Introdução ao Processamento de Dados - Notas de aula - IPD - UERJ - 2000/2

Ricardo Jurczyk Pinheiro

17 de junho de 2007

Sumário

List of Algorithms

Capítulo 1

Introdução à Informática

1.1 O que é um computador?

O computador é uma máquina eletrônica que interpreta, transforma e gera dados, que são representações de informação e conhecimento.

1.2 Processos básicos

- Entrada de dados : ler os dados iniciais ou constantes.
- Processamento : efetua as operações necessárias.
- Saída de dados : apresenta os resultados.

1.3 Características

São específicas - difere das demais máquinas de cálculos

- alta velocidade na execução de suas operações.
- grande capacidade de armazenar informações (memória).
- capacidade de executar uma longa seqüência alternativa de operações (programa).

1.4 Um breve histórico

A necessidade de efetuar cálculos e operações repetitivas fez com que o homem tivesse a idéia de criar algum meio de efetuá-los repetidamente.

1.4.1 Geração pré-computador

2500 AC - China -Ábaco: O ábaco é uma peça de madeira com fios paralelos e contas deslizantes, onde a posição marcava a quantidade a ser trabalhada. Existiram ábacos chineses, russos e japoneses, os soroban, até hoje usados para cálculos simples.

Século XVI - Pascalina: A Pascalina foi uma máquina de calcular mecânica criada por Blaise Pascal para facilitar as suas contas no escritório de contabilidade do seu pai. Permitia, com dois jogos de engrenagens, encontrar resultados de somas, subtrações e multiplicações.

Século XVII - Régua de cálculo: Oughtred colocou os bastões de logaritmos de Napier sobre uma régua, com escala. Fazendo um movimento sobre a régua, era possível encontrar já calculados os valores de logaritmos. A divisão e o produto de logaritmos você obtinha através da adição e subtração de comprimentos na régua.. A régua de cálculo e sua sucessora, a régua de cálculo cilíndrica, foram usadas até os anos 70 do século XX.

Século XVIII - Teares automáticos: Jacquard observou o funcionamento dos teares (usados para trançar linhas e fazer tecelagem), e criou os primeiros programas em cartões perfurados. Esses cartões continham a ordem de funcionamento do tear, e como trançar certos padrões mais complicados. Uma espécie de leitora de cartões perfurados reconhecia os furos e ajustava automaticamente os teares. A idéia dos cartões perfurados persistiu na informática até o início da década de 80 do século XX.

Século XIX - O matemático Charles Babbage apresentou à Sociedade Real de Astronomia a sua Máquina de Diferenças e posteriormente, a Máquina Analítica. Essas máquinas são consideradas como os precursores dos computadores modernos. A Máquina Analítica, por exemplo, tinha aplicabilidade mais ampla do que a primeira máquina, e também a mesma divisão das máquinas atuais: unidade de controle de memória, unidade lógica e aritmética e de entrada e saída.

1.4.2 Primeira geração

- Conhecida como a geração das válvulas.
- Durou de 1940 a 1955.
- Usada para resolução de cálculos.
- Exemplos: Mark I, UNIVAC, ENIAC, etc.

1.4.3 Segunda geração

- Uso de transistores, o que possibilitou diminuir o tamanho do computador e aumentar o seu poder de processamento.
- Durou de 1955 a 1964.
- Exemplos: copmputadores comerciais da IBM , linguagens de programação Fortran e Cobol.

1.4.4 Terceira geração

- Uso de circuitos integrados, onde a miniaturização aumentou, assim como a sofisticação da área.
- Durou de 1964 a 1972
- Exemplos: DEC PDP/11, IBM 1130, etc.

1.4.5 Quarta geração

- Uso de circuitos integrados de larga escala, onde possibilitou o surgimento dos microcomputadores.
- Foi inventado o microprocessador, por um grupo de engenheiros[1], que posteriormente seria a base da computação.
- Dura de 1972 até hoje.
- Exemplos: Apple II, MSX, IBM-PC, etc.

1.5 Divisões

1.5.1 Hardware

Conjunto de circuitos eletrônicos inalteráveis. O hardware é a máquina em si.

1.5.2 Software

Conjunto de instruções alteráveis - isto porque os programadores podem facilmente mudá-las.

1.5.2.1 Programa

Um programa de computador é uma série de comandos seguindo uma lógica para desenvolver uma tarefa. Mesmo com esta explicação, apenas programando o leitor compreenderá o que é programar. Alguns chegam ao cúmulo de *programar por instinto*. Em última instância, podemos dizer que programar é uma arte[2]. Mas programação por instinto foge ao escopo do texto e deixamos como exercício para os leitores mais sagazes.

O texto que contém os comandos da linguagem de programação é o código-fonte. O resultado do programa, que é o que o computador pode entender, é chamado de código-objeto, ou módulo-objeto.

Todos os erros são do programador, por mais absurdo que isso pareça. Mesmo se for um problema relacionado às rotinas internas do computador (elas foram feitas por um programador, que errou).

1.5.2.2 Tipos de software

- Software Básico - Conjunto de programas que supervisionam e auxiliam a execução dos diversos softwares aplicativos. O software básico é. em geral, formado pelos seguintes programas principais:
- Sistema Operacional - é responsável pela interface (interação) entre hardware e o usuário, o hardware e outros softwares aplicativos. Exemplos: MS-DOS, WINDOWS 95, UNIX, OS/2, QNX, BeOS, Amoeba, Plan 9, etc.
- Compiladores e Interpretadores: traduzem ou interpretam os programas escritos em diferentes linguagens.

[1] Esse mesmo grupo de engenheiros fundou a Intel.
[2] "The Art of Computer Programming", Donald Knuth

O software básico é fornecido pelo próprio fabricante do computador e, em geral, está escrito em linguagem de máquina, gravado no computador. Não pode ser mudado.

- Software Aplicativo - Programa específico escrito para executar alguma operação (ou resolver um problema) de interesse do usuário. Em geral é escrito em Linguagem de Alto Nível pelo próprio usuário.

1.5.3 Linguagens de programação

Tanto o software aplicativo como o básico trabalham em linguagem de máquina, isto é, em código binário, que é a única codificação aceita pelo hardware ou arquitetura do computador. O usuário, em geral, não manipula diretamente valores ou códigos binários, mas trabalha com valores decimais, hexadecimais e códigos Basic. Pascal, C, etc. Os programas do software básico encarregam-se de efetuar a tradução dos códigos e a conversão dos valores.

1.5.3.1 Compiladores e interpretadores

- Linguagem de alto nível - Existem duas formas primárias usadas para se rodar um programa de computador que está escrito em uma lingua humana (geralmente, em inglês). Quanto mais próximo o programa esta' da lingua humana, mas alto seu "nível". O Pascal, por exemplo, por se utilizar de palavras humanas para designar comandos de computador, está classificada como uma linguagem de alto nível. Quanto mais alto o nivel, mais lenta e grande, mas tanto mais fácil de ser compreendida pelo programador.

- Linguagem de baixo nível - Linguagens como Assembly, que possuem mais influência da máquina do que de humanos, são chamadas de linguagens de baixo nível. Quanto mais baixo o nível, mais rápida e simples a linguagem, mas tanto mais complexa de ser compreendida.

Interpretador - O interpretador é um programa que interpreta cada linha de texto do programa (chamado de código-fonte) e faz a tradução dos comandos "humanos" que aquela linha contém para linguagem de máquina (chamado de código-objeto). Se uma das linhas é executada mais de uma vez, ela deve ser analisada, traduzida e executada a cada vez, degradando demasiadamente a solução do problema em mãos.

Compilador - Um compilador, por sua vez, é um programa que olha todo o código-fonte e o traduz para código-objeto de uma só vez. Quando o computador o executa, ele já está no formato de linguagem de máquina, então é executado a uma velocidade muito maior.

Então pra que existe o interpretador? Simples: o programa, uma vez compilado, não pode ser facilmente modificado. Você o monta e nunca mais o altera, mesmo se encontrar um erro nele. Então, para testar o programa antes de finalizá-lo, é muito mais fácil rodá-lo interpretado. Assim, quando algo der errado, voce interrompe o programa, conserta o erro e roda o mesmo de novo. Se você fosse compilar o programa inteiro de novo, teria muito tempo gasto. Uma ouitra vantagem do interpretador é a *portabilidade*: podemos pegar um programa interpretado e executá-lo sem

alterações em outro computador, bastando ter um interpretador compatível.

1.5.4 Sistema operacional

O sistema operacional deve ser adaptado às características do hardware assim como as linguagens de programação e ferramentas do usuário final devem ser adaptados ao sistema operacional. Conhecer o sistema operacional pode ajudar a resolver alguns problemas que a princípio nos parecem complicados. Além disso, possui utilitários especiais para a formatação de discos, listagens em vídeo/impressora, criação/copia/exclusão e alterações de arquivos. Podemos dizer que o sistema operacional é um conjunto de rotinas, ou seja, uma lista de instruções passadas para o microprocessador com a finalidade promover a comunicação do usuário com o hardware.

1.5.4.1 Diretórios e arquivos

Os diretórios são como armários e gavetas, cuja função é organizar os arquivos. O sistema operacional permite o gerenciamento dos arquivos em forma de árvore onde cada galho é chamado Diretório/ Subdiretório. Vejamos:

C: DOS5\DADOS\CONTAB\COMPRAS\VendAS\WINDOWS

Isso significa que dentro do diretório principal C: temos um diretório chamado DOS5, outro diretório chamado DADOS e finalmente outro diretório chamado WINDOWS. Porém dentro do diretório DADOS temo três outros diretórios assim intitulados: CONTAB, COMPRAS, VendAS.

Os arquivos são divididos em dois tipos:

- Arquivo Programa - Conjunto de instruções para o computador juntados em um só arquivo.
- Arquivo Dados - Conjunto de caracteres (dados) que podem ser documentos, banco de dados e etc.

Devem ser utilizados nome de arquivos de fáceis associações ao assunto a que se referem. Os nomes de arquivos normalmente possuem duas partes separadas por um ponto. A primeir parte é o nome do arquivo, e a segunda parte, a extensão, que é opcional. Geralmente a extensão especifica o tipo de arquivo.

Exemplos:

.COM - Utilizado para arquivos de comandos (Programas)

.EXE - Utilizado para arquivos executáveis (Programas)

.BAT - Utilizado para arquivos em lote (Batch) - que são criados em um editor de texto qualquer e possuem uma seqüência de comandos do DOS

.PAS - Arquivos de Programas em Pascal

.C - Arquivos de Programas em C

.DBF - Arquivos de bancos de dados no formato dBase

.DOC - Arquivos de texto no formato do Word

.XLS - Arquivos de planilha no formato do Excel

.TXT - Arquivos de texto.

.JPG - Arquivo de imagem no formato J-PEG.

1.6 Arquitetura de computadores

1.6.1 Bit e byte

Cada sinal elétrico que o computador processa é chamado de BIT, ou Binary Digit, e é representado por 0 ou 1.

1 - 5 volts - ligado, i.e., tem corrente elétrica

0 - 0 volts - desligado, i.e., não tem corrente elétrica

- Bit

É a menor partícula de informação em um computador, mas um único bit não consegue representar todas as letras, números e caracteres especiais com os quais o computador trabalha. É necessário agrupá-los e cada grupo é chamado de Byte.

- Byte

É usualmente um grupo (conjunto) de 8 bits e equivale a um caracter.

- Caracter

É a unidade básica de armazenamento de informação na maioria dos sistemas, ou seja, é a representação gráfica de uma letra, número ou símbolo especial do alfabeto. A tabela de código representada por bytes chama-se **ASCII** (American Standard Code for Information Interchange).

- ASCII

É o conjunto de caracteres contém os dígitos de 0 a 9, todas as letras minúsculas e maiúsculas, sinais de pontuação, 32 caracteres de controle e 128 caracteres especiais que incluem frações, letras de alfabeto estrangeiro e gráficos de linha para desenhar quadros e formas. Usa-se um byte para representar todos os 256 caracteres que compõem essa tabela.

- Palavra

É a quantidade de bits que a CPU processa por vez.

1.6.2 Bases de numeração

Base	Conjunto de números
Decimal (10)	(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
Binário (2)	(0, 1)
Octal (8)	(0, 1, 2, 3, 4, 5, 6, 7)
Hexadecimal (16)	(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F)

1.6.3 Unidades de medida

Tanto para quantificar a memória principal do equipamento como para indicar a capacidade de armazenamento, são usados múltiplos de bytes, como:

Letra	Medida	Quantidade (decimal)	Tamanho em bytes	Potência de 2
K	Kilo	mil	1024 bytes	2^{10} bytes
M	Mega	milhão	1048576 bytes	2^{20} bytes
G	Giga	bilhão	1073741824 bytes	2^{30} bytes
T	Tera	trilhão	1099511627776 bytes	2^{40} bytes

Capítulo 2

Algoritmos

2.1 Definição e exemplos

Um algoritmo é uma sequência finita de ações, com início e fim, que solucionam um determinado problema.

Exemplos:

- **Algoritmo sem repetição**:

1. Problema: Após o banho, um homem resolve vestir-se, com camisa, calça, cueca, meias, sapatos, gravata, cinto e paletó. Como resolver?
2. Análise do problema: Não podemos vestir algumas peças antes das outras: Sapatos antes das meias, calças antes das cuecas, ou gravata e paletó antes da camisa.
3. Solução:

- Vestir a cueca
- Calçar as meias
- Vestir a camisa
- Vestir a calça
- Calçar os sapatos
- Colocar o cinto
- Colocar a gravata
- Vestir o paletó

Acabamos de montar um algoritmo para a resolução desse problema, no caso um *algoritmo não-estruturado*.

- **Algoritmo com repetição**:

1. Problema: Temos que cravar um prego numa trave de madeira.
2. Análise do problema: Para realizar essa tarefa, teremos que segurar o prego sobre a madeira e bater com o martelo tantas vezes quantas forem necessárias até que o prego entre pôr inteiro.

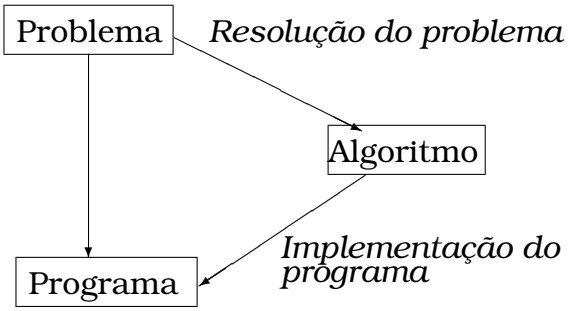
3. Solução: Repetir até que o prego esteja pregado.

- segurar o prego sobre a madeira.
- bater com o martelo no prego.

Ou seja, o que temos com esses 2 exemplos são algoritmos simples para a resolução de problemas. Existem três estruturas básicas para resolver o problema, como vimos acima:

- Sequência
- Repetição
- Seleção

Para a resolução no computador, os passos são um pouco diferentes, e estão ilustrados abaixo:



2.2 Programação Estruturada

Não existe uma definição aceita por todos para a programação estruturada; ao contrário, ela é conceituada por vários teóricos de ciência da computação. Num sentido mais restrito, o conceito de programação estruturada diz respeito à forma do programa e do processo de codificação.

A programação estruturada é um conjunto de convenções que o programador pode seguir para produzir o código-fonte estruturado. As regras de codificação impõem limitações sobre o uso das estruturas básicas de controle, estruturas de composição modular e documentação.

Desse modo, a programação estruturada evita o uso de desvios incondicionais (como o comando GOTO, de linguagens não-estruturadas, como Basic) e trabalha com apenas três estruturas básicas de controle: seqüência, seleção e iteração.

Em um sentido mais amplo, a programação estruturada é dirigida tanto aos métodos de programação como à forma do programa. Implica seguir uma metodologia estruturada para organizar o projeto e a implantação do programa.

Os objetivos da programação estruturada são melhorar a confiabilidade e tornar o programa mais legível, diminuindo sua complexidade e facilitando a manutenção do código. Desse modo forma-se aos poucos uma metodologia disciplinada no programador, aumentando a sua produtividade.

O Pascal, assim como C, Java e outros são exemplos de linguagem estruturada. Outras, como Basic (e suas variações, como Visual Basic, por exemplo), APL e outros, são linguagens não-estruturadas.

Algorithm 1 Exemplo de um programa estruturado, escrito em Pascal

```
Program Exemplo1;                    CABEÇALHO DO PROGRAMA
TYPE

        X: array of integer;         DECLARAÇÃO DE TIPOS

CONST

        Constante1=1;
        Constante2=2;                DECLARAÇÃO DE CONSTANTES
        PI=3.1415927;

var

        a: integer;                  DECLARAÇÃO DE VARIÁVEIS
        b: real;

begin

        a:=Constante1+Constante2;
        b:=b+a;          SEÇÃO DE CÓDIGO OU CORPO DO PROGRAMA
        WRITELN(b);

end.
```

2.3 Algoritmo e linguagem estruturada

Quando precisamos resolver um problema utilizando um computador, podemos usar a idéia da programação estruturada, e em decorrência, do algoritmo estruturado. Desse jeito simplifica-se e muito a solução de problemas grandes e complexos. Como o objetivo do curso é trabalhar com Pascal, estaremos escrevendo os exemplos em Pascal (linguagem estruturada), não em pseudo-código, como é mais comum.

2.3.1 Elementos básicos

- Letras (de a até z, de A até Z, e _ - sublinhado)
- Dígitos (0 a 9)
- Símbolos especiais (+ - * / = < > [ ] ( ) { } . , : ; # $)

2.3.2 Palavras reservadas

Essas palavras são partes integrantes da linguagem. Elas não podem ser mudadas, serem usadas como variáveis, constantes ou identificadores padrão, pois possuem uso esecífico dentro da linguagem.

2.3.3 Identificadores

Conjunto de caracteres (letras, números ou símbolos) que dão nome a constantes, variáveis, tipos, arquivos, módulos, etc. Os identificadores:

- Não podem conter espaços em branco;

- Não podem ser palavras reservadas;
- Não podem conter símbolos especiais;
- Não podem iniciar por número.

As linguagens definem um número de **identificadores padrão** de tipos pré-definidos, constantes, variáveis, procedimentos e funções. Qualquer um desses identificadores podem ser redefinidos, mas perdem o seu sentido original e podem causar confusão.

Os elementos da linguagem devem ser delimitados por pelo menos um **delimitador**, como um espaço em branco, um comentário ou um caracter de fim de linha. O comprimento máximo de uma linha é de 127 caracteres. Além disso é ignorado pelo compilador. Logo, tome cuidado com linhas realmente extensas.

2.3.4 Operadores

Os operadores lidam com elementos do programa, como expressões aritméticas, comparações ou mesmo operações lógicas. Abaixo temos uma lista dos operadores usados no Pascal:

- Aritméticos e funções

Operador	Função
+	Adição
-	Subtração
*	Multiplicação
/	Divisão
div	Divisão inteira
mod	Módulo (resto da divisão)

- Relacionais:

Operador	Função
=	Igual
<>	Diferente
>	Maior
<	Menor
>=	Maior ou igual
<=	Menor ou igual

- Lógicos:

Operador	Função
AND	Conjunção
OR	Disjunção
XOR	Disjunção exclusiva
NOT	Negação

- Atribuição: :=
- Comentários: { e }, ou (* e *)
- Prioridade dos operadores:

Parênteses e funções, * e /, + e -, operadores relacionais, operadores lógicos de negação, conjunção e disjunção.

2.3.4.1 Hierarquia

A ordem de precedência determina que a multiplicação e a divisão sejam efetuadas antes da adição e subtração, e que qualquer operação entre parênteses seja calculada primeiro.

Nível 1	Sinal de subtração unário, Sinal de adição unário
Nível 2	parênteses
Nível 3	multiplicação, divisão
Nível 4	adição, subtração

2.4 Estrutura de um programa em Pascal

O programa em Pascal divide-se em:

Cabeçalho do programa

Nome do programa
Diretivas do compilador

Seção de dados

Declarações de Constante
Declarações de Tipo
Declarações de Variável

Seção de código

Rotinas
Funções
Bloco de Programa

A seguir, vamos comentar cada uma delas.

2.4.1 Cabeçalho do programa

A seção de cabeçalho do programa não é obrigatória ser preenchida, mas é usada para identificação e determinação de alguns parâmetros para uso do compilador.

O nome do programa faz a identificação do programa com um nome. Aqui, as mesmas regras para as variáveis continuam valendo, e não podemos ter um outro termo (variável, constante, tipo, etc.) com um nome semelhante ao nome do programa.

A palavra reservada que precede o nome é **program.** Ao final do nome, colocamos ponto-e-vírgula (;).

Exemplo: program Calcula_Area;

As diretivas do compilador também estão localizadas nessa seção, e servem para definir várias opções para o compilador fazer uso. Essas diretivas servem para definir tarefas como verificação de erros, alinhamento de dados na memória, se o programa deve preparar-se para ser examinado (caso seja necessário) pelo depurador, verificar erros de entrada e saída em tempo de compilação, etc.

Definimos a diretiva entre chaves ({ e }), e com um sinal de cifrão ($) entre as chaves, mais a letra correspondente.

Exemplo: {$I+}, {$A-, D+, N+, R+}

2.4.2 Seção de dados

As variáveis globais, constantes, tipos de dados definidos pelo usuário são declarados logo abaixo do cabeçalho do programa.

2.4.2.1 Declaração de constantes

Essa área introduz identificadores como sinônimos para valores constantes. Esses valores não poderão ser mudados ao longo do programa, serão fixos. Usaremos a palavra reservada **const**. Conforme vimos na seção 3.3, podemos ter constantes com e sem tipo definido.

Exemplo: const DiasDaSemana = 7; (**constante sem tipo definido**)

const II : Integer = 100; (**constante com tipo definido**)

2.4.2.2 Declaração de tipos

Um tipo de dado em Pascal pode ser descrito diretamente na área de declaração de variáveis ou ser referenciada por um identificador de tipo. Usaremos a palavra reservada **type**, e os usuários podem criar tipos de dados novos, baseados nos tipos originais, como poderá ser visto na seção 3.2.

Exemplo: type TipoDePagamento = (Salario, HoraTrabalhada);

2.4.2.3 Variáveis

As variáveis são áreas da memória a que você atribui um nome e um tipo de dados que poderá estar lá. Usaremos a palavra reservada **var**, seguida pelo identificador da variável, dois-pontos e o tipo de dado. Podemos definir várias variáveis do mesmo tipo de uma só vez, separando por vírgula (,).

Exemplo: var SalarioBruto, SalarioLiquido: real;

2.4.3 Seção de Código

Esta é a terceira e maior parte do programa. Aqui estarão as instruções passo-a-passo que o programa deverá executar. Essa seção contém sempre pelo menos um bloco de programa (delimitado pelas palavras reservadas **begin** e **end**), e constantemente, procedimentos - ou rotinas, e funções. O bloco do programa é a parte que é definida por último, mas é a primeira a ser executada.

Algorithm 2 Exemplo de uma área de declarações.

```
const

        PI = 3.1415927;
        NumeroDeDiasNoMes = 30;

type

        lista = array [1..NumeroDeDiasNoMes] of Real;

var

        A, B, C, IDADE: integer;
        NOME: string[40];
        ALTURA: real;
        Medidas: lista;
```

2.4.4 Rotinas ou Procedimentos

Os procedimentos são iniciados com a palavra reservada **procedure**, e são subrotinas do programa. Um procedimento é ativado quando necessário, e veremos em detalhes o seu funcionamento posteriormente.

2.4.5 Funções

As funções iniciam-se com a palavra reservada **function**, e funcionam parecido com os procedimentos. A diferença é que a função computa um resultado e retorna um valor para o trecho de código que a chamou. Veremos como funciona posteriormente.

2.4.6 Bloco do programa

O programa propriamente dito em Pascal está aqui, onde especificamos as ações a serem executadas pelo programa. Iniciamos essa área com **begin** e finalizamos com a palavra reservada **end**, seguida de um ponto (.).

```
begin
<instruções>
(...)
end.
```

Capítulo 3

Pascal - introdução

3.1 Breve histórico

A linguagem Pascal recebe esse nome em homenagem ao grande filósofo e matemático francês **Blaise Pascal**, que viveu entre 1623 e 1662. Entre outras criações, Pascal inventou a *Pascalina,* uma máquina de calcular mecânica, criou o *Princípio de Pascal* para a hidrodinâmica, e o *Triângulo de Pascal*, usado em Matemática.

No anos 60, na Universidade de Zurique, o professor **Niklaus Wirth**, então lecionando as linguagens ALGOL[1] e PL/1 para seus alunos, pensa em criar uma linguagem de programação que fosse mais simples, mas ao mesmo tempo pudesse passar os conceitos básicos da programação estruturada. A primeira notação da linguagem, então chamada de **Pascal**, surgiu em 1968, e o primeiro compilador, em 1970. O Prof. Wirth procurou desenvolver uma forma de linguagem que fosse de fácil assimilação e aceitação dos alunos de programação, e atingiu seu objetivo. Posteriormente, o Prof. Wirth desenvolveu outras linguagens, como Oberon, Modula-2 e Modula-3.

Em 1970, é criado o primeiro compilador para a linguagem. Com o passar dos anos e o crescimento do uso, surgiram algumas variações, como o **Pascal-ISO** (padrão, seguindo à risca as orientações de Wirth), **VAX Pascal**, **HS Pascal** (Pascal interpretado), mas das suas variações, a mais conhecida é o **Turbo Pascal.**

Em 1983, Philippe Kahn, então dono de uma pequena empresa[2], começa a vender por correspondência um compilador Pascal para MS-DOS e CP/M por meros US$ 50. Até então, os compiladores custavam 10 vezes esse preço e exigiam muito mais do que havia disponível nos computadores de até então.

O sucesso foi imediato: Em 1985, o Turbo Pascal já estava na versão 3.0, e logo surgiram as versões 4.0 (a primeira com o recurso das **units**, bibilotecas pré-compiladas), 5.0 e 5.5 (onde pela primeira vez introduziu-se o conceito de **programação orientada a objetos**). Nos anos 90, foram lançadas as versões 6.0 e 7.0, assim como surgiu o **Delphi**, uma linguagem visual, compilada, orientada a objetos, baseada no Pascal original, e que faz um enorme sucesso, estando hoje em dia na versão 5.0.

Existem compiladores Pascal para vários sistemas operacionais hoje em

[1] ALGOL - ALGOrithmic Language. Caiu em desuso por motivos políticos.

[2] Borland International, de Scotts Valley, Califórnia. Hoje em dia a Borland tornou-se Imprise, e Philippe Kahn saiu, fundando a a Starfish Corp.

dia, além do quase- onipresente Turbo Pascal. Podemos citar também o **Pascal +**, o **Mac Pascal**, para plataformas Macintosh e o **fpc**[3] (Free Pascal Compiler), um compilador gratuito e compatível para plataformas DOS, Win32, Unix, Amiga, assim como o **Projeto Lazarus**[4].

3.2 Tipos de dados e instruções primitivas

Qualquer coisa que fazemos em um computador, obedece aos processos básicos (seção 1.2) de entrada, processamento e saída dos dados. Esse trabalho ocorre através da execução de instruções de comando que são ordenadas de forma lógica para executar determinada ação.

Os dados são representados pelas informações que iremos armazenar e processar no computador. Um tipo de dados define o conjunto de valores que uma variável poderá assumir. Cada variável em um programa deve estar associada a apenas um tipo de dados. Podemos compor tipos de dados bem sofisticados no Pascal, mas os tipos básicos, não-estruturados, são simples.

Os tipos de dados podem ser *escalares, reais, caracteres e lógicos.*

3.2.1 Tipos de dados escalares

Esses tipos de dados lidam com números inteiros.

Tipo de dado escalar	Abrangência	Memória ocupada
shortint	$[-128,\ 127]$	1 byte
integer	$[-32.768,\ 32.767]$	2 bytes
longint	$[-2.147.483.648,\ 2.147.483.647]$	4 bytes
byte	$[0,\ 255]$	1 byte
word	$[0,\ 65535]$	2 bytes

Trabalharemos com os tipos integer e byte.

3.2.2 Tipos de dados reais

Esses tipos de dados lidam com números reais, logo englobando inteiros e fracionários.

Tipo de dado real	Abrangência	Memória ocupada	Mantissa
real	$[-2,9\,.\,10^{-39},\ 1,7\,.\,10^{+38}]$	6 bytes	12 dígitos
single	$[-1,5\,.\,10^{-45},\ 3,4\,.\,10^{+38}]$	4 bytes	8 dígitos
double	$[-5,0\,.\,10^{-324},\ 1,7\,.\,10^{+308}]$	8 bytes	16 dígitos
extended	$[-1,1\,.\,10^{-4932},\ 1,1\,.\,10^{+4932}]$	10 bytes	20 dígitos
comp	$[-2\,.\,10^{-63},\ 2\,.\,10^{+63}]$	8 bytes	20 dígitos

Nos exemplos, trabalharemos com o tipo real.

[3] `http://www.freepascal.org`

[4] `http://www.lazarus.freepascal.org/`

3.2.3 Tipos de dados caracteres

São caracterizados tipos caracteres as sequências contendo letras, números e símbolos especiais. No Pascal, representaremos essa sequência entre apóstrofos (‘ ’). Os caracteres estão ordenados segundo a tabela ASCII[5].

Tipo de dado caracter	Abrangência	Memória ocupada
char	1 caracter	1 byte
string	1 a 255 caracterer	1 a 255 bytes

O tipo string tem o tamanho definido pelo programador.
Exemplos: ‘1‘≠1; ‘A‘<‘B‘; var a: string[40];

3.2.4 Tipos de dados lógicos

Um tipo lógico (ou booleano, alusão à Álgebra Booleana[6]) podem assumir apenas dois estados, denotados por true (verdadeiro) ou false (falso).

Tipo de dado lógico	Abrangência	Memória ocupada
boolean	true ou false	1 byte

Exemplo:
A:=5;
B:=7;
ESTADO:=(B<A);
Conteúdo da variável ESTADO: false

3.3 Constantes

As constantes são como variáveis, tem nomes, mas são fixas desde o início da execução do programa. O Turbo Pascal trabalha com dois tipos de constantes, os sem tipo definido e com tipo definido. As que não tem tipo definido são constantes verdadeiras, não podem ter seu valor alterado. Já as com tipo definido, podem ter seu valor alterado.
Formato: Identificador da Constante = valor da constante
Exemplos:
1) Constantes sem tipo definido

- PI = 3.1415927;
- DiasDoMesComercial = 30;
- AnoComercial = 350;

2) Constantes com tipo definido

- PI: real = 3.1415927;
- DiasDoMesComercial: integer = 30;
- AnoComercial: integer = 350;

[5]ASCII - American Standard Code for Interchanging Information - tabela de caracteres padrão para todos os computadores, permitindo desse modo que todas os computadores possam compartilhar dados.
[6]A álgebra booleana foi uma criação de um matemático, Boole.

3.4 Variáveis

As variáveis são um espaço na memória de um computador, previamente identificado e com a finalidade de armazenar os dados de um programa temporariamente. Uma variável armazena apenas um dado de cada vez. De acordo com o tipo de dado, a variável pode conter um dado lógico, inteiro, real ou caractere.

O nome de uma variável, assim como o nome de uma constante, é utilizado para a sua identificação e posterior uso dentro do programa. Existem algumas regras para serem utilizadas:

- Os nomes podem ter pelo menos um caracter;
- O primeiro caracter não pode ser numérico ($0\ a\ 9$);
- O nome não pode ter espaços em branco;
- O nome não pode ser uma palavra reservada do Pascal;
- O nome não pode ter outros caracteres a não ser números e letras, a não ser o caractere underscore ('_').

Formato: Identificador da variável: Tipo da variável
Exemplo:

- NomeDaVariavelInteira: integer;
- NotaDeAluno: real;
- C: array of boolean;

3.5 Conjuntos

No Pascal, os conjuntos são grupos de números ou caracteres que estão relacionados entre si, e usamos para verificar se um caracter ou número pertence ao conjunto. Por exemplo, um conjunto com as letras maiúsculas, de A a Z serve para verificarmos se outros caracteres pertencem a esse conjunto, o das letras maiúsculas. Usaremos a palavra reservada **set**.

Formato: IdentificadorDoConjunto: set of [conjunto ou tipo de dado];

Exemplo: Zero_ate_Nove: set of 0..9;

TodasAsLetras: set of char;

Ingredientes: set of (ovos, leite, manteiga, farinha);

3.6 Expressões aritméticas

Trabalharemos com expressões aritméticas para a resolução de cálculos, como o operador de atribuição (:=), operadores de igualdade (=), operadores aritméticos (+, -, *, /), entre outros. Particularmente, o operador de atribuição é usado para indicar que estamos armazenando numa variável o resultado de uma expressão aritmética.

Separamos parte das expressões por parênteses, para poder resolver na ordem correta. Se tivéssemos, por exemplo, a área do triângulo, que é dada

por $Area = \frac{Base\ .\ Altura}{2}$, em Pascal escreveríamos assim: $AREA := (BASE * ALTURA)/2$. Desse modo será calculado primeiro o numerador, para depois fazer a divisão pelo denominador.

Se tormarmos a fórmula da área do círculo teremos: $AREA = \pi\ .\ RAIO^2$. Em Pascal, obteríamos $AREA := 3.1415927 * RAIO * RAIO$. Note que colocamos o resultado da operação na variável $AREA$.

3.7 Bibliotecas ou units

As bibliotecas, ou units, são conjuntos de rotinas pré-compiladas que o Turbo Pascal, a partir da versão 4.0, introduziu. Nesse formato, você pode adicionar o conjunto de rotinas que for necessário, podendo deixar de fora outras que não tem utilidade para aquele programa. Usaremos a palavra reservada **uses**, que deve ser colocada na seção de cabeçalho do programa.

No Turbo Pascal, é necessário que a unit **CRT** (para manipular telas de texto) seja acrescentada, para que possamos imprimir na tela.

Exemplo: uses crt, math;

Adicionamos as bibliotecas CRT e MATH (para manipulação matemática).

3.8 Um primeiro programa em Pascal

Vamos construir como exemplo um primeiro programa em Pascal.

Algoritmo para cálculo de salário

1. Ler horas trabalhadas e armazenar na variável HT;
2. Ler hora trabalhada e armazenar na variável VH;
3. Ler percentual de desconto e armazenar na variável PD;
4. Calcular o salário bruto, multiplica HT por VH e coloca em SB;
5. Calcular o total de desconto TD, fazendo o valor de PD dividido por 100;
6. Calcular o salário líquido SL, deduzindo o desconto do salário bruto;
7. Apresentar os valores do salário bruto e líquido.

Explicação:

1. Começamos o corpo do programa com o comando **begin**, encerramos com **end** e um ponto final (.) para apontar o fim do corpo do programa.
2. Os comandos **write** e **writeln** servem para imprimir dados na tela, com uma diferença: **write** simplesmente imprime, enquanto que **writeln** gera um salto de linha e imprime (ln significa *line new*).
3. Os comandos **read** e **readln** fazem a leitura dos dados, e armazenamos nas variáveis que estão entre parênteses. A mesma diferença entre write e writeln vale para read e readln.

Algorithm 3 Programa em Pascal de cálculo do salário.

```
program SALARIO;
var

        HT, VH, PD, TD, SB, SL: real;

begin

        write(´Quantas horas de trabalho?´); readln(HT);
        write(´Qual o valor da hora trabalhada?´); readln(VH);
        write(´Qual o percentual de desconto?´); readln(PD);
        SB:=HT*VH;
        TD:=(PD/100)*SB;
        SL:=SB-TD;
        writeln(´Salario bruto.....:´,SB:7:2);
        writeln(´Desconto.....:´,TD:7:2);
        writeln(´Salario liquido....:´,SL:7:2);

end.
```

4. No comando writeln, os valores que estão depois das variáveis (**:7:2**) tem um motivo para estarem lá. O valor 7 nos garante que teremos até 7 ´casas´ separadas para a impressão do número. O valor 2, seguinte, diz-nos quantas casas decimais teremos para a parte fracionária. Logo, poderemos imprimir valores até 9999, contando uma casa para o sinal $(+ \; ou \; -)$, e 2 casas para a parte fracionária.

Outro exemplo:

Vamos desenvolver um pequeno programa em que vamos calcular a área de uma lata de óleo. Como entrada, temos a altura e o diâmetro da base da lata. A fórmula do volume é $VOLUME = \pi \; . \; \frac{D^2}{4} \; . \; ALTURA$.

3.9 Funções úteis do Pascal

Abaixo listaremos algumas funções muito úteis:

3.9.1 Funções algébricas e aritméticas:

abs(número) Retorna o valor absoluto do número especificado como parâmetro, e o tipo de dado de entrada corresponde ao tipo de dado da saída: Se número for Integer, a saída será Integer.

frac(número) Retorna a parte fracionária do número especificado. O resultado será Real.

exp(número) Retorna a exponencial do número especificado. O resultado será Real.

int(número) Retorna a parte inteira do número especificado, ou seja: O maior número inteiro menor ou igual a **número**. O resultado será Real.

Algorithm 4 Exemplo de algoritmo para cálculo do volume de uma lata.

```
program VOLUME_LATA;
const

        PI = 3.1415927;

var

        ALTURA, VOLUME, DIAMETRO: real;

begin

        writeln(´Qual o diâmetro do fundo da lata?´);
    read(DIAMETRO);
        writeln(´Qual a altura da lata?´); read(ALTURA);
        VOLUME:=(PI*ALTURA*(D*D))/4;
        writeln(´Volume da lata:´,VOLUME:7:2,´ cm3´);

end.
```

ln(número) Retorna o logaritmo neperiano do número especificado. O resultado será Real.

sqr(número) Retorna o quadrado do número especificado. O resultado será de acordo com o tipo de dado de saída.

sqrt(número) Retorna a raiz quadrada do número especificado. O resultado será Real.

random(número) Retorna um valor aleatório, onde $0 \leq valor\ aleatório \leq número$. O resultado será Integer.

3.9.2 Funções trigonométricas:

arctan(número) Retorna o arco-tangente do número especificado. O resultado será Real.

cos(número) Retorna o cosseno do número especificado. O resultado será Real.

sin(número) Retorna o seno do número especificado. O resultado será Real.

3.9.3 Funções diversas:

chr(número) Retorna o caractere com o valor ordinal dado pela expressão inteira **número.** O resultado será Char.

ord(variável) Retorna o número ordinal do valor **variável** no conjunto definido pelo tipo **variável**, que pode ser de qualquer tipo escalar, exceto Real. O resultado será Integer.

round(número) Retorna o valor de **número**, arredondado para o inteiro mais próximo. O resultado será Integer.

sizeof(nome) Retorna o número de bytes ocupado na memória pela variável ou tipo de dado **nome**. O resultado é do tipo Integer.

trunc(número) Retorna o valor de **número** com a parte decimal removida. O resultado será Integer.

3.10 Exercícios

Escreva para cada caso abaixo um programa em Pascal que:

1. Ler quatro números (variáveis A, B, C e D) e executar a seguinte expressão abaixo: $A \,.\, (B + C + D)$.

2. Calcular quantos litros de combustível gastos em uma viagem, supondo que o automóvel faça $12\ km/litro$. Para isso, o usuário deve fornecer o tempo gasto na viagem (variável TEMPO) e a velocidade média (variável VELOCIDADE). Use as fórmulas $DISTANCIA = TEMPO * VELOCIDADE$ e $LITROSGASTOS = DISTANCIA/12$. O programa também deverá apresentar os valores da velocidade média, tempo gasto na viagem, distância percorrida e quantidade de litros gasta.

3. Ler o valor da termperatura em graus Celsius (^oC) e apresentar o valor convertido para fraus Fahrenheit (^oF). A fórmula de conversão é: $F = \frac{9C+160}{5}$.

4. Fazer a leitura de três valores, armazenando-os nas variáveis A, B e C, e resolver a equação de $2^{\underline{o}}$ grau, apresentando as duas raízes (variáveis X1 e X2). Use uma variável, DELTA, para calcular o delta.

Capítulo 4

Estruturas de decisão

Até o momento, trabalhamos com entradas, processamentos e saídas, usando variáveis, constantes e operadores aritméticos. Mas até aqui, os recursos estudados são limitados. Haverá momentos onde precisaremos tratar um certo valor dentro de um programa, e em cima desse valor, tomar uma decisão, seguir um rumo no processamento.

Se temos um programa para calcular a média de um aluno, podemos usar o que conhecemos como **estrutura de decisão** para dizer se o aluno, segundo aquela média, está aprovado, reprovado ou em recuperação.

4.1 Desvio condicional simples

Para resolver o problema proposto acima (o da média do aluno), precisaremos definir um novo par de instruções, **if...then**. A instrução **if...then** tem por finalidade tomar uma decisão e efetuar um desvio no procedimento, dependendo da condição atribuída ser Verdadeira ou Falsa. Se a condição for Verdadeira, será executada a a instrução que estiver abaixo do **if...then**. Se precisarmos colocar mais de uma instrução, deveremos colocar a estrutura dentro de um bloco, definido com as instruções **begin** e **end**. A sintaxe do comando é:

if [condição] then

[instrução para condição verdadeira];

[instrução executada após a condição ser verdadeira];

A condição **[condição]** define o rumo do desvio: Se for Verdadeira, será executada a expressão **[instrução para condição verdadeira]**. Se falsa, a mesma não será executada. Essa expressão pode ser um bloco de instruções, na verdade. Logo, definiremos esse bloco com **begin** e **end**, usando um ponto-e-vírgula (;) após o **end**.

Exemplificaremos o funcionamento da estrutura abaixo, com dois exemplos:

Algoritmo de cálculo:

1. Ler variável A;
2. Ler variável B;

Algorithm 5 Programa-exemplo em Pascal para demonstrar o uso da instrução **if...then**.

```
program ADICIONA_NUMEROS;
var

        A, B, C: integer;

begin

        write(´Valor para A:´); readln(A);
        write(´Valor para B:´); readln(B);
        writeln;
        C:=A+B;
        if (C>10) then
          writeln(´O valor de C é:´,C);

end.
```

3. Calcular a soma de A e B, colocando os valores em C;
4. Apresentar o valor da soma contido em C, caso seja esse valor maior do que 10.

Algoritmo:

1. Ler variável A;
2. Ler variável B;
3. Verificar se A é maior do que B;
4. Se for verdadeiro, trocar os valores entre as variáveis. Se for falso, apenas apresentar o valor das variáveis.
5. Apresentar o valor das variáveis.

4.2 Desvio condicional composto

Poderemos fazer uma extensão ao uso das instruções **if...then**, estendendo-os para o caso de **if...then...else.** Sendo a condição verdadeira, será executada a instrução localizada entre os comandos **then** e **else**. Sendo a condição falsa, executaremos o que estiver abaixo do **else**. Se tivermos que usar mais do que um comando abaixo do **then** ou do **else**, precisaremos usar as palavras reservadas **begin** e **end** para definir um bloco de comandos. A sintaxe do comando é:

```
if [condição] then

        [instrução para condição verdadeira]

    else
    [instrução para condição falsa];
```

Algorithm 6 Programa-exemplo em Pascal para demonstrar o uso de **if...then**, com a uso de **begin** e **end**.

```
program ORDENA_DOIS_NUMEROS;
var

        A, B, C: integer;

begin

        write(´Valor para A:´); readln(A);
        write(´Valor para B:´); readln(B);
        writeln;
        if (A>B) then
        begin
          C:=A;
          A:=B;
          B:=C;
        end;
        writeln(´Os valores, agora ordenados, são:´);
        write(A, ´ ´,B);

end.
```

Logo, precisamos ressaltar que o **end**, seguido de ponto (.) só será usado no término do bloco principal do programa. o **end**, seguido de ponto-e-vírgula (;) é usado para terminar todos os blocos que precisarmos definir dentro do nosso programa. No caso da instrução que precede a instrução **else** deve ser escrita sem o ponto-e-vírgula. Isso ocorre porque o **else** é uma extensão do **if...then**, e sendo assim o final da condição somente ocorre depois do **else** ser processado.

Abaixo temos um exemplo:

4.3 Desvios aninhados

Podemos aninhar ou encadear vários desvios condicionais simples ou compostos, de modo que testemos múltiplas condições.

4.4 Desvios múltiplos

O Pascal e outras linguagens estruturadas tem o recurso de desvios múltiplos, representado pela palavra reservada **case**. Esse desvio consiste em uma variável seletora e uma lista de opções. Podemos usar o case para substituir o uso de múltiplos **if...then**. A sintaxe do comando é:

case [variável] of

[opção 1]: [operação 1]

[opção 2]: [operação 2]

...

[opção N]: [operação N]

else

Algorithm 7 Programa que mostra o uso do desvio condicional composto (**if...then...else**)

```
program USO_ELSE;
var

        NumId: integer;

begin

        write('Numero de identificacao:');
        readln(NumId);
        writeln;
        if not (NumId = 12345) then
          write('Nao eh o numero que estamos procurando.')
        else
          write('Então, descobrimos, é você!');

end.
```

Algorithm 8 Exemplo de desvios condicionais encadeados

```
if (a=1) then
  if (b=2) then
    writeln('Caso A')
  else
    if (b>2) then
      writeln('Caso B')
    else writeln('Caso C')
else writeln('Caso D');
```

Algorithm 9 Exemplo de duas estruturas com case.

```
case operador of

        ´+´:resultado := resposta + resultado;
        ´-´:resultado := resposta - resultado;
        ´*´:resultado := resposta * resultado;
        ´/´:resultado := resposta / resultado;

end;
case ano of

    Min..1939: begin

        tempo:=AntesSegundaGuerra;
        writeln(´O mundo em paz...´);

    end;
    1946..Max: begin

        tempo:=DepoisSegundaGuerra;
        writeln(´Reconstruindo o mundo...´);

    end;
    else begin

        tempo:=SegundaGuerra;
        writeln(´Estamos em guerra.´);

    end;

end;
```

[operação N+1]

end;

As operações podem ser apenas um comando ou um bloco de comandos, delimitados por **begin** e **end**. As opções podem ser um valor ou um subconjunto de valores. Esses subconjuntos podem ser delimitados por vírgulas (caso sejam constantes separadas) ou por dois pontos (".."). O tipo das constantes precisam ser do mesmo tipo da variável. Podemos trabalhar com quaisquer tipos de dados, excetos tipos reais.

4.5 Operadores

4.5.1 Operadores lógicos

Existem ocasiões onde precisamos encadear mais de uma condição ao mesmo tempo dentro do mesmo **if...then** para fazer testes múltiplos. Nesse caso, usaremos operadores lógicos, ou booleanos, que são três: **AND**, **OR** e **NOT**.

4.5.1.1 Operador AND

Usamos esse operador quando dois ou mais relacionamentos lógicos de uma condição precisam ser verdadeiros. Abaixo exibiremos a tabela verdade para esse tipo de operador:

Condição 1	Condição 2	Resultado
Falsa	Falsa	Falso
Verdadeira	Falsa	Falso
Falsa	Verdadeira	Falso
Verdadeira	Verdadeira	Verdadeiro

Desse jeito, a operação é executada se todas as condições interligadas com **AND** forem verdadeiras.

4.5.1.2 Operador OR

O Operador OR é usado quando precisamos que pelo menos um dos relacionamentos lógicos de uma condição forem verdadeiros. Abaixo temos a tabela verdade para esse operador:

Condição 1	Condição 2	Resultado
Falsa	Falsa	Falso
Verdadeira	Falsa	Verdadeiro
Falsa	Verdadeira	Verdadeiro
Verdadeira	Verdadeira	Verdadeiro

Logo, a operação será executada se pelo menos uma das condições interligadas com **OR** for verdadeira.

4.5.1.3 Operador NOT

O operador NOT é usado quando precisamos estabelecer que uma condição não pode ser verdadeira ou não pode ser falsa. Na verdade, o operador inverte o estado lógico de uma condição. Abaixo é apresentada a tabela verdade para esse operador:

Condição	Resultado
Verdadeiro	Falso
Falso	Verdadeiro

O operador **NOT** faz com que seja executada uma determinada operação, invertendo o resultado lógico da condição.

Abaixo temos um exemplo para os três operadores:

4.6 Exercícios

1) Com base nas tabelas-verdade, encontre o resultado lógico das expressões abaixo, sendo que: $X = -1,\ A = 3,\ B = 7,\ C = 8,\ D = 6$.

Algorithm 10 Programa-exemplo para demonstrar os operadores lógicos.

```
program TESTE_LOGICA;
var

        sexo: string[9];
        idade: integer;
        NumId: integer;

begin

        write ('idade:');
        readln(idade);
        writeln;
        if (idade>=16) and (idade<=65) then
              writeln('Idade certa para votar!');
        write('Sexo:');
        readln(sexo);
        writeln;
        if (sexo='masculino') or (sexo='feminino') then
              writeln('Sexo valido.');
        write('Numero de identificacao:');
        readln(NumId);
        writeln;
        if not (NumId = 12345) then
              write('Nao eh o numero que estamos procurando.');

end.
```

- NOT (X>3)
- (X>=2) OR (X<7)
- (X<1) AND NOT (B>D)
- (A>B) OR NOT (C>B)
- NOT (D<0) AND (C>5)
- 2 > 3
- (6 < 8) OR (3 > 7)
- (((10 DIV 2) MOD 6) > 5) NOT OR (3 < (2 MOD 2)) NOR (2 < 3)

2)Indique qual será o resultado de X nos trechos de código abaixo. Considere $A = 3,\ B = 2,\ C = 5,\ D = 7$.

- Resposta:

```
if not (D>5) then

        X:=(A+B)*D;

else

        X:=(A-B)/C;
```

```
writeln(X);
```

- Resposta:

```
if (A>2) and (B<7) then
        X:=(A+2)*(B-2);
else
        X:=(A+B)/D*(C+D);
writeln(X);
```

3) Faça um programa que leia os valores A, B, C e diga se a soma de A + B é menor que C.

4) Faça um programa que leia o nome e as três notas de uma disciplina de um aluno. No final escreva o nome do aluno, sua média e se ele foi aprovado (a média para aprovação é 7).

5) Faça um programa que leia 3 números inteiros e imprima o menor deles.

6) Dados três valores distintos, fazer um programa que, após a leitura destes dados coloque-os em ordem crescente.

7) Dados três valores X, Y, Z, verificar se eles podem ser os comprimentos dos lados de um triângulo, e se forem, verificar se é um triângulo equilátero, isósceles ou escaleno. Se eles não formarem um triângulo, escrever uma mensagem.

8) Monte um programa que lê um número representando um determinado mês do ano e escreve por extenso qual o mês lido. Caso o número digitado não esteja na faixa de 1 a 12 escreva uma mensagem informando o usuário do erro da digitação.

9) Faça um programa que leia um número qualquer. Caso o número seja par menor que 10, escreva **Número Par Menor que Dez**, caso o número digitado seja ímpar menor que 10 escreva **Número Ímpar Menor Que Dez**, caso contrário escreva **Número Fora Do Intervalo**.

10) Repita o programa para cálculo das raízes da equação de 2^{o} grau, fazendo o teste do delta: Menor do que zero, raízes complexas (não calcula); igual a zero, uma raiz dupla (calcula); maior do que zero, duas raízes reais e distintas (calcula).

Capítulo 5

Estruturas de repetição

As estruturas de repetição servem para fazer com que um comando ou um bloco de comandos sejam executados repetidamente, num loop, ou laço. Desse modo executaremos esse trecho tantas vezes quanto quisermos. Temos três tipos de estruturas de repetição com que podemos trabalhar.

5.1 Repetição com variável de controle

A primeira estrutura de repetição que usaremos usa contadores finitos. O laço termina depois de que o contador alcançou o valor desejado. Faremos uso de uma variável de controle, para garantir que a repetição acontecerá um determinado número de vezes. A palavra reservada a ser usada é **for**, e o seu funcionamento é controlado por esta variável de controle, podendo ser crescente ou decrescente. Usamos for quando sabemos antecipadamente quantas vezes será necessária a repetição daquele comando ou bloco de comandos (limitados por **begin** e **end**). A sintaxe do comando é:

for [variável]:=[início] to/downto [fim] do

[instruções]

- Decrescente:

for [variável]:=[início] downto [fim] do

[instruções]

Onde:

[variável] Variável de controle - só pode ser do tipo inteiro ou caractere.

[início]/[fim] limites inferior e superior do laço.

to/downto indicador de sequência crescente/decrescente.

Exemplo:

Algoritmode cálculo:

1. Definir um contador variando de 1 a 5;

Algorithm 11 Programa que mostra o uso da estrutura de repetição com variável de controle (**for**)

```
program USO_FOR;
var

        A, B, C, I: integer;

begin

        for I:=1 to 5 do
        begin
          write(´Entre um valor para A: ´);
          readln(A);
          write(´Entre um valor para B: ´);
          readln(B);
          writeln;
          R:=A+B;
          writeln(´O resultado correspondente é: ´,R);
          writeln;
        end;

end.
```

2. Ler dois valores, e colocar nas variáveis A e B;
3. Calcular a soma e colocar o resultado na variável C;
4. Apresentar o resultado;
5. Repetir os passos 2 a 4 até encerrar o contador.

Utilizando a instrução **for**, podemos fazer um programa simples para calcular o fatorial de um número: $n! = n \,.\, (n-1) \,.\, (n-2) \,.\, (n-3) \,. \ldots . \, 2 \,.\, 1$. Desse modo, quem determina o limite superior do laço (o campo **[fim]** na sintaxe) é o próprio usuário.

5.2 Repetição com teste lógico no início

Esse tipo de repetição usa um teste lógico no início do laço, verificando se é possível executar o comando ou o bloco de comandos que está abaixo. Para usarmos essa estrutura, usaremos as palavras reservadas **while...do**. A sintaxe é:

while [condição] do

[instruções para condição verdadeira]

Enquanto a condição for verdadeira, o laço será executado. Se a condição não for mudada em nenhum momento da execução do código, esse laço estará em **loop infinito**, ou seja, manterá a sua execução indefinidamente, até que o usuário aborte a sua execução.

Exemplo:

Algoritmode cálculo:

1. Crie uma variável que será o contador com valor inicial igual a 1;

Algorithm 12 Programa que calcula o fatorial de um número usando o comando **for**.

```
program FATORIAL_COM_FOR;
var

        A, N, FAT: integer;

begin

        FAT:=1;
        writeln(´Cálculo do fatorial de um número:´);
        write(´Fatorial de que número: ´); readln(N);
        for A:=1 to N do
          FAT:=FAT*A;
        writeln(´Fatorial de ´,N,´ equivale a ´,FAT);

end.
```

Algorithm 13 Programa que mostra o uso da estrutura de repetição com teste lógico no início (**while...do**)

```
program EXEMPLO_WHILE;
var

        A, B, R, I: integer;

begin

        I:=1;
        while (I<=5) do
        begin
          write(´Entre um valor para A:´); readln(A);
          write(´Entre um valor para B: ´); readln(B);
          R:=A+B;
          writeln(´O resultado é: ´,R);
          I:=I+1;
        end;

end.
```

2. Enquanto o contador for menor ou igual a 5, repetir os passos 3 a 5;
3. Leia os valores;
4. Calcular e colocar o valor em R;
5. Imprimir na tela o valor de R;
6. Somar um ao contador;
7. Se for maior do que um, encerra o processamento.

Criamos uma variável I para servir de contador. Essa variável controlará quantas vezes o trecho de código será executado. Enquanto o valor de I for menor do que 5, a condição será verdadeira. Desse modo, quando I for maior do que 5, a condição tornar-se-á falsa, e o laço será interrompido.

Algorithm 14 Programa que mostra o uso da estrutura de repetição com teste lógico no final, sem o uso de contador (**repeat...until**)

```
program EXEMPLO_WHILE_2;
var

        A, B, R: integer;
        RESPOSTA: string[3];

begin

        RESPOSTA:=´SIM´;
        while (RESPOSTA=´SIM´) or (RESPOSTA=´S´) do
        begin
          write(´Entre um valor para A:´); readln(A);
          write(´Entre um valor para B: ´); readln(B);
          R:=A+B;
          writeln(´O resultado é: ´,R);
          write(´Deseja continuar?(SIM/NAO)´); readln(RESPOSTA);
        end;

end.
```

Mas por vezes, poderemos querer montar um laço onde não saberemos quando ele vai acabar. Pode ser executado uma, duas, ou mil vezes. Queremos que o laço seja interrompido quando o usuário quiser, o que pode ser aferido com uma pergunta. Logo, podemos construir um programa do seguinte modo:

Exemplo:

Algoritmode cálculo:

1. Crie uma variável que será usada como resposta;
2. Enquanto a resposta for sim, executar os passos 3 a 5;
3. Ler os valores;
4. Efetuar o cálculo, colocando o resultado em R;
5. Apresentar o valor calculado contido em R;
6. Quanto a resposta for diferente de sim, encerrar o processamento.

Por último, vamos repetir o programa para cálculo do fatorial de n, $n!$, que desenvolvemos usando o comando **for**, para uso com o **while**.

Algorithm 15 Programa que calcula o fatorial de um número usando o comando **while**.

```
program FATORIAL_COM_WHILE;
var

        A, N, FAT: integer;

begin

        FAT:=1;
        A:=1;
        writeln(´Cálculo do fatorial de um número:´);
        write(´Fatorial de que número: ´); readln(N);
        while (A<=N) do
        begin
          FAT:=FAT*A;
          A:=A+1;
        end;
        writeln(´Fatorial de ´,N,´ equivale a ´,FAT);

end.
```

5.3 Repetição com teste lógico no fim

Existe uma terceira estrutura de repetição que é usada no Pascal, e nessa o teste lógico é realizado no final de um laço. Essa estrutura parece com a estrutura **while...do**, e é na verdade a estrutura **repeat...until**. Seu funcionamento é controlado pela decisão, e desse modo, o laço será executado pelo menos uma vez antes de ser encerrado, pois o teste lógico é no final. Se a condição, no caso da estrutura **while...do** é falso, o laço não é nem executado.

Dessa forma, o **repeat** funciona ao contrário do **while**, pois sempre irá processar um conjunto de instruções, pelo menos uma vez, até que a condição seja verdadeira. No caso do **repeat**, enquanto a condição for falsa, o conjunto será executado. A sintaxe é:

repeat

[instruções até que a condição seja verdadeira]

until [condição];

Exemplo:

Algoritmode cálculo:

1. Crie uma variável que será usada como contador com valor inicial 1;
2. Ler os valores;
3. Efetuar o cálculo, colocando o resultado em R;
4. Apresentar o valor calculado contido em R;
5. Incrementar o contador;
6. Repetir os passos 2 a 5 até que o contador seja maior do que 5.

Algorithm 16 Programa que mostra o uso da estrutura de repetição com teste lógico no final (**repeat...until**)

```
program EXEMPLO_REPEAT;
var

        A, B, R, I: integer;

begin

        I:=1;
        repeat
          write(´Entre um valor para A:´); readln(A);
          write(´Entre um valor para B: ´); readln(B);
          R:=A+B;
          writeln(´O resultado é: ´,R);
          I:=I+1;
        until (I>5);

end.
```

Criamos uma variável, I, para servir de contador, para controlar quantas vezes o trecho de código será executado. Inicializamos a variável I com o valor 1. Devido ao **repeat**, todo trecho de instrução situado até o **until** deverá ter o seu processamento repetido até que a sua condição seja verdadeira. Logo, enquanto o valor de I for menor do que 5, a condição será falsa, e o trecho será executado. Desse modo, quando I for maior do que 5, a condição tornar-se-á verdadeira, e o laço será interrompido.

Semelhante à estrutura **while...do**, o **repeat...until** pode ser parado não por um contador, mas pela mudança da condição lógica. O laço será interrompido quando o usuário quiser, o que pode ser aferido com um teste de uma condição lógica. Logo, podemos construir um programa do seguinte modo:

Exemplo:

Algoritmode cálculo:

1. Crie uma variável que será usada como resposta;
2. Ler os valores;
3. Efetuar o cálculo, implicando o resultado em R;
4. Apresentar o valor calculado contido em R;
5. Perguntar ao usuário se deseja continuar executando o programa;
6. Repetir os passos 2 a 5 até que a resposta do usuário seja não.

Por último, veremos um exemplo do cálculo do fatorial de n, $n!$, desenvolvido antes com **for...do** e **while...do**, e agora com **repeat...until**.

Algorithm 17 Programa que mostra o uso da estrutura de repetição com teste lógico no final, sem o uso de contador (**repeat...until**)

```
program EXEMPLO_REPEAT_2;
var

        A, B, R: integer;
        RESPOSTA: string[3];

begin

        RESPOSTA:=´SIM´;
        repeat
          write(´Entre um valor para A:´); readln(A);
          write(´Entre um valor para B: ´); readln(B);
          R:=A+B;
          writeln(´O resultado é: ´,R);
          write(´Deseja continuar?(S/N)´); readln(RESPOSTA);
        until ((RESPOSTA<>´SIM´) and (RESPOSTA<>´sim´));

end.
```

Algorithm 18 Programa que calcula o fatorial de um número usando a estrutura **repeat...until**.

```
program FATORIAL_COM_REPEAT;
var

        A, N, FAT: integer;

begin

        FAT:=1;
        A:=1;
        writeln(´Cálculo do fatorial de um número:´);
        write(´Fatorial de que número: ´); readln(N);
        repeat
          FAT:=FAT*A;
          A:=A+1;
        until (A>N);
        writeln(´Fatorial de ´,N,´ equivale a ´,FAT);

end.
```

5.4 Exercícios

Use as estruturas **for**, **while** e **repeat** para criar os programas abaixo:

1. Imprimir na tela todos os valores numéricos inteiros ímpares entre 0 e 100.
2. Imprimir na tela a soma de cem números inteiros, sendo o primeiro dado pelo usuário.
3. Ler um número N, onde $1 \leq N \leq 50$ (lembre-se de testar se N está entre esses valores), e apresentar o valor obtido da multiplicação sucessiva de N por 2 enquanto o produto for maior do que 250 ($N * 2,\ N * 2 * 2,\ N * 2 * 2 * 2,\ etc.$), e por fim, apresentar o resultado.
4. Apresentar os quadrados dos números inteiros entre 15 e 200.
5. A sequência de Fibonacci é formada pela soma de um termo posterior com o seu anterior subseqüente. Logo, a seqüência formada é $1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ 34, ...$ Faça um programa que calcule essa série até o n-ésimo termo.
6. Faça um programa para somar os números pares positivos < 1000 e ao final imprimir o resultado.
7. Leia o valor x e calcule $y = x + 2x + 3x + 4x + ... + 20x$
8. Leia e escreva o nome, idade e sexo de um número indeterminado de alunos. Ao final escreva o total de alunos lidos.
9. Leia 20 valores reais e escreva o seu somatório.
10. Dada uma frase de exatamente 80 caracteres, escreva a frase de trás para frente, um caracter por linha.
11. Faça um programa que leia dois valores, x e n, e calcule $y = x - 2x + 4x - 6x + ... + nx$.

Capítulo 6

Variáveis compostas homogêneas

Vimos no início, que podemos dar um nome a uma posição de memória, e nesta teremos associado um valor qualquer. Essa posição de memória com nome é o que chamamos de **variável** (subseção 2.4.2.3). Só que às vezes esta forma de alocação de memória não é suficiente para resolver certos problemas: Imagine como faríamos para construir um algoritmo que lesse o nome de n pessoas e que imprimisse um relatório destes nomes, agora ordenados alfabeticamente? Seria complicado, se tivéssemos 1000 pessoas, por exemplo, teríamos que definir 1000 variáveis do tipo string, para comportar todos os nomes:

Considere o tamanho do algoritmo e o trabalho braçal para construí-lo. Isto se tivéssemos apenas 1.000 nomes. E se fossem 1.000.000 de pessoas? A construção deste algoritmo fica inviável na pratica. Para resolver problemas como este e outros, foi criado um novo conceito para alocação de memória sendo desta forma também criado uma nova maneira de definir variáveis, a qual foi denominada de **variável indexada**.

Uma variável indexada corresponde a uma seqüência de posições de memória, a qual daremos um único nome, sendo que cada uma destas pode ser acessada através do que conhecemos por índice. O índice corresponde a um valor numérico (que não pode ser real), ou a um valor caracter (que não pode ser string). Cada uma das posições de memória de uma variável indexada pode receber valores no decorrer do algoritmo

Algorithm 19 Trecho de programa demonstrando a definição de 1000 variáveis, uma a uma.

```
program Loucura;
var
  Nome1,
  Nome2,
  Nome3,
  ....
  Nome999,
  Nome1000: string;
begin
   <Comandos>
end.
```

Algorithm 20 Exemplo de uso da palavra reservada **array,** com vetores.

```
program Exemplo;
var
  Vet: array [1..10] OF REAL;
begin
  <Comandos>;
end.
```

Algorithm 21 Exemplo de programa fazendo uso de um vetor.

```
program Atribui
var
  Nomes: array [1..20] OF string;
  I: integer;
begin
  Nomes[1]:=João da Silva;
  for i:=2 to 20 do
    READ(Nomes[i]);
end.
```

como se fosse uma variável comum, a única diferença reside na sintaxe de utilização desta variável.

6.1 Vetores

Os vetores são variáveis indexadas unidimensionais, que, como o próprio nome diz, possui apenas uma dimensão, sendo ser possível definir variáveis com quaisquer tipo de dados válidos do Pascal. A definição é:

var
<Nome>: array [INICIO..FIM] OF <tipo>;

Obs.:
a) array é uma palavra reservada do Pascal;
b) Os valores INICIO e FIM correspondem aos índices inicial e final;
c) Uma variável indexada pode ser apenas de um tipo de dado;

Exemplo:
Definir uma variável indexada como sendo do tipo real, sendo que a mesma deverá corresponder a 10 posições de memória.
No exemplo acima, a atribuição de valores é feita do seguinte modo:
VET

1	2	3	4	5	6	7	8	9	10

Estes valores numéricos correspondem ao índice da variável. Atribuímos os valores do seguinte modo:

<Nome>[<Índice>]: =Valor;

Logo, usaremos o índice entre colchetes, para definir em que posição de memória contida naquele vetor estaremos usando.
Exemplo:

Algorithm 22 Exemplo de programa fazendo uso de um vetor para armazenamento.

```
program LISTA_NOME;
var
  Nome: array[1..10] OF string[40];
  I: integer;
begin
  writeln(´Listagem dos nomes:´);
  {Entrada dos dados}
  for I:=1 to 10 do
    begin
      write(´Digite o ´,I:2,´ o. nome: ´);
      readln(NOME[I]);
    end;
  writeln;
  {Saida dos dados}
  for I:=1 to 10 do
    writeln(´Nome: ´,I:2,´ —> ´,NOME[I]);
  writeln;
end.
```

Outro exemplo:

Algoritmo:

1. Definir a variável I do tipo inteira para controlar a malha de repetição;
2. Definir o vetor NOME do tipo caracter para 10 elementos;
3. Leia os 10 nomes;
4. Apresentar na tela os 10 nomes.

6.2 Matrizes

As matrizes são variáveis indexadas multidimensionais, que, como o próprio nome diz, possui mais do que uma dimensão (de duas até quantas possíveis forem), sendo ser possível definir variáveis com quaisquer tipo de dados válidos do Pascal. A definição é:

var
<Nome>: array [INICIO1...FIM1,INICIO2...FIM2,...INICION...FIMN] OF <tipo>;

Aqui, INICIO1, FIM1, INICIO2, FIM2,...INICION, FIMN são os índices das dimensões da matriz.

Exemplo:

Definir uma variável indexada bidemensional para armazenar os dados de uma matriz 4 por 4 de números do tipo REAL, sendo que a mesma deverá corresponder no total a 16 posições de memória.

No exemplo acima, após a definição da variável, a memória estará como mostrado no esquema abaixo:

Algorithm 23 Exemplo de uso para a palavra **array**, com matrizes.

```
program Exemplo;
var
  MAT: array [1..4,1..4] OF REAL;
begin
  <Comandos>;
end.
```

Algorithm 24 Exemplo de programa fazendo uso de uma matriz.

```
program Atribui;
var
  Nomes: array[1..4,1..4] OF string;
  I,J: integer;
begin
  FOR  I: = 1 TO 4 DO
    FOR  J := 1 TO 4 DO
      READ (Nomes[ I,J] );
end.
```

MAT:

1,1	1,2	1,3	1,4
2,1	2,2	2,3	2,4
3,1	3,2	3,3	3,4
4,1	4,2	3,4	4,4

Os valores numéricos apresentados acima correspondem aos índices da variável. Nesse caso, a atribuição de valores é feita do seguinte modo:

<Nome>[<Índice>,<Índice>]: =Valor;

Separaremos os índices por vírgulas, para colocar o índice certo.

Exemplo:

Podemos criar vetores e matrizes de quaisquer dimensões e tipos (inteiros, caracteres, reais e lógicos), e manipulá-las usando estruturas de repetição, decisão, entre outros. Apenas devemos lembrar de usar corretamente os índices, para que possamos usar o conteúdo certo do vetor, na posição desejada.

Vejamos outro exemplo, de um programa de agenda para cadastrar nome, endereço, CEP, bairro e telefone de até 10 pessoas.

Algoritmo:

1. Definir a matriz DADO do tipo caracter para 5 x 10 elementos;
2. Entrar os nomes em sequência;
3. Perguntar se deseja continuar. Se sim, volta ao passo 2;
4. Se não, imprime o resultado na tela e finaliza o programa.

Algorithm 25 Exemplo de programa que usa uma matriz para armazenamento.

```
program AGENDA;
var
  DADO: array[1..10,1..5] of string;
  I: integer;
  RESP: char;
  X: string;
begin
  writeln(´Programa agenda´);
  writeln;
  I:=0;
  RESP:=´S´;
  while (RESP=´S´) or (RESP=´s´) do
  begin
    I:=I+1;
    write(´Nome:´);
    readln(DADO[I,1]);
    write(´Endereço:´);
    readln(DADO[I,2]);
    write(´CEP:´);
    readln(DADO[I,3]);
    write(´Bairro:´);
    readln(DADO[I,4]);
    write(´Telefone:´);
    readln(DADO[I,5]);
    writeln;
    write(´Deseja continuar (S/N)?´);
    readln(RESP);
  end;
  for J:=1 to I do
  begin
    writeln(´Nome—>´,DADO[J,1]);
    writeln(´Endereço->´,DADO[J,2]);
    writeln(´CEP—->´,DADO[J,3]);
    writeln(´Bairro—>´DADO[J,4]);
    writeln(´Telefone–>´DADO[J,5]);
    writeln;
    writeln(´Tecle [ENTER] para continuar´);
    readln;
  end;
end.
```

6.3 Aplicações práticas

Podemos usar vetores e matrizes em uma ampla gama de opções dentro da programação. Existem algoritmos publicados na literatura que nos são úteis para a resolução de problemas.

6.3.1 Ordenação

Algoritmos de ordenação são muito usados em programação, quando precisamos colocar em ordem crescente um conjunto de dados, sejam numéricos ou alfanuméricos. Existem dezenas de algoritmos, que vão desde os lentos e de fácil compreensão, aos rápidos e complexos. Observaremos aqui um dos mais comuns e simples, conhecido também como ordenação por troca, método de ordenação "da bolha", ou **bubble sort**.

O nome bubble sort vem da idéia de que os elementos mais "leves" tendem a "subir" na lista, enquanto que os mais pesados descem para o "fundo". O algoritmo começa a vasculhar a partir do início do vetor a ser ordenado, e executa a ordenação até chegar ao final do vetor. O mesmo compara cada elemento com o seu elemento anterior. Caso o procedente seja maior, é feita a troca. Caso contrário, é mantida a ordem. O algoritmo termina quando alcança o início do vetor.

Exemplificaremos abaixo o funcionamento do algoritmo. Suponhamos que o nosso vetor tenha 5 dados:

Índice	Elemento
1	9
2	8
3	7
4	5
5	3

Após a primeira passagem, teremos que os dados na tabela, ao serem comparados, serão trocados de posição, e a tabela ficará assim:

Índice	Elemento
1	3
2	9
3	8
4	7
5	5

Logo, vemos que ainda não é o bastante, pois o menor valor tornou-se o primeiro, mas ainda precisamos fazer mais algumas passagens, para ordenar o resto do vetor. Logo, não precisaremos mexer no elemento cujo índice é 1, pois ele já é o menor de todos. Poderemos fazer com que a nossa estrutura de repetição a partir do elemento de índice 2. Logo, teremos, numa segunda passagem, o seguinte resultado:

Algorithm 26 Trecho de código em Pascal exemplificando o método de ordenação por troca.

```
FOR I:=1 to N-1 DO
  FOR J:=I+1 to N DO
    IF (VETOR[I]>VETOR[J]) THEN
    begin
      A:=VETOR[I];
      VETOR[J]:=VETOR[I];
      VETOR[J]:=A;
    end;
```

Índice	Elemento
1	3
2	5
3	9
4	8
5	7

E por aí vai, até a quarta passagem, onde obteremos o seguinte resultado:

Índice	Elemento
1	3
2	5
3	7
4	8
5	9

Em Pascal, o algoritmo ficaria escrito do seguinte modo:

Note que temos duas estruturas de repetição (FOR) aninhadas, uma controlada pela variável de controle I e outra, pela variável J. Somente quando J atinge o valor de N, é que o laço se encerra. Nesse caso, teremos que:

Quando I for	J será
1	2,3,4,5,...,N
2	3,4,5,...,N
3	4,5,...,N
...	...
N-1	N

Note que fizemos a comparação entre dois valores do vetor, que estão em posições diferentes (VETOR[I] e VETOR[J]). Caso seja correto, usaremos também uma variável auxiliar (A) para fazer a troca dos valores.

Podemos comparar quaisquer valores do mesmo tipo, inclusive do tipo caracter. Na tabela ASCII, temos que a letra "A" precede a letra "B", e assim por diante. Também temos que as letras maiúsculas precedem as letras minúsculas. Logo, se fizermos a seguinte comparação:

´FABIO´<´RICARDO´, teremos que o resultado será **TRUE**.

Podemos então definir o algoritmo:

Algorithm 27 Exemplo de programa usando um vetor para armazenamento e apresentando o resultado ordenado.

```
program LISTA_ORDENADA_NOME;
var
  Nome: array[1..10] OF string[40];
  X: string[40];
  I,J: integer;
begin
  writeln(´Ordenação dos nomes:´);
  {Entrada dos dados}
  for I:=1 to 10 do
    begin
      write(´Digite o ´,I:2,´ o. nome: ´);
      readln(NOME[I]);
    end;
  writeln;
  {Ordenacao dos dados}
  for I:=1 to 9 do
    for J:=I+1 to 10 do
      if (NOME[I]>NOME[J]) then
      begin
        X:=NOME[I];
        NOME[J]:=NOME[I];
        NOME[J]:=X;
      end;
  {Saida dos dados}
  for I:=1 to 10 do
    writeln(´Nome: ´,I:2,´ —> ´,NOME[I]);
  writeln;
end.
```

Algoritmo:

1. Definir a variável I do tipo inteira para controlar a malha de repetição;
2. Definir o vetor NOME do tipo caracter para 10 elementos;
3. Leia os 10 nomes;
4. Colocar em ordem crescente os elementos do vetor;
5. Apresentar na tela os 10 nomes.

O método da bolha é simples e eficiente para vetores pequenos. Quando trabalhamos com vetores muito grandes (com centenas ou milhares de posições), vemos que ele é ineficiente, por levar muito tempo nos laços aninhados. Existem outros algoritmos mais eficientes e rápidos, como a ordenação shell (**shell sort**) e ordenação por segmentação (**quick sort**), que podem servir apropriadamente para resolução desses problemas.

6.3.2 Pesquisa

Usando vetores e matrizes, poderemos rapidamente ter tabelas muito grandes, onde seria difícil localizar um elemento de uma forma rápida no meio

de tantos. Devido a isso foram desenvolvidos algoritmos para fazer pesquisas em vetores e matrizes, de modo a localizar um elemento dentro de um vetor.

6.3.2.1 Pesquisa seqüencial

Esse método consiste em fazer a procura seqüencialmente, começando do primeiro elemento até o último. Localizando o elemento desejado, ele é apresentado. Este método é lento mas eficiente quando o vetor está com os dados desordenados.

Exemplificando:

Vetor de dados:

3	1	15	9	5	78	39	56	91

Objeto de procura: 78
Compara com o dado na posição 1 do vetor: Não.
Compara com o dado na posição 2 do vetor: Não.
Compara com o dado na posição 3 do vetor: Não.
Compara com o dado na posição 4 do vetor: Não.
Compara com o dado na posição 5 do vetor: Não.
Compara com o dado na posição 6 do vetor: Sim.
Localizamos o valor desejado.
Total de iterações para localizar o dado: 6.

Algoritmo:

1. Inicializa o contador com 1.
2. Repete enquanto o contador não atingir o valor máximo, ou o elemento a ser procurado não for encontrado.
3. Lê o elemento do vetor (ou matriz) que está na posição dada pelo contador.
4. Compara com o elemento a ser procurado.
5. Se forem diferentes, incremente o contador e vá para o passo 2.
6. Se forem iguais, finalize o laço e passe para o passo 7.
7. Apresente na tela a posição do vetor onde foi encontrado o resultado da pesquisa.

6.3.2.2 Pesquisa binária

O segundo método de pesquisa é em média mais rápido que a pesquisa seqüencial, só que para funcionar, o vetor deve estar previamente ordenado. Esse método divide o vetor em duas partes, e procura ver se o objeto de procura está acima ou abaixo da linha de divisão. Localizado em que intervalo o dado está, o outro intervalo é descartado, e verifica-se se a informação foi encontrada. Senão, é novamente dividida em duas partes, e assim por diante. O nome pesquisa binária é dado por causa da ação do algoritmo em dividir o volume de dados em duas partes.

Exemplificando:

Algorithm 28 Exemplo de programa fazendo uso de pesquisa sequencial.

```
program PESQUISA_SEQUENCIAL;
var
  Nome: array[1..10] OF string[40];
  PESQ: string[40];
  I: integer;
  RESP: string[3];
  ACHA: BOOLEAN;
begin
  writeln(´Pesquisa sequencial dos nomes:´);
  {Entrada dos dados}
  for I:=1 to 10 do
    begin
      write(´Digite o ´,I:2,´ o. nome: ´);
      readln(NOME[I]);
    end;
  writeln;
  {Pesquisa dos dados}
  RESP:=´SIM´;
  while (RESP=´SIM´) or (RESP=´sim´) do
  begin
    write(´Entre o nome a ser pesquisado:´);
    readln(PESQ);
    I:=1;
    ACHA:=false;
    while (I<=10) and (ACHA=false) do
    if (PESQ=NOME[I]) then
      ACHA:=true
    else
     I:=I+1;
    if (ACHA=true) then
      writeln(PESQ,´ foi localizado na posição ´,I:2);
    else
      writeln(PESQ,´ não foi localizado´);
    writeln;
    write(´Deseja continuar? (SIM/NAO):´);
    readln(RESP);
  end;
end.
```

- Vetor de dados:

1	3	5	9	15	39	56	78	91

Objeto de procura: 78
Divisão: O vetor tem 9 posições, tomaremos para divisão a posição $5 = \frac{9+1}{2}$.
Dado na posição 5 do vetor de dados: 15.
Está abaixo ou acima? Acima.
Fazemos o descarte das posições 1 a 5.

- Vetor de dados:

39	56	78	91

Objeto de procura: 78
Divisão: O vetor tem 4 posições, tomaremos para divisão a posição $2 = \frac{4+1}{2}$ (lembre-se que a divisão é inteira, com resto).
Dado na posição 2 do vetor de dados: 56.
Está abaixo ou acima? Acima.
Fazemos o descarte das posições 1 a 2.

- Vetor de dados:

78	91

Objeto de procura: 78
Divisão: O vetor tem 2 posições, tomaremos para divisão a posição $1 = \frac{2+1}{2}$ (lembre-se que a divisão é inteira, com resto).
Dado na posição 1 do vetor de dados: 78.
Localizamos o valor desejado.
Total de iterações para localizar o dado: 3.

Algoritmo:

1. Divide o vetor em duas partes;
2. Verifica se a informação a ser pesquisada está acima ou abaixo da divisão.
3. Se estiver acima, a metade abaixo da linha de divisão é desprezada.
4. Se estiver abaixo, a metade acima é descartada.
5. Se a informação não foi encontrada, fazemos nova divisão no pedaço do vetor que não foi descartado e volta para o passo 1

Algorithm 29 Exemplo de programa fazendo uso de pesquisa binária.

```
program PESQUISA_BINARIA;
var
  Nome: array[1..10] OF string[40];
  PESQ, RESP, X: string[40];
  I,J, COMECO, MEIO, FIM: integer;
  ACHA: boolean;
begin
  writeln(´Pesquisa binária:´);
  {Entrada dos dados}
  for I:=1 to 10 do
    begin
      write(´Digite o ´,I:2,´ o. nome: ´);
      readln(NOME[I]);
    end;
  writeln;
  {Ordenacao dos dados}
  for I:=1 to 9 do
    for J:=I+1 to 10 do
      if (NOME[I]>NOME[J]) then
      begin
        X:=NOME[I];
        NOME[J]:=NOME[I];
        NOME[J]:=X;
      end;
  {Pesquisa binária}
  RESP:=´SIM´;
  while (RESP=´SIM´) or (RESP=´sim´) do
  begin
   write(´Nome a ser pesquisado:´);
   readln(PESQ);
   COMECO:=1;
   FINAL:=10;
   ACHA:=false;
   while (COMECO<=FINAL) and (ACHA=false) do
   begin
    MEIO:=(COMECO+FINAL) div 2;
    if (PESQ=NOME[MEIO]) then
     ACHA:=true
    else
     if (PESQ<NOME[MEIO]) then
      FINAL:=MEIO-1;
     else
      COMECO:=MEIO+1;
   end;
  {Saida dos dados}
   if (ACHA=true) then
    writeln(PESQ,´ foi localizado na posicao ´,MEIO:2);
   else
    writeln(PESQ,´ não foi localizado);
   writeln;
   write(´Deseja continuar (SIM/NAO):´);
   readln(RESP);
  end;
end.
```

6.4 Exercícios:

1) Faça um programa que leia, via teclado, 200 valores do tipo inteiro, os guarde na memória, imprima na tela: todos os números pares que você leu, o maior e o menor valor nesse vetor, e os valores que são maiores do que a média desses valores todos.

2) Faça um programa que leia 10 nomes, guarde na memória, e imprima na tela o maior e o menor nome do vetor.

3) Faça um programa que leia 20 palavras, e após a leitura, inverta os caracteres de cada uma das palavras.

4) Faça um programa que leia 10 nomes e os guarde na memória. Após a leitura, emita um relatório com todos os nomes que são palíndromos. Uma palavra palindroma é aquela que a sua leitura é a mesma da esquerda para a direita e vice versa. Exemplo: ARARA, ANA, etc.

5) Faça um programa que leia, Nome idade e sexo de N pessoas. Após a leitura faça:
a) Imprima o Nome, idade e sexo das pessoas cuja idade seja maior que a idade da primeira pessoa.
b) Imprima o Nome e idade de todas as mulheres.
c) Imprima o Nome dos homens menores de 21 anos.

6) Faça um programa que leia 20 valores e imprima os que são maiores que a média dos valores.

7) Faça um programa que leia nome e 4 notas de N alunos de um colégio. Após a leitura faça:
a) Imprima o nome e a média dos alunos aprovados (Média >= 7.0).
b) Imprima o nome e a média dos alunos em Recuperação (5.0 >= Média < 7.0).
c) Imprima o nome e a média dos alunos reprovados (Média < 5.0).
d) Imprima o percentual de alunos aprovados.
e) Imprima o percentual de alunos reprovados.

8) Faça um programa que crie uma tabela no vídeo do computador com todos os caracteres ASCII, sendo que deverá ser impresso o caracter, bem como o seu valor decimal.

9) Faça um programa para ler 50 valores inteiros. Após imprima tais valores em ordem crescente.

10) Dada uma frase, faça um programa que determine qual a consoante mais utilizada.

11) Faça um programa que:
a) Leia um vetor com N elementos formado por valores do tipo inteiro.
b) Após a leitura, modifique o vetor de forma que o mesmo contenha na parte superior somente valores pares, e na parte inferior os valores impares.
c) Ordene crescentemente os números pares, e decrescentemente os números impares.

12) Em uma cidade do interior, sabe-se que, de janeiro a abril de 1976 (121 dias), não ocorreu temperatura inferior a 15o C nem superior a 40o C. As temperatura verificadas em cada dia estão disponíveis em uma unidade de entrada de dados.

Fazer um programa que calcule e escreva:
a)a menor temperatura ocorrida;
b)a maior temperatura ocorrida;
c)a temperatura média
d)o número de dias nos quais a temperatura foi inferior a média à temperatura média.

14) Faça um programa que:
a)leia uma variável indexada A com 30 valores reais;
b)leia uma outra variável indexada B com 30 valores reais;
c)leia o valor de uma variável X;
d)verifique qual o elemento de A é igual a X;
e)escreva o elemento de B de posição correspondente à do elemento A igual a X;

15) Faça um programa para ler e imprimir uma matriz 2x4 de números inteiros.

16) Dado uma matriz de ordem 3x3 faça um programa que:
a) Calcule a soma dos elementos da primeira coluna;
b) Calcule o produto dos elementos da primeira linha;
c) Calcule a soma de todos os elementos da matriz;
d) Calcule a soma da diagonal principal;

17) Dado uma matriz de ordem $n\ x\ n$ faça um programa que verifique se a matriz é simétrica ($a_{ij} = a_{ji}$).

18) Dada uma matriz $m\ x\ n$ de valores reais faça um programa que faça a leitura destes valores e ao final da leitura de todos, imprimir o relatório:
a) Qual a soma dos valores de cada coluna da matriz;
b) Listar os valores que são menores que a média dos valores;
c) Qual a soma dos elementos da diagonal secundária;

19) Dada uma matriz $m\ x\ n$ de valores inteiros faça um programa que leia estes valores e ao final coloque os elementos ordenados primeiro pela linha e depois pela coluna.

20) Dadas duas matrizes A e B de ordem $n\ x\ n$, faça um programa que some as duas e gere a matriz C. Os elementos da matriz C são a soma dos respectivos elementos de A e B.

21) Dada uma matriz $m\ x\ n$ de valores inteiros determine a sua matriz transposta e imprima.

22) Faça um programa que efetue um produto matricial. Seja A($m\ x\ n$) e B ($m\ x\ n$) as matrizes fatores, sendo $m \leq 40$ e $n \leq 70$. Deverão ser impressas as matrizes A, B e a matriz-produto obtida.

Capítulo 7

Sub-rotinas, procedimentos e funções

No geral, problemas complexos exigem algoritmos igualmente complexos, mas é possível dividir um problema grande em problemas menores, subdividindo-o em partes menores, demandando algoritmos mais simples. Esse algoritmo simples é o que chamamos sub-rotina. Uma sub-rotina pode ser chamada por qualquer ponto do programa principal, ou mesmo por outra sub-rotina. Terminado o seu uso, o controle volta ao programa principal, para a linha de instrução seguinte à linha que efetuou a chamada à sub-rotina.

Trabalhando com essa técnica, podemos precisar dividir uma sub-rotina em outras tantas quantas forem necessárias, buscando uma solução mais simples do problema.

7.1 Tipos

O Pascal traz suporte a 3 tipos de sub-rotinas: **Unidades** (units), **Procedimentos** (procedure) e **Funções** (function). Na verdade, uma procedure ou uma function é um bloco de programa, contendo início e fim, podendo definir suas próprias variáveis, constantes e tipos, sendo chamada por um nome. A procedure ou a function pode ser referenciada em qualquer parte do programa principal nas sub-rotinas que estão abaixo na listagem do programa.

Mas, além dos procedimentos e funções, no Pascal (especificamente no Turbo Pascal, da versão 4 em diante), você pode encontrar conjuntos de rotinas embutidas, as unidades ou units.

7.2 Unidades, ou units

As units são conjuntos de rotinas prontas, pré-compiladas para serem usadas pelo programador. Abaixo vai uma rápida lista das units básicas do Turbo Pascal e sua função:

CRT : Contém a maior parte das rotinas usadas para controle e geração de som, controle de teclado e vídeo. Talvez seja a unidade mais usada de todas.

DOS: Contém rotinas para utilização de recursos do sistema operacional.

GRAPH: Contém rotinas para utilização da capacidade gráfica do computador.

OVERLAY : Permite gerenciar as atividades de um programa, como compartilhar uma região da memória para executar rotinas diferentes, por exemplo.

PRINTER: Permite lidar com a impressora, como associar um arquivo do tipo texto e associá-lo ao dispositivo LST.

SYSTEM: Contém a maior parte das rotinas padrão do Pascal. O Turbo Pascal já a executa de forma automática.

Para fazer uso das unidades, usaremos a palavra reservada **uses**, antes da declaração **var**.

uses <unidade>

7.3 Procedimentos

A sintaxe dos procedimentos (ou procedures) em Pascal é definida como:

procedure [nome] ([parâmetros])
var

[variáveis]

begin

[instruções]

end;

Agora, como usar esses procedimentos? Vamos exemplificar com um algoritmo:

Algoritmo:

Esse programa é uma calculadora de 4 operações, e deverá conter 5 rotinas, a saber: Principal, Adição, Subtração, Multiplicação e Divisão. A rotina Principal fará o controle sobre as outras 4 rotinas, que pedirão 2 valores e farão a operação, mostrando o resultado. Para deixar claro como vai estar estruturado o programa, escreveremos cada um dos algoritmos em separado.

A. Programa principal

1. Apresentar um menu de seleção com 5 opções: Adição, Subtração, Multiplicação, Divisão, Fim do programa
2. Ao ser selecionada uma opção, a rotina correspondente será executada.
3. Se for escolhido o valor 5, o programa será encerrado.

B. Adição

1. Ler 2 valores e colocar nas variáveis A e B;

2. Efetuar a soma de A e B e colocar o resultado em X;
3. Apresentar o valor de X;
4. Voltar ao programa principal.

C. Subtração

1. Ler 2 valores e colocar nas variáveis A e B;
2. Efetuar a subtração de A e B e colocar o resultado em X;
3. Apresentar o valor de X;
4. Voltar ao programa principal.

D. Multiplicação

1. Ler 2 valores e colocar nas variáveis A e B;
2. Efetuar a multiplicação de A e B e colocar o resultado em X;
3. Apresentar o valor de X;
4. Voltar ao programa principal.

E. Divisão

1. Ler 2 valores e colocar nas variáveis A e B;
2. Efetuar a divisão de A e B e colocar o resultado em X;
3. Apresentar o valor de X;
4. Voltar ao programa principal.

7.4 Variáveis Globais e Locais

No programa anterior, definimos algumas variáveis, como OPCAO, X, A e B. Note que a primeira foi criada para uso do programa principal, e as subseqüentes, como variáveis de cada um dos procedimentos. Logo, podemos afirmar que OPCAO é uma variável *global*, e X, A e B, variáveis *locais*.

Uma variável é considerada global quando é declarada no início de um programa. Essa variável é visível para todo o programa, incluindo os procedimentos. Logo, qualquer subrotina pode modificar o valor dessa variável.

Já uma variável é considerada local quando é declarada dentro de uma subrotina, e ela é visível somente naquele procedimento onde ela foi criada. Você pode ter inclusive variáveis diferentes com nomes iguais, mas todas sendo locais. Ou mesmo uma variável local e uma global, com o mesmo nome, e mesmo assim, não haver confusão.

Exemplo:

Nesse caso, há um fato interessante: A variável X só existirá quando a sub-rotina TROCA for executada. Quando a execução estiver no programa principal, a variável X será desconsiderada.

Algorithm 30 Programa-exemplo do uso de procedimentos.

```
program CALCULADORA;
var
 OPCAO: char;

{Subrotinas de calculos do programa}

  procedure ADICAO;
  var X,A,B: real;
  begin
    writeln(´Adição:´);
    write(´Valor de A:´); readln(A);
    write(´Valor de B:´); readln(B);
    X:=A+B;
    writeln(´Resultado:´,X:4:2);
    writeln(´Tecle algo.´);
    readln;
  end;

  procedure SUBTRACAO;
  var X,A,B: real;
  begin
    writeln(´Subtração:´);
    write(´Valor de A:´);  readln(A);
    write(´Valor de B:´);  readln(B);
    X:=A-B;
    writeln(´Resultado:´,X:4:2);
    writeln(´Tecle algo.´);
    readln;
  end;

  procedure MULTIPLICACAO;
  var X,A,B: real;
  begin
    writeln(´Multiplicação:´);
    write(´Valor de A:´);  readln(A);
    write(´Valor de B:´);  readln(B);
    X:=A*B;
    writeln(´Resultado:´,X:4:2);
    writeln(´Tecle algo.´);
    readln;
  end;

  procedure DIVISAO;
  var X,A,B: real;
  begin
    writeln(´Divisão:´);
    write(´Valor de A:´);  readln(A);
    write(´Valor de B:´);  readln(B);
    X:=A/B;
    writeln(´Resultado:´,X:4:2);
    writeln(´Tecle algo.´);
    readln;
  end;
```

Algorithm 31 Continuação do programa CALCULADORA

```
{Programa principal}

begin
  writeln(´Calculadora:´);
  OPCAO:=´0´;
  while (OPCAO<> ´5´) do
    writeln(´Menu principal:´);
    writeln(´1-Adição´);
    writeln(´2-Subtração´);
    writeln(´3-Multiplicação´);
    writeln(´4-Divisão´);
    writeln(´5-Fim do programa´);
    writeln(´Escolha:´);
    read(OPCAO);
    if OPCAO=´1´ then ADICAO;
    if OPCAO=´2´ then SUBTRACAO;
    if OPCAO=´3´ then MULTIPLICACAO;
    if OPCAO=´4´ then DIVISAO;
  end;
{Note no programa que poderíamos ter substituído os 5 comandos if...then
por uma estrutura case...of, como foi visto na seção 4.4. Modificando
o programa principal, teríamos algo como podemos ver no programa se-
guinte.}
end.
```

Algorithm 32 Programa principal fazendo uso da estrutura **case...of**.

```
{Programa principal}
begin
  writeln(´Calculadora:´);
  OPCAO:=´0´;
  while (OPCAO<> ´5´) do
    writeln(´Menu principal:´);
    writeln(´1-Adição´);
    writeln(´2-Subtração´);
    writeln(´3-Multiplicação´);
    writeln(´4-Divisão´);
    writeln(´5-Fim do programa´);
    writeln(´Escolha:´);
    read(OPCAO);
    if OPCAO<>´5´ then
      case OPCAO of
        ´1´: ADICAO;
        ´2´: SUBTRACAO;
        ´3´: MULTIPLICACAO;
        ´4´: DIVISAO;
      else
        writeln(´Opção inválida´);
      end;
  end;
end.
```

Algorithm 33 Programa que exemplifica o uso de variáveis locais e globais.

```
program TROCA_VALORES;
var
  A,B: integer;
procedure TROCA;
var
  X: integer;
begin
  X:=A;
  A:=B;
  B:=A;
end;
begin
  readln(A);
  readln(B);
  TROCA;
  writeln(A);
  writeln(B);
end.
```

7.5 Uso de parâmetros

Os parâmetros servem para comunicação entre uma sub-rotina e o programa principal, ou entre sub-rotinas. Desse jeito podemos passar valores de uma sub-rotina par ao programa principal e vice-versa, usando parâmetros *formais* ou *reais*.

7.5.1 Parâmetros formais e reais

Os parâmetros são tidos como formais quando são declarados através de variáveis juntamente com a identificação do nome da sub-rotina, sendo tratados do mesmo jeito que são as variáveis locais ou globais. Os parâmetros são reais quando estes substituem os parâmetros formais, usando uma sub-rotina através de um programa ou outra sub-rotina. Abaixo temos um exemplo:

7.5.2 Passagem de parâmetros

A passagem de parâmetro ocorre quando substituímos os parâmetros formais pelos reais no momento de execução da sub-rotina. Essa passagem pode ser feita tanto por valor como por referência.

7.5.2.1 Passagem por valor

Quando ocorre a passagem de parâmetro por valor, o valor do parâmetro real não é alterado quando o parâmetro formal é manipulado dentro da sub-rotina. Nesse modo, o parâmetro formal recebe o valor passado pelo parâmetro real e manipula-o dentro da sub-rotina, funcionando como se fosse uma variável local. Logo, alterando a variável local da sub-rotina, o parâmetro real correspondente não vai ser alterado.

Algorithm 34 Programa fazendo uso de parâmetros formais e reais.

```
program CALC;
var
  X,Y,W,T: integer;
procedure ADICAO(A,B: integer);<———Parâmetros formais
var
  Z: integer;
begin
  Z:=A+B;
  writeln(Z);
end;
begin
  readln(X);
  readln(Y);
  ADICAO(X,Y);<————————Parâmetros reais
  readln(W);
  readln(T);
  ADICAO(W,T);<————————-Parâmetros reais
  ADICAO(6,4);<————————Parâmetros reais
  writeln(B);
end.
```

Algorithm 35 Programa que faz uso de passagem de parâmetros por valor.

```
program FATORIAL1;
var
  LIMITE: integer;
procedure FATORIAL(N: integer);
var
  I, FAT: integer;
begin
  FAT:=1;
  for I:=1 to N do
    FAT:=FAT*I;
  writeln(´Fatorial de ´,N:2,´ equivale a´,FAT:4);
end;
begin
  writeln(´Fatorial:´);
  write(´Informe um valor inteiro:´);
  readln(LIMITE);
  FATORIAL(LIMITE);
  writeln(´Tecle <ENTER>´);
  readln;
end.
```

Algorithm 36 Programa que faz uso de passagem de parâmetros por referência.

```
program FATORIAL2;
var
  LIMITE, RETORNO: integer;
procedure FATORIAL(N: integer;var FAT: integer);
var
  I, FAT: integer;
begin
  FAT:=1;
  for I:=1 to N do
    FAT:=FAT*I;
end;
begin
  writeln(´Fatorial:´);
  write(´Informe um valor inteiro:´);
  readln(LIMITE);
  FATORIAL(LIMITE,RETORNO);
  writeln(´Fatorial de ´,LIMITE:2,´ equivale a´,RETORNO:4);
  writeln(´Tecle <ENTER>´);
  readln;
end.
```

7.5.2.2 Passagem por referência

Quando ocorre a passagem de parâmetro por referência, o valor do parâmetro real é alterado quando o parâmetro formal é manipulado dentro da sub-rotina. O parâmetro formal recebe o valor passado pelo parâmetro real e manipula-o dentro da sub-rotina, mas depois retornando o valor para o parâmetro real.

Abaixo temos um exemplo de como usar apropriadamente sub-rotinas e passagens de parâmetros por valor e por referência.

7.6 Funções

As funções (definidas pela palavra reservada **function**) funcionam de modo análogo aos procedimentos. As funções são idênticas aos procedimentos, logo valendo todas as regras de passagem de parâmetros, estruturação do código, variáveis globais e locais, etc. Mas ambas são diferentes em apenas uma coisa: a função sempre retorna um valor. O valor de uma função é retornado no próprio nome da função. Quando é dito valor, devem ser levados em consideração valores numéricos, lógicos ou literais, como caracteres. A sintaxe das funções (ou functions) em Pascal é definida como:

function [nome] ([parâmetros]):<tipo>;
var

[variáveis]

begin

[instruções]

end;

Algorithm 37 Exemplo de programa com sub-rotina e passagem de parâmetros por referência.

```
program ORDENA_NUMEROS;
var
  VETOR: array[1..10] OF integer;
  TECLA: char;
  I,J: integer;
{Sub-rotina de ordenação}
procedure ORDENA(var A, B: integer);
var
  X: integer;
begin
  if (A>B) then
  begin
    X:=A;
    A:=B;
    B:=X;
  end;
end;
{Programa principal}
begin
  writeln(´Ordenação:´);
{Entrada dos dados}
  for I:=1 to 10 do
    begin
      write(´Digite o ´,I:2,´ numero: ´);
      readln(VETOR[I]);
    end;
{Ordenacao dos dados}
  for I:=1 to 9 do
    for J:=I+1 to 10 do
      ordena(VETOR[I],VETOR[J]);
{Saida dos dados}
  for I:=1 to 10 do
    writeln(´Numero : ´,I:2,´ —> ´,VETOR[I]);
  writeln;
end.
```

Algorithm 38 Exemplo de uma função (**function**) e o seu uso.

```
program FATORIAL_FUNCAO;
var
  N: integer;
{Função fatorial}
function FATORIAL (N: integer):integer;
var
  I,FAT: integer;
begin
  FAT:=1;
  for I:=1 to N do
    FAT:=FAT*I;
  FATORIAL:=FAT;
end;
{Programa principal}
begin
  writeln(´Entre com um numero:´);
  read(N);
  writeln(´O fatorial de ´,N:2,´ é ´,FATORIAL(N):4);
end;
```

Exemplo: Abaixo temos a função FATORIAL, que calcula o fatorial de um número n, $n!$. Fazemos a passagem de parâmetros por valor, no caso N, e o retorno é dado pela própria sub-rotina. No cao de um procedimento, a passagem de parâmetros teria que ser por referência, e o resultado seria repassado para uma variável.

Ao lidar com funções, podemos fazer a passagem de parâmetros por referência também, mas depende da situação em que ela torna-se necessária. Uma função pode receber diversos valores ao mesmo tempo, e lidar com eles. O programa da calculadora, que vimos anteriormente, pode ser reescrito dessa forma:

7.7 Recursividade

Diz-se que uma função (**function**) ou um procedimento (**procedure**) é recursiva, quando ela chama e executa a si mesma. Mas como uma rotina pode chamar a ela mesma? Este conceito é difícil de entender, é estranho ou até mesmo desnecessária devido ao nível de programas o qual estamos trabalhando. Mas o uso da recursividade muitas vezes é a única forma de resolver problemas complexos. No nível que será dado este curso, bastará saber o conceito e o funcionamento de uma subrotina recursiva. Usaremos o clássico exemplo da função fatorial.

No exemplo, temos 2 versões do programa. A versão não-recursiva já foi apresentada anteriormente: a multiplicação repetidas vezes por valores inteiros sucessivos até n. A versão recursiva funciona multiplicando n repetidas vezes pelo fatorial do número imediatamente anterior. A versão recursiva é mais elegante do que a não-recursiva, mas apresenta desvantagens. A principal é que a versão recursiva gasta muito mais memória do que a não-recursiva: O compilador aloca espaço temporário para armazenamento da rotina. Já a versão não-recursiva normalmente é tida como mais fácil de ser entendida por muitos programadores.

O que usar, afinal? Depende do caso. Existem algoritmos em que a

Algorithm 39 Programa-exemplo do uso de funções e procedimentos.

```
program CALCULADORA_2;
var
 OPCAO: char;
 X, A, B: real;

  procedure ENTRADA;
  begin
    write(´Valor de A:´);
    readln(A);
    write(´Valor de B:´);
    readln(B);
  end;
  procedure SAIDA;
  begin
    write(´O resultado é ´,X:4:2);
    write(´Tecle algo´);
    readln;
  end;

{Subrotinas de calculos do programa}

  function CALCULO (R, T: real; OP: char): real;
  begin
    case OP of
     ´+´:CALCULO:=R+T;
     ´-´:CALCULO:=R-T;
     ´*´:CALCULO:=R*T;
     ´/´:CALCULO:=R/T
    end;
  end;
  procedure Adicao;
  begin
    ENTRADA;
    CALCULO(A,B,´+´);
    SAIDA;
  end;
  procedure Subtracao;
  begin
    ENTRADA;
    CALCULO(A,B,´-´);
    SAIDA;
  end;
  procedure Multiplicacao;
  begin
    ENTRADA;
    CALCULO(A,B,´*´);
    SAIDA;
  end;
  procedure Divisao;
  begin
    ENTRADA;
    CALCULO(A,B,´/´);
    SAIDA;
  end;
```

Algorithm 40 Continuação do programa CALCULADORA_2.

```
{Programa principal}

begin
  writeln(´Calculadora:´);
  OPCAO:=´0´;
  while (OPCAO<> ´5´) do
    writeln(´Menu principal:´);
    writeln(´1-Adição´);
    writeln(´2-Subtração´);
    writeln(´3-Multiplicação´);
    writeln(´4-Divisão´);
    writeln(´5-Fim do programa´);
    readln(OPCAO);
    if OPCAO<>´5´ then
      case OPCAO of
        ´1´: Adicao;
        ´2´: Subtracao;
        ´3´: Multiplicacao;
        ´4´: Divisao;
      else
        writeln(´Opção inválida´);
      end;
  end;
end.
```

recursividade "cai como uma luva", logo adaptá-lo para uma forma não-recursiva não faz sentido. Varia de acordo com o gosto e opção do programador.

7.8 Exercícios:

Faça um programa que:

1. Contenha uma subrotina que faça a soma dos N primeiros números naturais, definidos por um operador.
2. Contenha uma subrotina capaz de calcular a série de Fibonacci de N termos.
3. Contenha uma função capaz de calcular a potência de um número qualquer. Por exemplo, POTENCIA(2,3) gera como resultado o valor 8. Desenvolva essa mesma rotina como um procedimento, usando passagem de parâmetro por referência.
4. Contenha uma subrotina para ler uma matriz mxn do tipo inteiro. Os valores m e n deverão ser lidos. Depois, outra subrotina que ordene os valores da matriz, em cada linha.
5. Contenha uma subrotina para ler um vetor A de n elementos, e um vetor B de m elementos. Os valores m e n deverão ser lidos. Coloque depois uma função que faça a soma dos valores, tomando cuidado com os tamanhos de m e n.

Algorithm 41 Exemplo de sub-rotinas não-recursiva e recursiva.

```
function FATORIAL (N: integer): real;
var
  R: real;
  I: integer;
begin
  R:=1;
  for I:=2 to N do
    R:=R*I;
  FATORIAL:=R;
end;

function FATORIAL (N: integer): real;
begin
  if N=0 then
    FATORIAL:=1
  else
    FATORIAL:=N*FATORIAL(N-1);
end;
```

6. Contenha um procedimento que informe se uma frase é palíndroma (ou seja, pode ser escrita de trás para frente, sem mudanças).

7. Faça um programa que leia um vetor de números inteiros. Após, emita um relatório com cada número diferente, e o número de vezes que o mesmo apareceu repetido no vetor.

8. Faça um programa para ler um vetor A com m elementos e um vetor B com n elementos (os valores m e n podem ou não serem iguais). Formar um terceiro vetor, C com os elementos dos vetores A e B intercalados. Exemplo: C[1] : = A[1]; C[2] : = B[1]; C[3] : = A[2]; C[4] : = B[2];

9. Faça um programa com uma função que calcule o valor de uma prestação em atraso. A fórmula é: $PREST = VALOR + (VALOR * (TAXA/100) * TEMPO)$.

Capítulo 8

Variáveis compostas heterogêneas

Anteriormente, quando foi tratado sobre vetores e matrizes (capítulo 6), pudemos ver que apenas um tipo de dado por vetor ou matriz pode ser trabalhado. Se for necessário lidar com dois tipos de dados diferentes, é preciso construir dois vetores, sendo uma de cada tipo.

8.1 Tipo de dado registro

Existe um tipo de dado em Pascal, chamado registro (**record**), onde é possível colocar vários tipos de dados diferentes (os campos) dentro de uma mesma estrutura. Por isso afirmamos que o registro é uma variável composta heterogênea.

Na estrutura da linguagem Pascal, o tipo registro deve ser declarado antes das definições das variáveis, pois assim depois podemos criar variáveis com o novo tipo criado.

Sintaxe:

type
  <identificador> = record
    [lista dos campos e seus tipos]
end;
var
  <variável>: <identificador>;

Onde:

<identificador>: O nome do tipo registro.

<variável>: A variável cujo tipo será registro.

8.2 Exemplos

Por exemplo, queremos montar um registro para armazenar dados de um aluno numa escola. O registro deverá conter nome do aluno, matrícula e as médias dos 4 bimestres. Logo, pela definição acima, o registro será feito do seguinte modo:

```
type
  CAD_ALUNO = record
```

Algorithm 42 Exemplo de uso do tipo de dado registro.

```
program EXEMPLO_DE_USO_DO_REGISTRO;
type
  cad_aluno = record
    nome: string[40];
    matricula: integer;
    nota1: real;
    nota2: real;
    nota3: real;
    nota4: real;
  end;
var
 ALUNO: array[1..50] of cad_aluno;
 RESP: char;
 I,J: integer;

begin
 RESP:= ´S´;
 I:=1;
 writeln(´Cadastro de alunos:´);
 while (RESP=´S´) or (RESP=´s´) do
 begin
  write(´Aluno ´,I:2,´:´);
  write(´Nome:´);   readln(ALUNO[I].nome);
  write(´Matricula:´);   readln(ALUNO[I].matricula);
  write(´1o. bimestre:´);   readln(ALUNO[I].nota1);
  write(´2o. bimestre:´);   readln(ALUNO[I].nota2);
  write(´3o. bimestre:´);   readln(ALUNO[I].nota3);
  write(´4o. bimestre:´);   readln(ALUNO[I].nota4);
  writeln(´Continua(S/N)?´);
  read(RESP);
  I:=I+1;
 end;
 for J:=1 to I do
 begin
  writeln(´Nome:´,ALUNO[J].nome);
  writeln(´Matricula:´,ALUNO[J].matricula:5);
  writeln(´1o. bimestre:´,ALUNO[J].nota1:2:2);
  writeln(´2o. bimestre:´,ALUNO[J].nota2:2:2);
  writeln(´3o. bimestre:´,ALUNO[J].nota3:2:2);
  writeln(´4o. bimestre:´,ALUNO[J].nota4:2:2);
 end;
end.
```

```
      nome: string[40];
      matricula: integer;
      nota1: real;
      nota2: real;
      nota3: real;
      nota4: real;
    end;
  var
    aluno: cad_aluno;
```

O registro está sendo denominado CAD_ALUNO, que é um conjunto de dados heterogêneos: Um campo do tipo string, com 40 caracteres, um campo do tipo inteiro e quatro campos do tipo real. Tanto a leitura quanto a escrita de um registro são feitas com os comando **read/readln** e **write/writeln**, seguidas do nome da variável do tipo registro e de seu campo correspondente separado por um ponto (.). No exemplo, temos um programa que demonstra como podemos manipular os dados que estão dentro de um registro.

Definimos acima apenas um registro, aluno. No primeiro exemplo, temos um vetor de registros, com 50 posições, onde cada uma das posições do vetor pode ser a ficha de um aluno e todo o vetor, o resultado de uma turma.

Poderíamos também alterar o registro e colocar dentro dele um vetor. Assim poderíamos construir um vetor de 4 posições, do tipo real, ao invés de termos 4 variáveis reais. Teríamos então uma definição do seguinte modo:

```
  type
    BIMESTRE = array[1..4] of real;
    CAD_ALUNO = record
              nome: string[40];
              matricula: integer;
              nota: bimestre;
    end;
  var
    aluno: array[1..100] of cad_aluno;
```

O exemplo ficaria diferente, poderíamos então inserir um vetor dentro do registro. Como podemos ver, o registro nos dá uma mobilidade muito grande, podendo tratar certos problemas de uma forma mais adequada, principalmente quando usamos dados heterogêneos. Podemos fazer muitas modificações, como incluir por exemplo uma rotina de ordenação para o vetor, que ficaria assim:

```
  if (aluno[I].nome)>aluno[I+1].nome) then
  begin
    X:=aluno[I];
    aluno[I]:=aluno[I+1];
    aluno[I+1]:=X;
  end;
```

Algorithm 43 Outro exemplo de uso do tipo de dado registro.

```
program EXEMPLO_2_DE_USO_DO_REGISTRO;
type
   BIMESTRE = array[1..4] of real;
   CAD_ALUNO = record
               nome: string[40];
               matricula: integer;
               nota: bimestre;
   end;
var
  ALUNO: array[1..100] of cad_aluno;
  RESP: char;
  I,J,K: integer;

begin
  RESP:= ´S´;
  I:=1;
  writeln(´Cadastro de alunos:´);
  while (RESP=´S´) or (RESP=´s´) do
  begin
    write(´Aluno ´,I:2,´:´);
    write(´Nome:´);   readln(ALUNO[I].nome);
    write(´Matricula:´);   readln(ALUNO[I].matricula);
    for K:=1 to 4 do
      write(K,´o. bimestre:´);   readln(ALUNO[I].nota[K]);
    writeln(´Continua(S/N)?´);
    read(RESP);
    I:=I+1;
  end;
  for J:=1 to I do
  begin
    writeln(´Nome:´,ALUNO[J].nome);
    writeln(´Matricula:´,ALUNO[J].matricula:5);
    for K:=1 to 4 do
      writeln(K,´o. bimestre:´,ALUNO[J].nota[K]:2:2);
  end;
end.
```

8.3 Exercício

Considerando o registro criado acima para controle dos alunos, construa um programa que através de um menu de seleção, seja capaz de:

- Cadastrar cada aluno e seus dados;
- Pesquisar registros - usando pesquisa binária;
- Fazer alteração de registro cadastrado com erro - usando pesquisa seqüencial;
- Remover um determinado registro - usando pesquisa seqüencial.

Cada uma das funções que exigem pesquisa, a mesma deverá ser realizada por nome. Não esqueça de acrescentar uma opção no menu para sair do programa.

Capítulo 9

Arquivos

Em capítulos anteriores, vimos como manipular tabelas em memória, através da utilização de matrizes. Depois, vimos como usar uma estrutura de dados heterogênea, o registro. O resultado desses capítulos serve como base para a utilização do que conhecemos como **arquivo**. O arquivo é uma tabela de informações gravada em um meio físico, como disquete, HD, fita magnética, CD-ROM, etc. O arquivo pode ser salvo e recuperado posteriormente para ser manipulado. Outra das vantagens do arquivo é que podemos armazenar mais registros no arquivo do que a memória comporta. O limite é o tamanho do meio físico usado para a gravação, que quase sempre é maior do que a memória RAM disponível.

9.1 Definição

A sintaxe do arquivo é dada por:

var

<variável>:[text][file [of <tipo>]];

Onde:

<variável>: A variável que será usada para representar o arquivo.

<tipo>: O tipo de dado que será colocado dentro do arquivo.

No Pascal, é possível trabalhar com três tipos diferentes de arquivo: Tipo texto (**[text]**), tipo definido (**[file of <tipo>]**) e sem tipo definido.

9.2 Manipulação de arquivos

Independentemente dos tipos de arquivos, algumas operações são comuns, como abertura, leitura, escrita, fechamento de um arquivo. Abaixo temos uma lista com os procedimentos usados para manipulação de arquivos:

assign *assign(<variável>,<arquivo>);* Associa um nome lógico ao arquivo.

rewrite *rewrite(<variável>);* Cria um arquivo novo para uso.

reset *reset(<variável>);* Abre um arquivo existente e coloca o ponteiro no início do arquivo.

append *append(<variável>);* Abre um arquivo existente e coloca o ponteiro no final do arquivo.

write *write(<variável>,<dado>);* Escreve <dado> no arquivo.

read *read(<variável>,<dado>);* Lê <dado> no arquivo.

close *close(<variável>);* Fecha o arquivo.

seek *seek(<variável>,**n**);* Move o ponteiro do arquivo para o registro de número **n**.

flush *flush(<variável>);* Salva no arquivo tudo que está na memória.

erase *erase(<variável>);* Apaga o arquivo associado com <variável>.

rename *rename(<variável>,<novo nome>);* Renomeia o arquivo associado com <variável>, para <novo nome>.

As funções abaixo também são usadas para manipulação de arquivos:

eof eof(<variável>):booleano - Função booleana. Testa se o ponteiro do arquivo está posicionado no fim do arquivo.

filepos filepos(<variável>): inteiro - Função inteira. Retorna a posição corrente do ponteiro do arquivo.

filesize filesize(<variável>): inteiro - Função inteira. Retorna o tamanho do arquivo expresso no número de componentes.

9.3 Formas de acesso ao arquivo

Os arquivos, ao serem criados em meios magnéticos, poderão ser acessados para leitura e/ou escrita na forma seqüencial, direta ou indexada.

Seqüencial: O acesso ocorre de forma contínua, um após o outro. Logo, para se gravar um novo registro, por exemplo, é necessário percorrer todo o arquivo a partir do primeiro registro e gravar na primeira posição vazia após o último registro.

Direto: Também conhecido como acesso randômico. O acesso ocorre através do que é conhecido como campo-chave. O campo-chave é utilizado para localizarmos o registro diretamente, sem precisar vasculhar os registros anteriores. Desse modo, ler ou gravar um registro dentro de um arquivo é uma operação instantânea.

Indexado: Ocorre quando acessamos de forma direta um arquivo seqüencial. Criamos um arquivo direto que serve como índice de consulta (daí o nome indexado) do arquivo seqüencial. Manipulando os 2 arquivos, podemos encontrar o registro desejado rapidamente.

Algorithm 44 Programa usado para manipular arquivos do tipo texto.

```
program MANIPULA_ARQUIVO_TEXTO;
var
  ARQ_TXT: text;
  MSG: string[80];

procedure GRAVA_ARQUIVO_TEXTO;
var
  RESP: char;
begin
  RESP:= ´S´;
  assign(ARQ_TXT,´ARQTESTE.TXT´);
  append(ARQ_TXT);
  while (RESP=´S´) or (RESP=´s´) do
  begin
    writeln(´Frase:´);
    readln(MSG);
    writeln(ARQ_TXT,MSG);
    writeln(´Continua(S/N)?´);
    read(RESP);
  end;
  close(ARQ_TXT);
end;

procedure LE_ARQUIVO_TEXTO;
begin
  assign(ARQ_TXT,´ARQTESTE.TXT´);
  reset(ARQ_TXT);
  while not eof (ARQ_TXT) do
  begin
    readln(ARQ_TXT,MSG);
    writeln(MSG);
  end;
  close(ARQ_TXT);
end;

begin
  GRAVA_ARQUIVO_TEXTO;
  LE_ARQUIVO_TEXTO;
end.
```

9.4 Arquivos do tipo texto

A sintaxe do arquivo do tipo texto é:

var

<variável>:[text];

O tipo de arquivo mais simples é o tipo texto. Este tipo de arquivo pode ter registros de tamanho diferentes, sendo que cada registro é finalizado com uma sequência CR/LF (Carriage Return / Line Feed). Esse tipo de arquivo é o mais simples, e utilizado para arquivos-texto (extensão TXT no DOS e Windows).

Comecemos pelo procedimento GRAVA_ARQUIVO_TEXTO. Esse procedimento gravará dados dentro do arquivo. O comando **assign** associa um nome de arquivo (no caso ARQTESTE.TXT) a uma variável (ARQ_TXT). O comando **rewrite** cria o arquivo ARQTESTE.TXT, e posiciona o ponteiro de controle de registros no início do arquivo. Se o arquivo já existia anteriormente, ele será apagado. Se usarmos o comando **append**, o arquivo é aberto e o ponteiro posicionado no final do arquivo.

Estamos usando um laço no procedimento GRAVA_ARQUIVO_TEXTO para fazer a entrada dos dados, assim enquanto o usuário quiser, ocorrerá a entrada de dados. O comando **writeln**, nesse caso, coloca o dado que está na variável MSG dentro do arquivo representado pela variável ARQ_TXT. Nesse caso, não há saída do comando **writeln** para a tela, mas para o arquivo. **Depois**, com o comando **close**, o arquivo é fechado e conseqüentemente, salvo em disco.

No procedimento seguinte, LE_ARQUIVO_TEXTO, destaquemos o comando **reset**, que apenas abre o arquivo. Nesse caso, o ponteiro do arquivo é posicionado no início do arquivo. Também temos o comando **readln**, que lê o dado contido na variável MSG dentro do arquivo representado pela variável ARQ_TXT. Usamos a função **eof**, que testa o fim de um arquivo: enquanto o arquivo não chega ao fim, o laço repete-se, imprimindo na tela os dados do arquivo.

Podemos criar uma função booleana que testa se o arquivo existe. Se o arquivo existe, a função retorna TRUE. Se não, retorna FALSE. Essa função faz uso de uma variável interna do Turbo Pascal, **IOresult**, e de uma diretiva de compilação, **{$I+/-}**.

A diretiva de compilação corrente liga (+) ou desliga (-) a checagem de entrada e saída. Nesse caso, é interessante desligarmos para fazer o teste da existência ou não de um arquivo. Assim, mesmo que ocorra um erro, a execução do programa não ocorrerá. A variável retorna um número, onde cada um aponta o resultado da operação:

IOResult:

0 Normal.

2 Arquivo não-encontrado.

3 Caminho não-encontrado.

4 Excesso de arquivos abertos.

5 Acesso negado ao arquivo.

6 Tratamento de arquivo inválido.

Algorithm 45 Função booleana usada para testar se um arquivo existe.

```
function EXISTE(NomeDoArquivo: string[12]): boolean;
var
  ARQ: file;
begin
  assign(ARQ,NomeDoArquivo);
  {$I-}
  reset(ARQ);
  {$I+}
  Exist:=(IOresult=0);
end.
```

12 Código de acesso ao arquivo inválido.

15 Número de unidade de disco inválido.

16 Impossível remover o diretório atual.

17 Impossível mudar nome através das unidades.

9.5 Arquivos com tipo definido

Os arquivos com tipo definido também são conhecidos como *arquivos tipados*, e permitem armazenar dados de um determinado tipo, como inteiro, real, registro, caracter, etc. A manipulação dos arquivos tipados é mais rápida do que os arquivos do tipo texto.

Sintaxe:

var

<variável>:[file of [tipo]];

O exemplo desta seção é idêntico ao exemplo da seção 9.4, e vale a pena ressaltar a instrução **seek(ARQ_INT,filesize(ARQ_INT)**. O comando **seek** coloca o ponteiro de registro numa determinada posição do arquivo, e **filesize** retorna o tamanho em registros do arquivo. Logo, usando ambos, apontaremos o ponteiro de registro para o último registro do arquivo.

Os registros, nesse caso, possuem o tamanho fixo de 128 bytes. Se o arquivo tiver mais do que 32 Kb (32768 bytes, ou 256 registros), os comandos seek e filesize não funcionarão, devendo então ser usados os comandos **longseek** e **longfilesize**, que tem a mesma sintaxe.

9.6 Exercícios

Criar um programa que:

1. Leia 20 valores numéricos inteiros num vetor e os salve em um arquivo.
2. Leia o arquivo do exercício anterior, e some todos os números ímpares.
3. Leia os dados de um vetor A, com 10 valores numéricos, crie um vetor B, onde $B = A^2$, e salve o vetor B em um arquivo.

Algorithm 46 Exemplo de programa para manipular dados em arquivos tipados.

```
program MANIPULA_ARQUIVO_INTEIRO;
var
  ARQ_INT: file of integer;
  NO: integer;

procedure GRAVA_ARQUIVO_INTEIRO;
var
  RESP: char;
begin
  RESP:= ´S´;
  assign(ARQ_INT,´ARQTESTE.INT´);
  append(ARQ_INT);
  while (RESP=´S´) or (RESP=´s´) do
  begin
    seek(ARQ_INT, filesize(ARQ_INT));
    writeln(´Numero:´);
    readln(NO);
    write(ARQ_INT,NO);
    writeln(´Continua(S/N)?´);
    read(RESP);
  end;
  close(ARQ_INT);
end;

procedure LE_ARQUIVO_INTEIRO;
begin
  assign(ARQ_INT,´ARQTESTE.INT´);
  reset(ARQ_INT);
  while not eof (ARQ_INT) do
  begin
    read(ARQ_INT,NO);
    writeln(NO);
  end;
  close(ARQ_INT);
end;

begin
  GRAVA_ARQUIVO_INTEIRO;
  LE_ARQUIVO_INTEIRO;
end.
```

4. Leia os dados de duas matrizes quadradas, A e B, de 5x5 elementos, some-os, colocando o resultado na matriz C, e a salve em um arquivo.

5. Apresente quatro opções: Criar arquivo, cadastrar palavras, ler palavras e sair. A rotina de cadastramento deve ser capaz de ler uma palavra de cada vez, e somente parar quando for digitada a palavra **fim**. Todas as palavras digitadas devem ser salvar em um arquivo do tipo texto. A rotina de exibição deve ser capaz de exibir todas as palavras salvas no arquivo.

Referências Bibliográficas

[1] ALGORITMOS - LÓGICA PARA DESENVOLVIMENTO DE PROGRAMAÇÃO - José Augusto N. G. Manzano e Jayr Figueiredo de Oliveira - Editora Érica, 1996

[2] PASCAL ISO - MANUAL DO USUÁRIO E RELATÓRIO - Kathleen Jensen e Niklaus Wirth - Editora Campus, 1988

[3] PROGRAMANDO EM TURBO PASCAL 7.0 - José Augusto N. G. Manzano e Wilson Y. Yamatumi - Editora Érica, 1996

[4] TURBO PASCAL VERSION 3.0 - REFERENCE MANUAL - Borland International, 1985

[5] TURBO PASCAL 6 - COMPLETO E TOTAL - Stephen O´Brien - Makron Books, 1993