37
38 Início
39     LerNomes(vetNomes);
40     OrdenarNomes(vetNomes);
41     ImprimirNomes(vetNomes);
42 Fim

```

110

Sub-rotinas

O algoritmo anterior lê 50 nomes e os imprime em ordem alfabética. Veja que tanto a sub-rotina **LerNomes** (linhas 4 a 12), quanto a sub-rotina **OrdenarNomes** (linhas 13 a 27) recebem como parâmetro um vetor do tipo **Caractere**. Como ambas necessitam modificar o conteúdo do vetor, a passagem de parâmetro foi feita por referência. Por outro lado, como a sub-rotina **ImprimirNomes** (linhas 28 a 35) irá apenas ler o conteúdo do vetor, a passagem do parâmetro foi feita por valor.

Perceba, também, que quando um vetor é passado por parâmetro não se deve informar seu tamanho, mas apenas seu tipo.

Considerações sobre o Uso de Parâmetros

Os nomes utilizados nos parâmetros dentro das sub-rotinas podem ser iguais ou diferentes dos nomes das variáveis passadas por parâmetro no momento da chamada da sub-rotina.

Não existe uma regra quanto a isto, mas o ideal é que os nomes definidos dentro das sub-rotinas não sejam iguais aos de variáveis globais, para explicitar que estas não estão sendo utilizadas dentro da sub-rotina.

Na passagem por valor, podemos utilizar como parâmetro, na chamada da sub-rotina, variáveis, constantes ou valores fixos (um número, por exemplo). Já na passagem por referência, não é possível (e nem faria sentido) passar como parâmetro nada além de variáveis.

Também não podemos deixar de citar que é possível utilizar a passagem de parâmetro por valor e por referência em uma mesma sub-rotina.

Veja um exemplo:

```
{Exemplo de passagem de parâmetro por valor e referência na
mesma sub-rotina}
01 Algoritmo EnqueteCopa
02 Tipos
03     RegSelecao : Registro
04             nome : Caractere;
05             inicial : Caractere;
06             votos : Inteiro;
07         Fim_Registro
08 Variáveis
09     vetSelecoes : Vetor[1..3] De RegSelecao;
10     opcao : Caractere;

11 Procedimento Votar(Var vetSel : Vetor De RegSelecao; voto : Inteiro)
12 Início
13     vetSel[voto].votos := vetSel[voto].votos + 1;
14     Escreva("Voto computado para: " + vetSel[voto].nome);
15 Fim

16 Procedimento EmitirResultado(vetSel : Vetor De RegSelecao)
17 Var
18     cont : Inteiro;
19 Início
20     Para cont De 1 Até 3 Passo 1 Faça
21         Escreva("-----");
22         Escreva("Seleção: " + vetSel[cont].nome);
23         Escreva("Votos: " + vetSel[cont].votos);
24     Fim_Para
25 Fim_Procedimento

26 Procedimento Menu()
27 Início
28     Escreva("Que seleção ficará melhor colocada na próxima copa?");
29     Escreva("( 1 = Argentina | 2 = Brasil | 3 = Paraguai | 0 = Fim )");
30 Fim
```

```

31 Procedimento InicializarSelecoes(Var vetSel : Vetor de RegSelecao);
32 Início
33     vetSel[1].nome := "Argentina";
34     vetSel[1].votos := 0;
35     vetSel[2].nome := "Brasil";
36     vetSel[2].votos := 0;
37     vetSel[3].nome := "Paraguai";
38     vetSel[3].votos := 0;
39 Fim

40 Início
41     Repita
42         Menu();
43         Leia(opcao);
44         Se((opcao >= 1) .E. (opcao <= 3)) Então
45             Votar(vetSelecoes, opcao);
46         Senão
47             Se(opcao <> 0) Então
48                 Escreva("Voto inválido.");
49                 Fim_Se
50             Fim_Se
51             Até_Que(opcao = 0)
52             Escreva("Fim da Votação");
53             EmitirResultado(vetSelecoes);
54 Fim

```

112

Sub-rotinas

O algoritmo acima permite a realização de uma enquete, computando votos até que seja informado o valor 0 (zero) para sair, quando, então, é exibido o resultado da votação.

Perceba que o procedimento **Votar** (linhas 11 a 15) recebe como parâmetro o vetor contendo o resultado parcial da votação e também a opção escolhida no voto atual. Como é necessário que o procedimento modifique o vetor, a passagem foi feita por referência. Já a opção de voto, que é apenas lida, foi passada por valor. Note que, para separar parâmetros de tipos diferentes, utilizamos o ponto e vírgula (;

Atividades

Todos os exercícios a seguir devem ser implementados em pseudocódigo. Utilize sub-rotinas na resolução dos exercícios e atente-se para a melhor opção de escopo para cada variável.

- Crie um algoritmo que calcule X^Y (X elevado a Y), sendo que os valores de X e Y devem ser números inteiros, positivos, informados pelo usuário. O resultado deve ser exibido na tela.

- 2) Construa um algoritmo que permita a consulta ao acervo de uma biblioteca. Para isso, o sistema deve:
- Ler os dados de 500 livros (código, título, autor, ano, estante e prateleira).
 - Entrar no modo consulta, onde o livro é pesquisado pelo título, sendo exibidas na tela todas as informações sobre a obra. Se o nome pesquisado for vazio, o programa encerra.
- 3) Crie um algoritmo que seja capaz de realizar as seguintes atividades:
- Ler os dados de uma turma de 40 alunos.
 - Calcular as médias de todos os alunos.
 - Gerar um relatório na impressora com os alunos aprovados ($\text{média} \geq 7$), listados em ordem alfabética.
 - Gerar um relatório na impressora com os alunos em exame ($\text{média} \geq 4$ e < 7), listados em ordem alfabética.
 - Gerar um relatório na impressora com os alunos reprovados ($\text{média} < 4$), listados em ordem alfabética.
 - Exibir na tela a maior e a menor média da turma, bem como a média geral.
- 4) Desenvolva um algoritmo para gerar uma lista de preços para uso dos vendedores de uma loja X. Seu algoritmo deverá contemplar as seguintes funcionalidades:
- A lista de produtos poderá ter até 100 itens. Caso seja informado um produto com código 0 (zero) antes de se chegar aos 100 itens, o programa deverá entender que os itens acabaram.
 - Para cada produto, deve-se solicitar código, nome e preço de custo.
 - Devem-se calcular os preços de venda conforme as regras a seguir:
 - Venda à vista: 20% de margem de lucro;
 - Venda com pagamento em 30 dias: 25% de margem de lucro;
 - Pagamento em 3x: 30% de margem de lucro.
 - A lista de preços deverá ser impressa e conter o nome e as três opções de preço de venda de todos os produtos, ordenados pelo código.

Nos exercícios a seguir, atente para a melhor opção de **passagem de parâmetro** para cada caso.

- 5) Crie um algoritmo que leia um número de 1 a 12 e retorne o nome do mês correspondente.
- 6) Desenvolva um algoritmo que contabilize a quantidade de vagas existentes em uma escola e a quantidade de alunos matriculados. Para isso, o programa deverá realizar as seguintes atividades:
 - a. Para cada turma, solicitar as seguintes informações: número da turma, capacidade máxima, quantidade de matriculados.
 - b. A capacidade máxima que o programa deve considerar é de 100 turmas. Deverá ser possível indicar o término das turmas, informando o valor 0 (zero) no código da turma.
 - c. Ao final, o programa deverá imprimir uma relação de todas as turmas, com sua capacidade máxima e sua capacidade ocupada. Além disso, deverá imprimir a quantidade total de vagas que a escola disponibiliza, bem como a quantidade total de alunos matriculados no momento.
- 7) Crie um algoritmo que simule uma calculadora, permitindo o uso das quatro operações básicas (multiplicação, divisão, adição e subtração). O funcionamento do programa deverá ser o seguinte:
 - a. Lê o primeiro valor.
 - b. Lê a operação desejada pelo usuário.
 - c. Lê o segundo valor.
 - d. Apresenta o resultado do cálculo e encerra o programa.
- 8) Crie um algoritmo que permita manter um cadastro de cidades, com as seguintes funcionalidades:
 - a. O limite de cidade cadastradas deve ser de 100.
 - b. Para cada cidade é necessário cadastrar: nome, estado, quantidade de habitantes, PIB e DDD.
 - c. A qualquer momento deve ser possível escolher entre cadastrar uma nova cidade ou pesquisar uma cidade existente. Para isso, deve ser exibido um menu ao usuário.
 - d. A pesquisa de cidades deverá ser feita por nome.
 - e. Deve haver uma opção no menu para sair do programa.

Introdução à Programação

Até este ponto, nós aprofundamos nossos estudos no aprendizado da lógica de programação. Embora os caminhos para o aprendizado da programação sejam muitos, estamos convencidos de que este é o ponto de partida ideal para a formação de um bom profissional da área de desenvolvimento de *software*. Porém, este é apenas o primeiro passo.

Além de conhecer a lógica de programação, o desenvolvedor precisa aprender uma linguagem de programação, conhecer um ambiente de desenvolvimento e, por fim, os **frameworks** e bibliotecas relacionados à tecnologia escolhida (caso existam).

Frameworks:

É um conjunto de objetos que colaboram para realizar um conjunto de responsabilidades para um domínio de aplicações de subsistemas.

O escopo deste livro se resume à lógica de programação e a uma introdução à linguagem Pascal, que é uma linguagem acadêmica, simples, porém completa o suficiente para servir como um excelente ponto de partida para que se aprendam outras linguagens, mais focadas ao mercado.

Entretanto, antes de iniciarmos o trabalho com a linguagem Pascal, é importante que você compreenda mais alguns conceitos.

Linguagem de Máquina

A **linguagem de máquina**, também chamada de linguagem binária, é a única linguagem que o computador entende. Em outras palavras, o computador somente consegue interpretar e executar instruções que estejam escritas nesta linguagem. Então, qualquer *software*, para ser executado, precisa, obrigatoriamente, ser convertido em linguagem de máquina.

A maneira como esta conversão de código fonte em linguagem de máquina será feita, dependerá da linguagem de programação em que o programa foi escrito.

Linguagem de Programação

As **linguagens de programação** são utilizadas para descrever algoritmos, de modo que os comandos que compõem esses algoritmos possam ser posteriormente convertidos em linguagem de máquina.

Cada linguagem possui seu próprio conjunto de comandos e padrões, mas muitos deles são parecidos ou, até mesmo, idênticos entre as diferentes linguagens. O mais importante é que a lógica de programação, quando aprendida, pode ser aplicada a qualquer linguagem.

As linguagens de programação evitam que os programadores tenham que escrever seus programas em linguagem de máquina, o que seria muito mais trabalhoso, lento e complexo. Na verdade, não conseguiríamos ter chegado ao nível de complexidade nos *softwares* que temos hoje, se tivéssemos que trabalhar somente com a linguagem de máquina.

Existem basicamente dois tipos de linguagens de programação: as compiladas e as interpretadas.

Linguagens Compiladas

As **linguagens de programação compiladas** são aquelas que se utilizam de um programa chamado Compilador para converter o código fonte em linguagem de máquina. De forma simples, podemos dizer que os compiladores atuam como tradutores, que leem os comandos escritos em uma determinada linguagem de programação e os escrevem em linguagem de máquina.

Neste processo, o código convertido fica gravado na forma de um programa executável (armazenado em um ou mais arquivos). Um programa compilado, portanto, pode ser executado inúmeras vezes sem que seja necessário realizar novamente o processo de compilação.

O Pascal é uma linguagem de programação compilada.

Linguagens Interpretadas

Nas linguagens interpretadas, não existe a criação de um programa executável como ocorre nas linguagens compiladas.

Nestas linguagens, os programas são diretamente executados pelos interpretadores, que vão convertendo os comandos de código fonte em linguagem de máquina, à medida em que o programa é executado.

Como não é gerado um programa executável gravado em disco, a execução do programa é feita sempre pelo interpretador.

Cada tipo de linguagem tem suas vantagens e desvantagens e o aprofundamento desta questão foge ao escopo deste livro. A seu tempo, você saberá escolher a melhor opção para cada caso.



Atividades

- 1) Pesquise pelo menos três linguagens de programação compiladas e descreva em que tipo de soluções elas geralmente são utilizadas.
- 2) Pesquise pelo menos três linguagens de programação interpretadas e descreva em que tipo de soluções elas geralmente são utilizadas.
- 3) Pesquise as principais vantagens e desvantagens das linguagens de programação compiladas.
- 4) Pesquise as principais vantagens e desvantagens das linguagens de programação interpretadas.
- 5) Descubra quais são as linguagens de programação mais utilizadas atualmente no mercado de desenvolvimento de *software*, e pesquise quais são as principais características destas linguagens.
- 6) Pesquise o que é uma IDE e quais são as suas contribuições para melhorar a vida do programador. Além disso, descubra quais as IDEs mais utilizadas atualmente e com que linguagens elas são utilizadas.
- 7) Descubra se pode haver diferença no salário de um programador dependendo do tipo de linguagem de programação com que ele trabalha. Caso positivo, reflita sobre os motivos desta diferença, e levante quais as tecnologias de desenvolvimento de *software* mais valorizadas atualmente pelo mercado.
- 8) Pesquise quais as tecnologias mais utilizadas pelas empresas de desenvolvimento de *software* da cidade em que você vive.

Linguagem Pascal

Fundamentos

Iniciaremos agora o estudo da linguagem de programação Pascal. Nossa abordagem não será profunda a ponto de explicitar todos os recursos da linguagem, pois o intuito é apresentar apenas os recursos equivalentes ao conteúdo estudado ao longo do livro, para que você possa transformar seus algoritmos em programas executáveis.

O Pascal é uma linguagem de programação de grande adoção no mundo acadêmico, especialmente por ser muito bem estruturada e permitir o aprendizado de todos os fundamentos necessários ao iniciante no mundo da programação. Favorece o aprendizado da lógica de programação, bem como a adoção de boas práticas de desenvolvimento de *software*, permitindo a criação de programas com códigos simples e de fácil compreensão.

Considere que o aprendizado da linguagem Pascal serve, antes de tudo, para a compreensão de fundamentos, o que é mais importante do que o aprendizado da linguagem em si. Linguagens de programação existem inúmeras, mas os fundamentos da lógica de programação se aplicam a praticamente todas elas. Uma vez que o estudante tenha compreendido estes fundamentos, ele estará mais preparado para aprender e produzir de forma satisfatória em qualquer tecnologia de desenvolvimento de *software*.

É importante ter em mente que, na grande maioria dos casos, um programador não utilizará apenas uma ou duas linguagens ao longo de sua vida profissional. As tecnologias mudam rápido, e quem quiser ser valorizado no mercado tem de se atualizar e estar sempre aprendendo. Por isso, os fundamentos são tão importantes, eles permitem que a pessoa tenha mais facilidade em aprender outras linguagens e tecnologias por conta própria, à medida que isso se fizer necessário. Nem sempre haverá um professor ao seu lado, pense nisso!

Para que você possa começar a programar com mais facilidade, o Apêndice 2 traz um guia para configuração de um ambiente de estudo de programação com Pascal, abordando os sistemas Windows e Linux.

Estrutura de um Programa Pascal

A estrutura básica de um programa Pascal segue o mesmo padrão que estudamos no pseudocódigo. Na verdade, durante nosso estudo utilizando pseudocódigo, nós adotamos propositalmente vários dos padrões utilizados pela linguagem Pascal, sendo que muitos também são adotados por outras linguagens.

A estrutura de um programa Pascal é a seguinte:

1. Cabeçalho contendo o nome do programa;
2. Importação de bibliotecas;
3. Declaração de constantes;
4. Declaração de tipos;
5. Declaração de variáveis;
6. Declaração de sub-rotinas;
7. Corpo principal do programa.

Embora alguns dos blocos de declaração possam ter sua posição trocada, esta é a sequência indicada para evitar erros que podem ocorrer em algumas situações.

Tipos de Dados

Os tipos de dados do Pascal equivalentes aos que estudamos no pseudocódigo são:

Tipo de dado no Pascal	Equivalência no Pseudocódigo
Integer	Inteiro
Real	Real
Boolean	Lógico
String	Caractere

*Existem outros tipos de dados no Pascal que não serão abordados neste contexto.

119

Linguagem Pascal

Declaração de Variáveis e Constantes

No Pascal, a declaração de variáveis segue o mesmo padrão estudado no pseudocódigo, porém o nome do bloco de declarações, ao invés de **Variáveis** é denominado **var**. O mesmo ocorre com as constantes, que, ao invés de **Constantes**, o Pascal adota a denominação **const** para o bloco de declaração.

O Pascal não é uma linguagem *case sensitive*, ou seja, não diferencia maiúsculas de minúsculas. Então é possível declarar uma variável chamada **valor** e acessá-la utilizando **Valor**, por exemplo. Mesmo assim, recomenda-se adotar o mesmo padrão de nomenclatura utilizado no pseudocódigo.

Também é possível escrever os comandos da linguagem em maiúsculas ou minúsculas, porém para uma melhor legibilidade do código, é interessante que utilizem-se sempre letras minúsculas ou, então, maiúsculas apenas na primeira letra do comando. Nos exemplos apresentados, utilizaremos sempre letras minúsculas para nomear os comandos.

Operadores

Operadores Aritméticos

A única diferença em relação ao que vimos no pseudocódigo é que em Pascal não há um operador de potenciação. Esta é feita a partir de funções predefinidas no Pascal.

Precedência	Operador	Descrição
1	*	Multiplicação
1	/	Divisão
2	+	Adição
2	-	Subtração

Operadores Relacionais

No Pascal, os operadores relacionais são exatamente iguais ao que estudamos no pseudocódigo:

Operador	Descrição
=	Igual
<>	Diferente
<	Menor
<=	Menor ou Igual
>	Maior
>=	Maior ou Igual

120

Operadores Lógicos

No Pascal, os operadores lógicos são representados da seguinte forma:

Precedência	Pascal	Pseudocódigo
1	NOT	.NÃO.
2	AND	.E.
3	OR	.OU.

Operador Literal

No Pascal, o operador literal, utilizado na concatenação de *strings*, não converte o valor à direita para *string*, caso ele seja de outro tipo. A linguagem oferece funções de conversão para atender a esta demanda.

Operador	Descrição
+	Concatenação

Estruturas de Seleção

No pseudocódigo, representamos as estruturas de seleção por meio dos comandos **Se...Então...Senão...Fim_Se**. No Pascal, estas estruturas são representadas pelos comandos **If...Then...Else**. Veja:

```
if(<condição>) then
begin
    <instruções>
end
else
begin
    <instruções>
end;
```

Embora não seja necessário delimitar o bloco de comandos com **begin...end** quando este for composto por apenas uma instrução, esta ação é sempre recomendada, pois ajuda a manter o código legível e a prevenir futuros erros.

Perceba que existe um ponto e vírgula após o último **end**.

Primeiro Exemplo em Pascal

Após conhecer alguns dos fundamentos da linguagem Pascal, vamos ao nosso primeiro exemplo (este é o mesmo exemplo apresentado em pseudocódigo no capítulo 4, que trata das estruturas de controle de seleção):

```
01 {Primeiro programa de exemplo em Pascal}
02 program PrimeiroExemplo;
03     uses crt;
04 var
05     nota1, nota2, media : real;
06     nome : string[30];
07     resultado : string[10];
08
09 begin
10     clrscr;
11     writeln('Digite o nome do aluno: ');
12     read(nome);
13     writeln('Digite a primeira nota: ');
14     read(nota1);
15     writeln('Digite a segunda nota: ');
16     read(nota2);
17
18     media := (nota1 + nota2) / 2;
19
```

```

20  if(media >= 7) then
21      begin
22          resultado := 'aprovado';
23      end
24  else
25      begin
26          if(media >= 4) then
27              begin
28                  resultado := 'em exame';
29              end
30          else
31              begin
32                  resultado := 'reprovado';
33              end
34          end;
35
36      writeln('O aluno ' + nome + ' está ' + resultado);
37      readkey;
38  end.

```

Ao invés de **Início** e **Fim**, o Pascal utiliza **begin** e **end** para delimitar os blocos. O último **end** do programa, indicador do fechamento do corpo principal, deve ser seguido de um ponto final (.).

Quando utilizamos o tipo *string*, é importante definir o tamanho máximo de caracteres que a variável deverá ser capaz de armazenar, conforme foi feito nas linhas 6 e 7 do exemplo. É possível declarar uma variável do tipo *string* sem definir o tamanho máximo, mas, neste modo, estaremos desperdiçando espaço, pois o Pascal reservará o tamanho máximo permitido para uma *string*, que é de 255 caracteres.

Perceba também que a manipulação de valores do tipo *string* é feita utilizando-se aspas simples (' ') e não aspas duplas (" ") como feito no pseudocódigo.

Outra diferença está no tipo **Real**, cujo separador de casas decimais é o ponto (.) e não a vírgula, como estamos acostumados. No Pascal, não se utiliza separador de milhar.

O comando **clrscr** (linha 10) serve para limpar a tela, é geralmente usado no início do programa. Além disso, para que a tela do programa continue sendo exibida ao final da execução, é necessário incluirmos o comando **readkey** (linha 37) antes do fechamento do corpo principal – este comando faz com que o programa aguarde o pressionamento de uma tecla antes de encerrar sua execução. Para que os comandos **clrscr** e **readkey** funcionem, é necessário importar no início do programa a biblioteca **crt**, o que é feito por meio do comando **uses crt**, na linha 3 do exemplo.

Para ler informações via teclado utilizamos os comandos **read()** ou **readln()**, equivalentes ao comando **Leia()** do pseudocódigo.

No pseudocódigo, para escrever na tela utilizamos o comando **Escreva()**, enquanto que no Pascal também existem dois comando para esta função: **write** e **writeln**. A diferença entre eles, é que **writeln** gera uma quebra de linha após a escrita do valor, de modo que a próxima informação a ser escrita na tela não fique ao lado desta que está sendo escrita no momento.

Reproduzindo no Computador

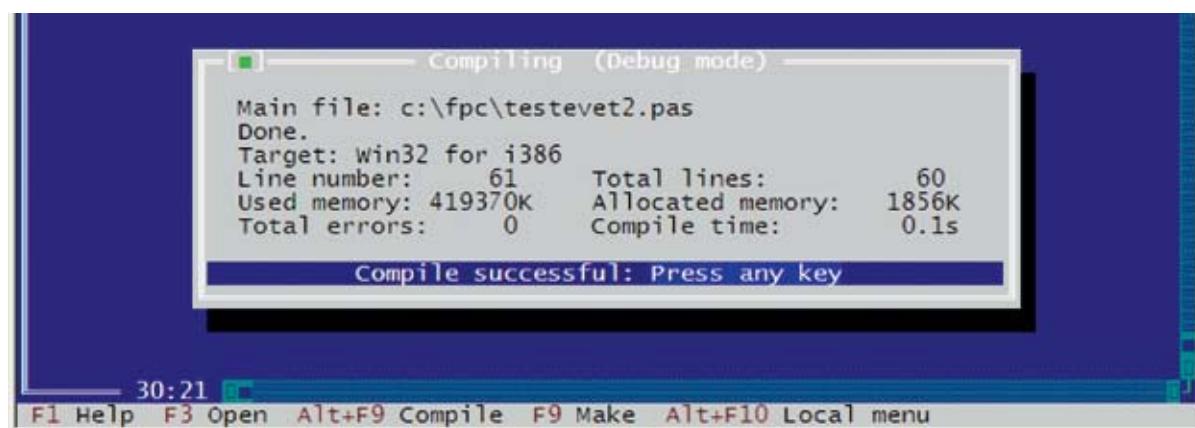
Para que você consiga reproduzir o exemplo anterior no computador, é necessário que esteja com o ambiente de desenvolvimento instalado (compilador + IDE).

A partir daí, o primeiro passo é abrir a IDE por meio do atalho criado durante a instalação e digitar o código fonte. Para salvar, pressione **F2**.

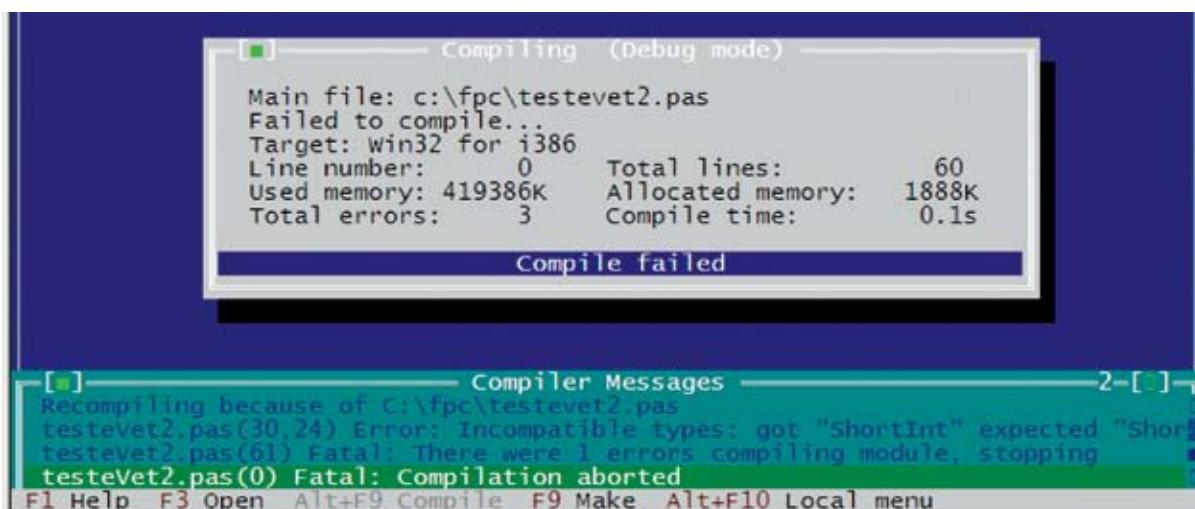
Todo arquivo de código fonte do Pascal é salvo por padrão com a extensão “.pas”. Não existe uma regra a respeito do nome do arquivo, porém é interessante utilizar o mesmo nome do programa definido no código fonte. Neste caso, nomeie seu arquivo como **PrimeiroExemplo.pas**.

Para compilar o programa, existem duas formas: pela linha de comando ou pela IDE. A compilação pela IDE é mais rápida e prática. Para compilar o programa pela IDE, selecione a opção copile do menu compile ou pressione **ALT+F9**. A IDE irá exibir uma janela com o resultado da compilação.

Caso a compilação tenha ocorrido sem problema algum, a janela exibida será parecida com a imagem abaixo, bastando pressionar qualquer tecla para fechá-la:



Caso tenha ocorrido algum erro de compilação, a janela de resultado da compilação irá exibir a quantidade de erros ocorridos, e será exibida uma outra janela contendo a lista de erros, com o número da linha e da coluna em que eles se encontram, conforme este exemplo:



IDE (Integrated Development Environment):

Um ambiente integrado para desenvolvimento de software.

Uma vez que o programa tenha sido compilado sem erros, é possível executá-lo pela IDE pressionando **Ctrl+F9**.

Também seria possível rodar o programa via linha de comando ou, ainda, por um atalho, porém, durante o processo de desenvolvimento, estas opções são menos práticas.

Estruturas de Repetição

O Pascal possui os três tipos de repetição que estudamos no pseudocódigo: Repetição com Pré-Teste, Repetição com Pós-Teste e Repetição com Variável de Controle.

Nas repetições do tipo Pré-Teste e Variável de Controle, sempre utilizaremos os demarcadores de bloco **begin** e **end**.

Repetição com Pré-Teste

No pseudocódigo, a repetição com Pré-Teste era representada pelo comando **Enquanto...Faça...Fim_Enquanto**. No Pascal, o comando para representar esta operação é **while...do**. Veja um exemplo:

```
1 {Exemplo de repetição do tipo Pré-Teste}
2 program ExemploPreTeste;
3   uses crt;
4 var
5   contador : integer;
6   nome : string[30];
7 begin
8   clrscr;
9   writeln('Digite seu nome: ');
10  read(nome);
11  contador := 1;
12
13  while(contador <= 10) do
14  begin
15    if(contador <= 5) then
16      begin
17        writeln(nome);
18      end
19    else
20      begin
21        writeln('[' + nome + ']');
22      end;
23
24    contador := contador + 1;
25  end;
26
27  readkey;
28 end.
```

Repetição com Pós-Teste

A repetição do tipo Pós-Teste é representada no pseudocódigo pelo comando **Repita...Até_Que**. No Pascal, este tipo de repetição é representado pelo comando **repeat...until**. Confira o exemplo:

```
01 {Exemplo de repetição do tipo Pós-Teste}
02 program ExemploPostTeste;
03     uses crt;
04 var
05     nota1, nota2, media : real;
06     resultado : string[10];
07     continuar : string[1];
08 begin
09     repeat
10         clrscr;
11         writeln('Digite a primeira nota: ');
12         readln(nota1);
13         writeln('Digite a segunda nota: ');
14         readln(nota2);
15
16         media := (nota1 + nota2) / 2;
17         if(media >= 7) then
18             begin
19                 resultado := 'aprovado';
20             end
21         else
22             begin
23                 if(media >= 4) then
24                     begin
25                         resultado := 'exame';
26                     end
27                 else
28                     begin
29                         resultado := 'reprovado';
30                     end
31             end;
32
33         write('A média é: ');
34         write(media:4:2);
35         writeln(' [' + resultado + ']');
36
37         repeat
38             write('Calcular outra média? (S/N)');
39             readln(continuar);
40             continuar := UpCase(continuar);
41             until((continuar = 'S') or (continuar = 'N'));
42
43         until(continuar = 'N');
44         writeln('Fim.');
45         readkey;
46     end.
```

Perceba que na linha 34, cuja média é exibida na tela, estamos utilizando **media:4:2**, a fim de formatar o valor para exibição na tela. O formato **:4:2** significa que desejamos preparar o valor para ser exibido em 4 dígitos, sendo 2 deles como fração.

Entre as linhas 37 e 41, criamos uma segunda repetição do tipo Pós-Teste, a fim de garantir que o usuário informe uma das opções válidas.

Na linha 40, utilizamos a função **UpCase** para transformar para maiúscula a entrada do usuário. Esta é uma das muitas funções predefinidas que o Pascal incorpora, a fim de facilitar a vida do programador. Contudo, o estudo destas funções está fora do nosso escopo neste livro.

Se for do seu interesse, você pode encontrar mais informações sobre as funções predefinidas do Pascal nas referências citadas no final do livro.

Repetição com Variável de Controle

O tipo de repetição baseado em variável de controle é representado no Pascal pelo comando **for...to...do**, com uma estrutura semelhante ao comando **Para...De...Até...Passo**, que utilizamos no pseudocódigo. A principal diferença é que no Pascal não é necessário informar o tamanho do passo, sendo o incremento feito automaticamente de 1 em 1 unidade. Veja o exemplo:

```
01 {Exemplo de repetição do tipo variável de controle }
02 program ExemploVarControle;
03   uses crt;
04 var
05   contador : integer;
06   tempMes, tempAnual : Real;
07 begin
08   clrscr;
09   writeln('Cálculo de temperatura média anual');
10   tempAnual := 0;
11
12   for contador := 1 to 12 do
13   begin
14     write('Digite a temperatura média do mês ');
15     writeln(contador);
16     read(tempMes);
17
18     if(tempMes > 35) then
19       begin
20         writeln('Que calor!');
21       end
22     else
23       begin
24         if(tempMes < 5) then
25           begin
26             writeln('Que frio!');
27           end
28       end;
29
```

```
30      tempAnual := tempAnual + tempMes;
31  end;
32
33  tempAnual := tempAnual / 12;
34  write('A temperatura média anual foi ');
35  write(tempAnual:4:2);
36  readkey;
37 end.
```

Estruturas de Dados

Vetores

No Pascal, os vetores são denominados *arrays* unidimensionais, e sua declaração e uso são bastante parecidos com o padrão utilizado no pseudocódigo. Veja o exemplo:

```
01 {Exemplo de uso de vetor}
02 program ExemploVetor;
03   uses crt;
04 var
05   valores : array[1..10] of integer;
06   contador : integer;
07 begin
08   clrscr;
09   for contador := 1 to 10 do
10   begin
11     write('Digite um valor para a posição ');
12     writeln(contador);
13     read(valores[contador]);
14   end;
15
16   for contador := 1 to 10 do
17   begin
18     write('O elemento da posição ');
19     write(contador);
20     write(' vale ');
21     writeln(valores[contador]);
22   end;
23
24   readkey;
25 end.
```

Matrizes

As matrizes também são utilizadas no Pascal de forma semelhante às do pseudocódigo, sendo denominadas, neste caso, *arrays multidimensionais*. Confira o exemplo:

```
01 {Exemplo de uso de matriz}
02 program ExemploMatriz;
03   uses crt;
04 var
05   nomeDespesas : array[1..4] of string[30];
06   valorDespesas : array[1..12, 1..4] of real;
07   i, j, ano : integer;
08   somatorio, totalAnual : real;
09 begin
10   clrscr;
11   nomeDespesas[1] := 'Aluguel';
12   nomeDespesas[2] := 'Condomínio';
13   nomeDespesas[3] := 'Energia Elétrica';
14   nomeDespesas[4] := 'Telefone e Internet';
15
16   writeln('Cálculo de despesas anuais.');
17   writeln('Informe o ano: ');
18   read(ano);
19
20   for i := 1 to 12 do
21   begin
22     for j := 1 to 4 do
23     begin
24       write('Informe o gasto com ' + nomeDespesas[j]
25             + ' no mês ');
26       writeln(i);
27       read(valorDespesas[i, j]);
28     end
29   end;
30
31   write('Relatório de gastos com moradia no ano de ');
32   writeln(ano);
33   totalAnual := 0;
34   for i := 1 to 4 do
35   begin
36     somatorio := 0;
37     for j := 1 to 12 do
38     begin
39       somatorio := somatorio + valorDespesas[j, i];
40     end;
41   end;
```

128

```

42     write('O total gasto com ' + nomeDespesas[i]);
43     write(' ao longo do ano foi ');
44     writeln(somatorio:7:2);
45     totalAnual := totalAnual + somatorio;
46   end;
47
48   write('O total gasto no ano foi ');
49   writeln(totalAnual:7:2);
50   write('A média mensal de gastos foi ');
51   writeln((totalAnual / 12):7:2);
52   readkey;
53 end.

```

Registros

O Pascal também permite a criação de registros de uma forma bastante parecida com a que estudamos no pseudocódigo. Veja o exemplo:

```

01 {Exemplo de uso de registro}
02 program ExemploRegistro;
03   uses crt;
04 type
05   RegAluno = record
06     nome : string[30];
07     nota1 : real;
08     nota2 : real;
09   end;
10 var
11   alunos : array[1..40] of RegAluno;
12   mediaTemp : real;
13   cont : integer;
14 begin
15   clrscr;
16   for cont := 1 to 40 do
17   begin
18     write('Informe o nome do aluno ');
19     writeln(cont);
20     readln(alunos[cont].nome);
21
22     write('Informe a nota 1 do aluno ');
23     writeln(cont);
24     readln(alunos[cont].nota1);
25

```

```

26     write('Informe a nota 2 do aluno ');
27     writeln(cont);
28     readln(alunos[cont].nota2);
29 end;
30
31 for cont := 1 to 40 do
32 begin
33     write('O nome do aluno ');
34     write(cont);
35     writeln(' , ' + alunos[cont].nome);
36
37 mediaTemp := (alunos[cont].nota1 + alunos[cont].nota2) / 2;
38 write('A média do aluno ');
39 write(cont);
40 write(' , ');
41 writeln(mediaTemp:4:2);
42 end;
43
44 readkey;
45 end.

```

130

Sub-rotinas

No Pascal, a definição de sub-rotinas também é feita de maneira muito próxima ao modo como trabalhamos no pseudocódigo. Basicamente, o que muda são os comandos utilizados para declarar as sub-rotinas. Além disso, o escopo de variáveis e a utilização de parâmetros seguem os mesmos padrões já estudados.

Funções

Para declarar funções no pseudocódigo, utilizamos o comando **Função**, ao passo que no Pascal o comando utilizado é **function**. Confira o exemplo:

```

01 {Exemplo de uso de função}
02 program ExemploFuncao;
03   uses crt;
04 var
05   cotDolar, cotEuro, cotPeso : real;
06   valorReal, valorDolar, valorEuro, valorPeso : real;
07
08 function Converte(cotação, valor : real) : real;
09 begin
10   Converte := valor / cotação;
11 end;
12

```

```

13 begin
14     clrscr;
15     writeln('Digite a cotação do dia para... ');
16     writeln('Dólar: ');
17     read(cotDolar);
18     writeln('Euro: ');
19     read(cotEuro);
20     writeln('Peso: ');
21     read(cotPeso);
22
23 repeat
24     writeln('-----');
25     writeln('Digite o valor da compra em Real:');
26     read(valorReal);
27
28     if(valorReal > 0) then
29     begin
30         valorDolar := Converte(cotDolar, valorReal);
31         write('Valor em Dólar: ');
32         writeln(valorDolar:7:2);
33
34         valorEuro := Converte(cotEuro, valorReal);
35         write('Valor em Euro: ');
36         writeln(valorEuro:7:2);
37
38         valorPeso := Converte(cotPeso, valorReal);
39         write('Valor em Peso: ');
40         writeln(valorPeso:7:2);
41     end;
42     until(valorReal = 0);
43     readkey;
44 end.

```

Procedimentos

No Pascal, os procedimentos são declarados pelo comando **procedure**, que é equivalente ao comando **Procedimento**, utilizado no pseudocódigo.

Veja um exemplo:

```

01 {Exemplo de uso de procedimento}
02 program ExemploProcedimento;
03     uses crt;
04 type
05     RegSelecao = record
06         nome : string[30];
07         votos : integer;
08     end;
09

```

```
10     vetRegistros = array[1..3] of RegSelecao;
11 var
12     vetSelecoes : vetRegistros;
13     opcao : integer;
14
15 procedure Votar(var vetSel : vetRegistros; voto : integer);
16 begin
17     vetSel[voto].votos := vetSel[voto].votos + 1;
18     writeln('Voto computado para: ' + vetSel[voto].nome);
19 end;
20
21 procedure EmitirResultado(vetSel : vetRegistros);
22 var
23     cont : integer;
24 begin
25     for cont := 1 to 3 do
26         begin
27             writeln('-----');
28             writeln('Seleção: ' + vetSel[cont].nome);
29             write('Votos: ');
30             writeln(vetSel[cont].votos);
31         end;
32     end;
33
34 procedure InicializarSelecoes(var vetSel : vetRegistros);
35 begin
36     vetSel[1].nome := 'Argentina';
37     vetSel[1].votos := 0;
38     vetSel[2].nome := 'Brasil';
39     vetSel[2].votos := 0;
40     vetSel[3].nome := 'Paraguai';
41     vetSel[3].votos := 0;
42 end;
43
44 procedure Menu();
45 begin
46     writeln('-----');
47     writeln('Que seleção ficará melhor colocada na próxima copa?');
48     writeln('1 = Argentina | 2 = Brasil | 3 = Paraguai | 0 = Fim');
49 end;
50
51 begin
52     clrscr;
53     InicializarSelecoes(vetSelecoes);
54     repeat
55         Menu();
56         readln(opcao);
57         if((opcao >= 1) and (opcao <= 3)) then
58             begin
59                 Votar(vetSelecoes, opcao);
60             end
61     until (opcao = 0);
```

```

61     else
62         begin
63             if(opção <> 0) then
64                 begin
65                     writeln('Voto inválido.');
66                 end;
67             end;
68
69         until(opcao = 0);
70         writeln('Fim da votação');
71         EmitirResultado(vetSelecoes);
72         readkey;
73     end.

```



Atividades

- 1)** Nas atividades de 1 a 3 do capítulo 4, você criou 3 algoritmos em pseudocódigo. Agora, sua atividade é convertê-los em programas Pascal. Caso você não os tenha feito anteriormente, aproveite para fazê-los agora.
- 2)** Converta para programas Pascal as atividades 5, 7 e 8 do capítulo 4, que haviam sido resolvidas por meio da criação de algoritmos em pseudocódigo. Caso você não os tenha feito anteriormente, aproveite para fazê-los agora.
- 3)** Na atividade 2 do capítulo 5, você criou 3 versões de algoritmo em pseudocódigo. Agora, sua atividade é convertê-las em programas Pascal. Caso você não os tenha feito anteriormente, aproveite para fazê-los agora.
- 4)** Converta para programas Pascal as atividades 1, 4 e 5 do capítulo 6, que haviam sido resolvidas por meio da criação de algoritmos em pseudocódigo. Caso você não os tenha feito anteriormente, aproveite para fazê-los agora.
- 5)** Converta para programa Pascal a atividade 2 do capítulo 7, que havia sido feita em pseudocódigo. Caso você não tenha feito o algoritmo anteriormente, aproveite para fazê-lo agora.
- 6)** Converta para programa Pascal a atividade 2 do capítulo 8, que havia sido feita em pseudocódigo. Caso você não a tenha resolvido anteriormente, aproveite para fazê-la agora.
- 7)** Nas atividades de 5 a 8 do capítulo 8, você criou 4 algoritmos em pseudocódigo. Agora, sua tarefa é convertê-los em programas Pascal. Caso você não os tenha feito anteriormente, aproveite para fazê-los agora.

Apêndice 1

Principais Figuras Utilizadas na Representação de Algoritmos Por Meio de Fluxogramas

Figura	Nome	Significado
	Terminador	Demarca o início ou término de um algoritmo.
	Processamento	Representa operações de processamento, como cálculos e atribuições de valores.
	Decisão	Representa uma tomada de decisão. Possui uma ou mais entradas, e sempre duas saídas (uma vez que a expressão analisada resultará sempre em verdadeiro ou falso).
	Vídeo	Representa a saída de dados em vídeo.
	Impressora	Representa a saída de dados em uma impressora.
	Teclado	Representa entrada de dados via teclado.
	Entrada/Saída	Representa a entrada ou saída de dados, por um dispositivo externo.
	Procedimento	Representa a execução de uma sub-rotina do tipo procedimento.
	Preparação	Representa a preparação de instruções para processamento em estruturas de repetição do tipo Variável de Controle.
	Disco	Leitura ou gravação de arquivo em disco.
	Fluxo	Indicador de fluxo de execução.
	Conector	Representa o particionamento do algoritmo na mesma página. Também simboliza a união de fluxos de execução.
	Conector de Página	Representa o particionamento do algoritmo em mais de uma página.
	Comentário	Utilizado para inserir comentários ao longo do fluxograma.

Apêndice 2

Configuração de um Ambiente de Estudo de Programação com Pascal

Para montar nosso ambiente de estudo de programação iremos utilizar o FreePascal, que é uma distribuição atualizada e gratuita de um compilador Pascal, acompanhado de uma IDE (ambiente de desenvolvimento). O site oficial do FreePascal é: <<http://www.freepascal.org/>>

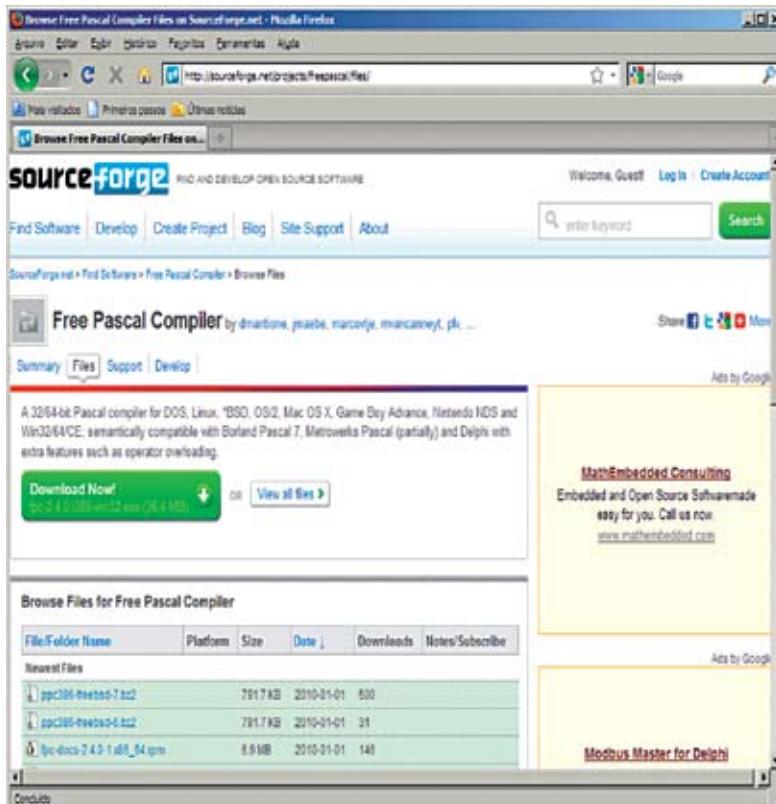
O FreePascal está disponível para diversos sistemas operacionais, o que facilita sua adoção devido à flexibilidade que proporciona. Você pode, por exemplo, utilizar o FreePascal durante o período de aula no laboratório Linux de sua escola e, posteriormente, desenvolver seus trabalhos com o FreePascal em um computador rodando MS Windows na sua casa.

Instalação do FreePascal em Ambiente Windows

O primeiro passo é baixar o FreePascal. Existem diversos locais onde o FreePascal pode ser encontrado para *download*, porém optamos por baixá-lo diretamente do SourceForge, que é um dos repositórios oficiais. O endereço para *download* é o seguinte:

<<http://www.sourceforge.net/projects/freepascal/files/>>

Abrindo o *link* acima, você deverá ver algo como a figura a seguir:



Perceba que o site já identifica o sistema operacional que você está usando e sugere (botão grande, em verde) o *download* da versão do FreePascal compatível com o seu sistema. Caso você deseje baixar uma versão para outro sistema, basta rolar a página até encontrar o seu sistema operacional.

Neste exemplo, iremos utilizar a versão para Windows em 32 bits (win32). Veja na figura abaixo que existem 3 versões disponíveis para win32, então iremos optar pela mais recente delas, que no caso é a 2.4.0.

136

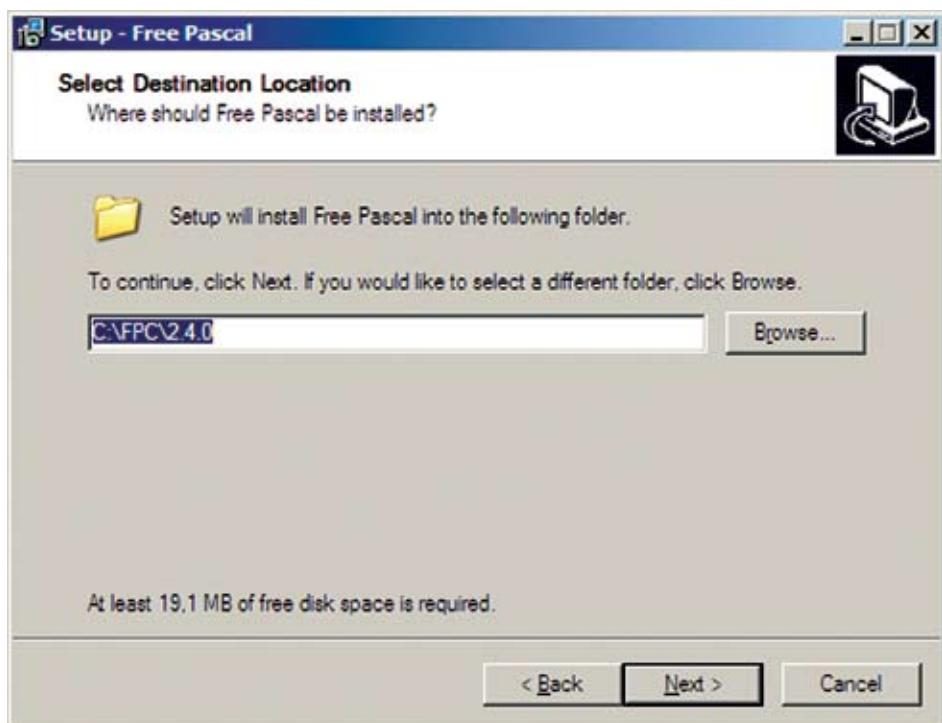
Nome	Tamanho	Data de Lançamento	Nº de Baixos	Ações
Bootstrap	4.0 MB	2010-01-01	564	[Download] [Detalhes]
Linux	856.8 MB	2010-01-01	61,925	[Download] [Detalhes]
Documentation	123.8 MB	2009-12-30	28,059	[Download] [Detalhes]
NDS	81.6 MB	2009-12-30	33,481	[Download] [Detalhes]
GBA	20.1 MB	2009-12-30	3,405	[Download] [Detalhes]
Source	411.2 MB	2009-12-30	15,212	[Download] [Detalhes]
DOS_Go32v2	233.0 MB	2009-12-30	7,103	[Download] [Detalhes]
Win32	191.3 MB	2009-12-30	328,854	[Download] [Detalhes]
2.4.0	65.1 MB	2009-12-30	15,677	[Download] [Detalhes]
fpc-2.4.0.x86_64-win64.exe	17.4 MB	2009-12-30	480	[Download] [Detalhes]
fpc-2.4.0.i386-win32.exe	36.4 MB	2009-12-30	15,002	[Download] [Detalhes]
fpc-2.4.0.arm-wince.exe	11.4 MB	2009-12-30	195	[Download] [Detalhes]
2.2.4	64.3 MB	2009-04-12	123,247	[Download] [Detalhes]
2.2.2	61.9 MB	2008-08-09	189,730	[Download] [Detalhes]
FreeBSD	170.5 MB	2009-12-30	1,518	[Download] [Detalhes]
OS_2	197.7 MB	2009-12-26	966	[Download] [Detalhes]
Mac OS X	314.7 MB	2009-12-24	12,206	[Download] [Detalhes]
DOS Single Pieces	24.4 MB	2008-08-09	6,172	[Download] [Detalhes]

Dentro da versão 2.4.0 ainda existem 3 possibilidades. Iremos escolher a “[fpc-2.4.0.i386-win32.exe](#)” pois é a versão compatível com computadores rodando Windows de 32 bits. Basta clicar no *link*, realizar o *download* e rodar o instalador.

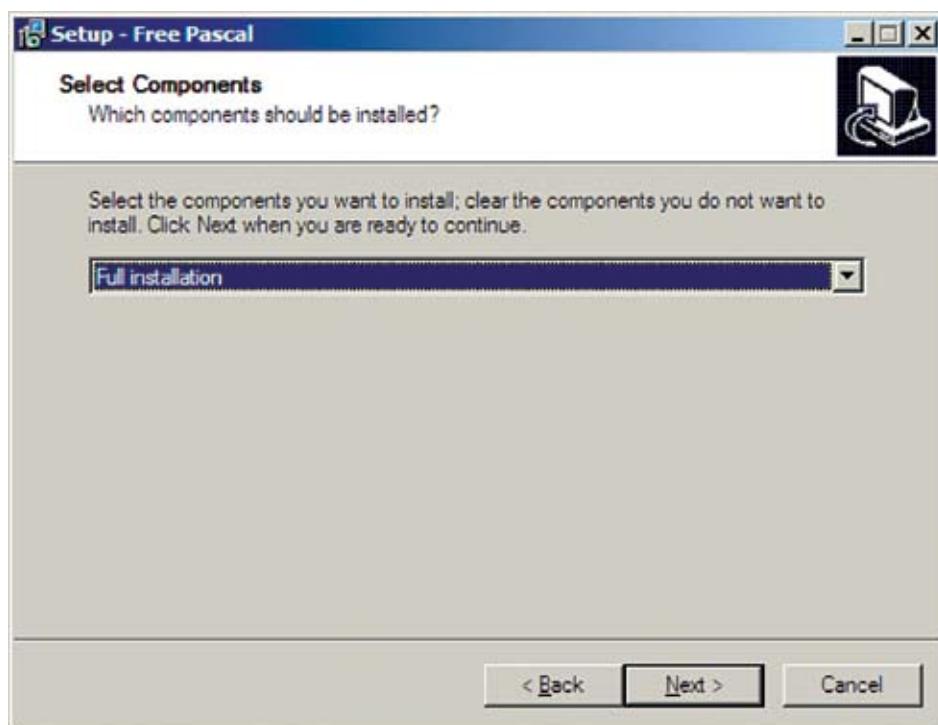
Ao executar o instalador, a primeira tela que você verá será esta:



Basta clicar em **Next**, e será exibida a tela abaixo, solicitando o local de instalação:

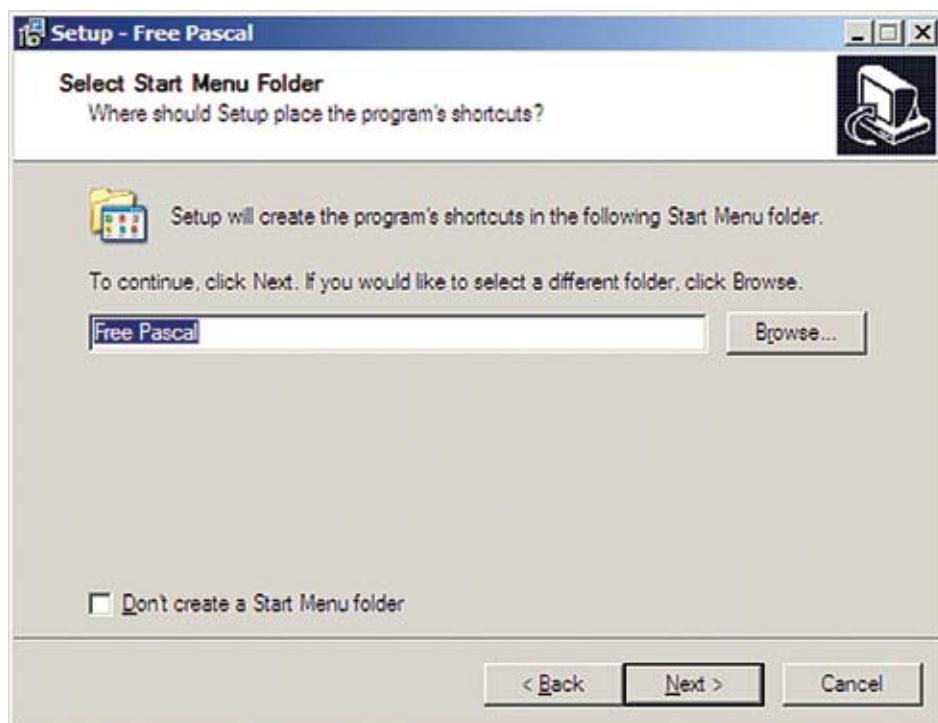


Caso necessário, modifique o local onde o FreePascal deve ser instalado, e clique em **Next**. Nesta próxima tela, será solicitado o tipo de instalação.

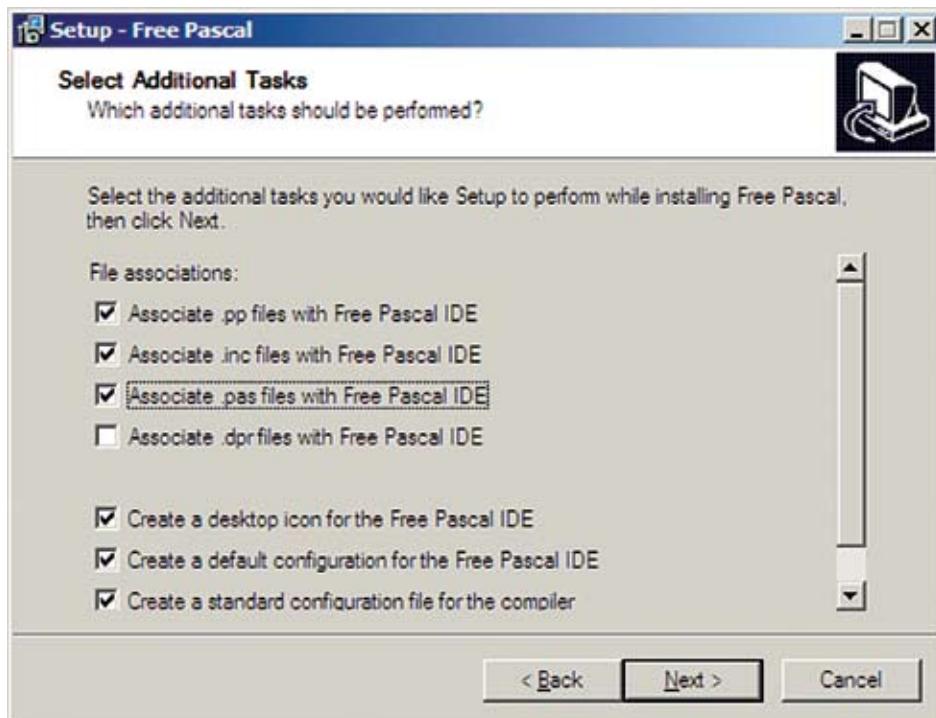


138

Basta deixar na opção **Full installation** e clicar novamente em **Next**. A tela a seguir permite modificar o nome da pasta onde o atalho para o programa ficará localizado no menu Iniciar. Apenas clique em **Next**.

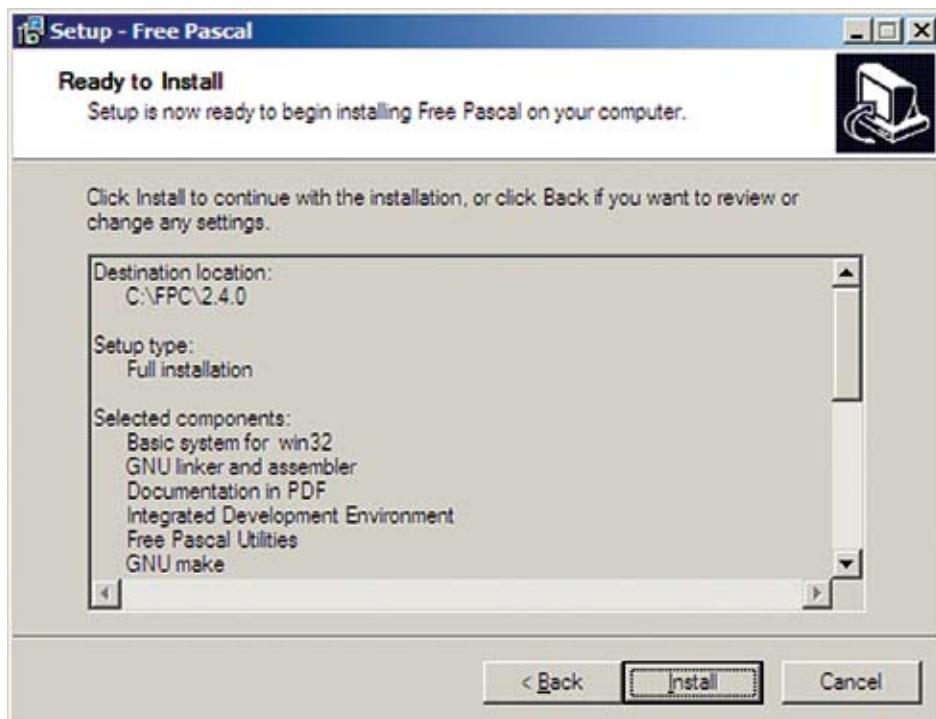


A próxima tela permite associar as extensões dos arquivos fonte pascal com a IDE, e, também, criar uma configuração padrão para o ambiente de desenvolvimento. Certifique-se de ter selecionado as mesmas opções da figura abaixo, e clique em **Next**.

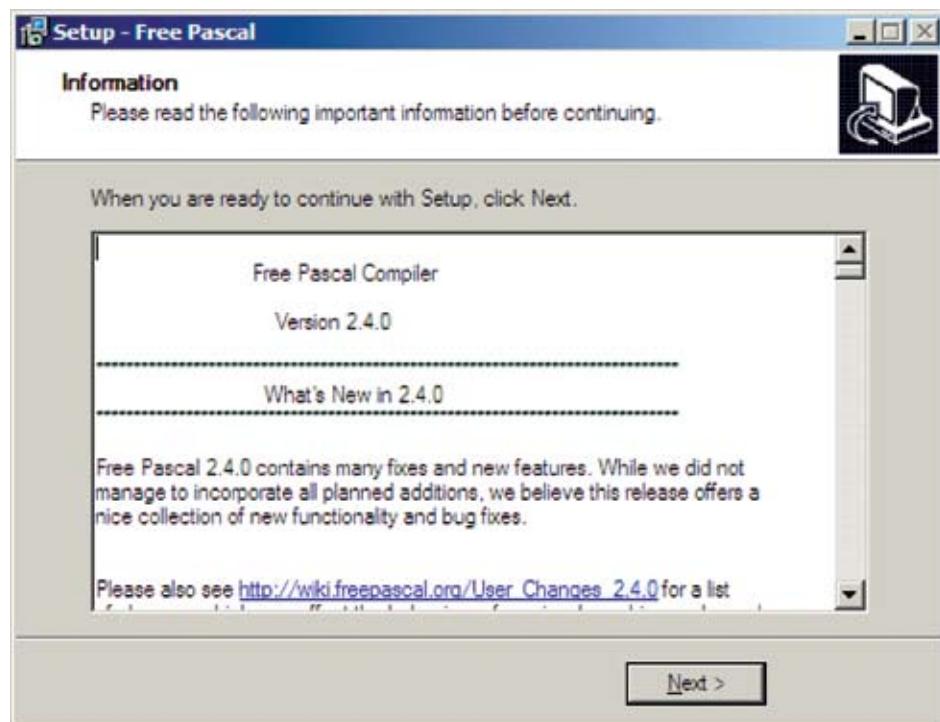


Esta próxima tela apenas exibe as opções de instalação que foram selecionadas. Basta clicar em **Install** e aguardar o término da instalação.

139



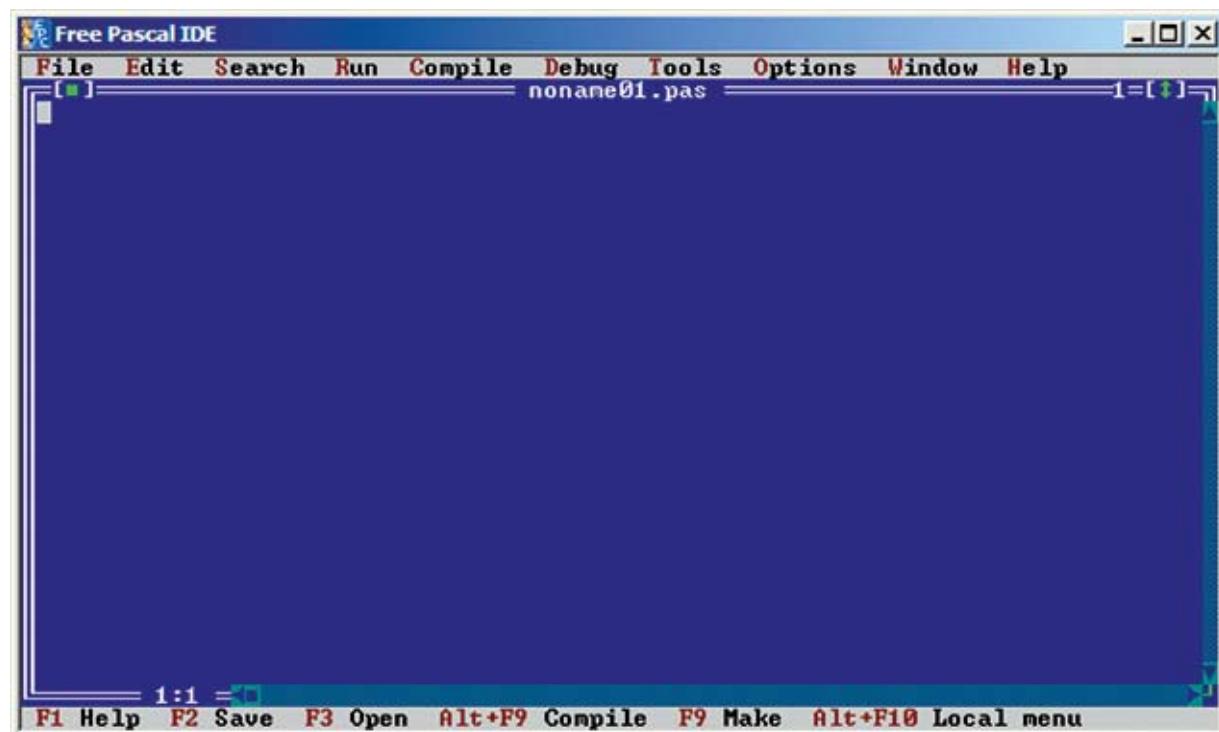
Ao final do processo de instalação, será exibida a tela a seguir:



Basta clicar em **Next** e, na próxima tela, desmarcar a opção **View readme.txt** e clicar em **Finish**.



Pronto, o FreePascal está instalado no computador e pronto para o uso. Abrindo o ambiente pelo *link* criado durante a instalação, você irá se deparar com a seguinte interface:



Agora você está com o ambiente pronto para programar em Pascal. Bom aprendizado!

Instalação do FreePascal em Ambiente Linux

Como existem diversas distribuições Linux diferentes, o processo de instalação poderá ser diferente de acordo com a distribuição escolhida. Neste exemplo iremos instalar o FreePascal no Ubuntu, que é uma das mais utilizadas atualmente.

Iremos realizar a instalação diretamente por meio dos repositórios oficiais do Ubuntu. Para isto, abra um console e digite o seguinte comando:

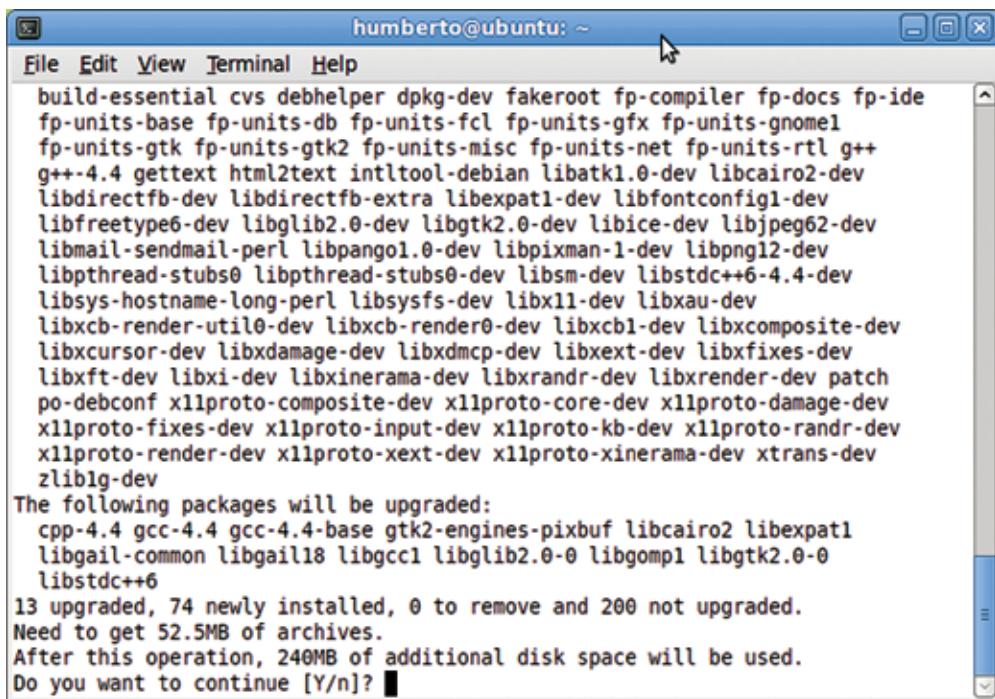
```
sudo apt-get install fp-compiler fp-docs fp-ide fp-units-base fp-
units-db fp-units-fcl fp-units-gfx fp-units-gnomel fp-units-gtk
fp-units-gtk2 fp-units-misc fp-units-net fp-units-rtl
```

Isto irá realizar a instalação de diversos pacotes referentes ao compilador e ao ambiente de desenvolvimento do FreePascal.

```
humberto@ubuntu: ~
File Edit View Terminal Help
humberto@ubuntu:~$ sudo apt-get install fp-compiler fp-docs fp-ide fp-units-base
fp-units-db fp-units-fcl fp-units-gfx fp-units-gnomel fp-units-gtk fp-units-gtk
2 fp-units-misc fp-units-net fp-units-rtl
```

A screenshot of a terminal window titled "humberto@ubuntu: ~". The window has a standard window title bar with icons for minimize, maximize, and close. The menu bar includes File, Edit, View, Terminal, and Help. The terminal itself shows the command "sudo apt-get install fp-compiler fp-docs fp-ide fp-units-base fp-units-db fp-units-fcl fp-units-gfx fp-units-gnomel fp-units-gtk fp-units-gtk2 fp-units-misc fp-units-net fp-units-rtl" being typed in.

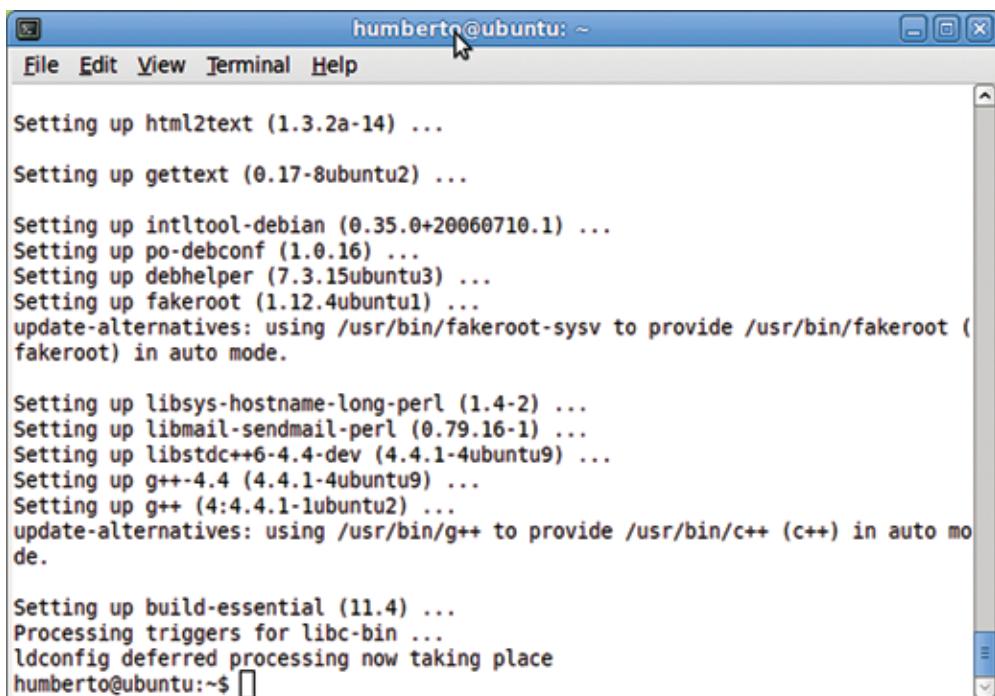
Após digitar o comando, pode ser necessário informar uma senha com poderes administrativos. Em seguida, será exibida uma lista dos pacotes que serão baixados/installados, seguida de uma solicitação de confirmação. É necessário confirmar informando “Y”.



A screenshot of a terminal window titled "humberto@ubuntu: ~". The window shows the output of a package manager command. It lists many packages being installed or upgraded, including build-essential, cvs, debhelper, dpkg-dev, fakeroot, fp-compiler, fp-docs, fp-ide, fp-units-base, fp-units-db, fp-units-fcl, fp-units-gfx, fp-units-gnomel, fp-units-gtk, fp-units-gtk2, fp-units-misc, fp-units-net, fp-units-rtl, g++, g++-4.4, gettext, html2text, intltool-debian, libatk1.0-dev, libcairo2-dev, libdirectfb-dev, libdirectfb-extra, libexpat1-dev, libfontconfig1-dev, libfreetype6-dev, libglib2.0-dev, libgtk2.0-dev, libice-dev, libjpeg62-dev, libmail-sendmail-perl, libpango1.0-dev, libpixman-1-dev, libpng12-dev, libpthread-stubs0, libpthread-stubs0-dev, libsm-dev, libstdc++6-4.4-dev, libsys-hostname-long-perl, libsysfs-dev, libx11-dev, libxau-dev, libxcb-render-util0-dev, libxcb-render0-dev, libxcb1-dev, libxcomposite-dev, libxcursor-dev, libxdamage-dev, libxdmcp-dev, libxext-dev, libxfixes-dev, libxft-dev, libxi-dev, libxinerama-dev, libxrandr-dev, libxrender-dev, patch, po-debconf, x11proto-composite-dev, x11proto-core-dev, x11proto-damage-dev, x11proto-fixes-dev, x11proto-input-dev, x11proto-kb-dev, x11proto-randr-dev, x11proto-render-dev, x11proto-xext-dev, x11proto-xinerama-dev, xtrans-dev, zlib1g-dev. A message follows: "The following packages will be upgraded: cpp-4.4 gcc-4.4-base gtk2-engines-pixbuf libcairo2 libexpat1 libgail-common libgail18 libgcc1 libglib2.0-0 libgomp1 libgtk2.0-0 libstdc++6". Then it says "13 upgraded, 74 newly installed, 0 to remove and 200 not upgraded. Need to get 52.5MB of archives. After this operation, 240MB of additional disk space will be used. Do you want to continue [Y/n]?"

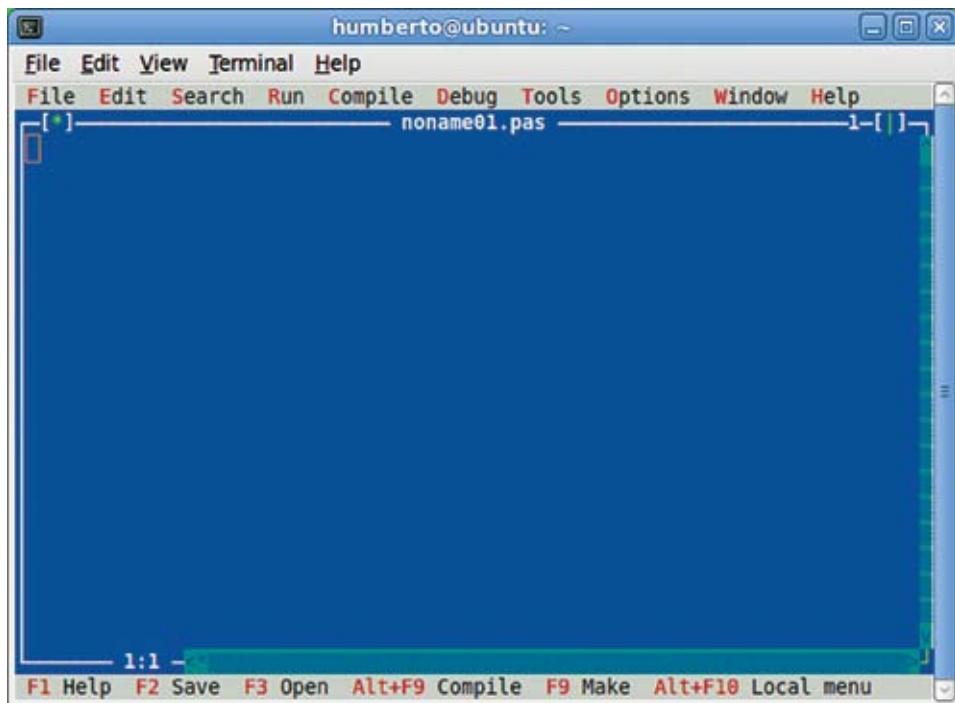
142

Feito isso, a instalação ocorrerá de maneira automática. Ao final da instalação, o terminal irá retornar ao *prompt* de comando, parecendo-se com a imagem abaixo:



A screenshot of a terminal window titled "humberto@ubuntu: ~". The window shows the completion of a package installation. It lists several packages being set up, including html2text (1.3.2a-14), gettext (0.17-8ubuntu2), intltool-debian (0.35.0+20060710.1), po-debconf (1.0.16), debhelper (7.3.15ubuntu3), fakeroot (1.12.4ubuntu1), libsys-hostname-long-perl (1.4-2), libmail-sendmail-perl (0.79.16-1), libstdc++6-4.4-dev (4.4.1-4ubuntu9), g++-4.4 (4.4.1-4ubuntu9), g++ (4:4.4.1-1ubuntu2), build-essential (11.4), and libc-bin. It also shows the processing of triggers and deferred ldconfig processing. The prompt "humberto@ubuntu:~\$ " is visible at the bottom.

Para abrir o ambiente de desenvolvimento, basta digitar o comando **fp** no terminal. A interface do ambiente de desenvolvimento do FreePascal para Linux é esta:



Se estiver utilizando outra distribuição Linux, você poderá baixar do SourceForge o instalador compatível com a sua distribuição:

A screenshot of a Mozilla Firefox browser window showing a list of files for the Free Pascal Compiler (FPC) on SourceForge.net. The URL in the address bar is "http://sourceforge.net/projects/freepascal/files/". The page lists various packages under the "Linux" category, specifically version 2.4.0. The listed files include:

File	Size	Last Modified	Type
fpc-docs-2.4.0-1.x86_64.rpm	6.9 MB	2010-01-01	832
fpc-2.4.0-1.x86_64.rpm	31.0 MB	2010-01-01	1.558
fpc-2.4.0-1.src.rpm	28.7 MB	2010-01-01	984
fpc-docs-2.4.0-1.i386.rpm	6.9 MB	2010-01-01	761
fpc-2.4.0-1.i386.rpm	29.5 MB	2010-01-01	2.720
fpc-2.4.0.x86_64-linux.tar	40.1 MB	2009-12-30	738
fpc-2.4.0.powerpc-linux.tar	36.0 MB	2009-12-30	51
fpc-2.4.0.powerpc64-linux.tar	40.3 MB	2009-12-30	37
fpc_2.4.0.orig.tar.gz	32.6 MB	2009-12-30	472
fpc_2.4.0-0_i386.changes	9.2 KB	2009-12-30	52
fpc_2.4.0-0_all.deb	11.5 KB	2009-12-30	1.327
fpc_2.4.0-0.dsc	1.4 KB	2009-12-30	38
fpc_2.4.0-0.diff.gz	18.3 KB	2009-12-30	50
fpc-source_2.4.0-0_all.deb	13.6 MB	2009-12-30	4.266

143

Pronto, agora você está com o ambiente configurado para programar em Pascal. Bom aprendizado!

Referências Bibliográficas

- CANTÙ, M. *Essential Pascal*. 2. ed. – eBook. Disponível em: <<http://www.marcocantu.com/epascal/>>. Acesso em: 20 dez. 2009. Edição do autor, 2003.
- EVARISTO, J. *Programando com Pascal*. 2. ed. São Paulo: Book Express, 2004.
- FARRER, H, et al. *Algoritmos Estruturados*. 3. ed. Rio de Janeiro: LTC Editora, 1999.
- LOPES, A; GARCIA, G. *Introdução à Programação*. Rio de Janeiro: Elsevier Editora, 2002.
- MANZANO, J. A. N. G.; OLIVEIRA, J. F. de. *Algoritmos: Lógica para Desenvolvimento de Programação*. 7. ed. São Paulo: Erica, s/d.
- VENANCIO, C. F. *Desenvolvimento de Algoritmos: Uma Nova Abordagem*. São Paulo: Érica, 1997.
- XAVIER, G. F. C. *Lógica de Programação*. 11. ed. São Paulo: Senac, 2009.