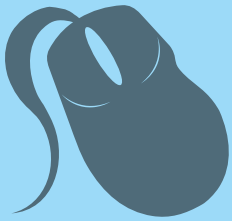


Computação

Linguagem de Programação I

Ricardo Reis Pereira


Geografia


História


Educação Física


Química


Ciências Biológicas


Artes Plásticas


Computação


Física


Matemática


Pedagogia

Informática

Linguagem de Programação I

Ricardo Reis Pereira

3ª edição
Fortaleza
2015


Geografia


História


Educação Física


Química


Ciências Biológicas


Artes Plásticas


Computação


Física


Matemática


Pedagogia

Editora Filiada à

ASSOCIAÇÃO BRASILEIRA DAS EDITORAS UNIVERSITÁRIAS

Editora da Universidade Estadual do Ceará – EdUECE
Av. Dr. Silas Munguba, 1700 – Campus do Itaperi – Reitoria – Fortaleza – Ceará
CEP: 60714-903 – Fone: (85) 3101-9893
Internet: www.uece.br – E-mail: eduece@uece.br

Secretaria de Apoio às Tecnologias Educacionais
Fone: (85) 3101-9962

Sumário

Apresentação

Sejam bem vindos ao mundo da programação de computadores! Trata-se de um mundo fascinante e sofisticado que evolui constantemente no intuito de agilizar o trabalho das pessoas envolvidas com tecnologia da informação. Não existe apenas uma forma de programar um computador. Na realidade cada área de conhecimento abrangida pela computação está ligada a alguma categoria de programação que se caracterizam por suas próprias ferramentas e linguagens.

Assim, por exemplo, um programador de sistemas operacionais trabalha no desenvolvimento de ferramentas que permitam melhor acessibilidade do hardware, um programador de jogos almeja usar ao máximo o potencial gráfico disponível para desenvolver aplicativos para entretenimento, um programador de sistemas informativos cria softwares que permitam às pessoas interagirem através de uma rede particular ou da internet, um programador de interfaces objetiva maximizar o aproveitamento de aplicações locais ou remotas utilizando todo aparato visual moderno e assim por diante. Cada uma destas áreas requer especialização.

Entretanto em todos os casos os programadores precisam de conhecimentos essenciais adquiridos em disciplinas básicas sobre programação de computadores. Estes conhecimentos passam pela estudo das chamadas linguagens de programação que, de forma geral, consistem de poderosos artifícios para a expressão de instruções a serem executadas por computadores.

Este livro tem por objetivo apresentar e explorar conceitos essenciais da linguagem de programação Pascal. Esta linguagem foi desenvolvida para ser fácil de compreender e utilizar. De fato ela foi criada para ser ensinada e por essa razão tem boa aceitação entre iniciantes em programação. Além do mais uma vez compreendida a linguagem Pascal torna-se consideravelmente mais simples tanto migrar para outras linguagens como ingressar numa área de especialização como aquelas citadas ante- riormente.

A estrutura básica do livro consiste de seis capítulos divididos em três unidades. A primeira unidade trata de toda parte fundamental da linguagem de forma que é possível ao final dela já construir aplicações que funcionem. As duas unidades finais são adendos para sofisticação e potencialização do que foi aprendido na primeira unidade. Em termos gerais o objetivo do livro é permitir que o leitor programe o mais prematu- ramente possível de forma prática, desafiadora e estimulante.

O autor

Parte 1

Introdução à Programação de Computadores

Capítulo 1

Fundamentos

Objetivos:

- Esta unidade tem por objetivo introduzir o estudante no aprendizado da linguagem de programação Pascal. No Capítulo 1 são apresentadas a sintaxe básica bem como as estruturas fundamentais para implementação de pequenos programas funcionais. No Capítulo 2 a linguagem é expandida através do estudo de estruturas sequenciais que permitem a representação e manipulação de listagens de dados auxiliando o estudante a lidar com a resolução de problemas mais complexos.

1. Introdução à Programação de Computadores

No princípio não havia a distinção entre hardware e software e os primeiros computadores eram programados diretamente em seus circuitos. Consistia de uma tarefa ardua e exigia profundo conhecimento da arquitetura e do funcionamento do equipamento. Tarefas elementares exigiam considerável tempo de trabalho e erros eram frequentes. A evolução do hardware e a concepção do conceito de software deram uma guinada na história dos computadores. O software passou a representar a alma de um computador e sua inoculação nos circuitos dos equipamentos passou a ser referenciada como *instalação de software*.

Softwares são instalados em camadas, ou seja, um software, para funcionar, utiliza outros softwares. A exceção é naturalmente a camada de software instalada diretamente sobre o hardware: o sistema operacional. As demais camadas se instalam sobre este ou sobre uma ou mais camadas existentes. É considerado um *programa* um software de aplicação específica. Há programas para edição de imagens, reprodução de arquivos de som, navegação na internet, edição de texto e etc.

Um programa é um arquivo que contém uma listagem de *instruções de máquina*. Estas instruções devem ser executadas por uma CPU na ordem que se apresentam no arquivo. Este processo é denominado *execução do*

Sistema Operacional: O sistema operacional é um conjunto de softwares que trabalham cooperativamente para proporcionar ao usuário acesso a recursos do computador. Ele esconde detalhes, muitas vezes complexos, de funcionamento do hardware através de abstrações simples. Exemplos são Linux, Windows, OS2, Unix e etc

programa. Para executar um programa ele primeiramente deve ser copiado da memória secundária (do arquivo) para a primária (tempo de carregamento) onde finalmente a CPU vai buscar, uma a uma, as instruções do programa até sua finalização (tempo de execução). Muitos programas podem estar em execução ao mesmo tempo em sistemas operacionais modernos e concorrerem entre si pela mesma CPU (ou mesmas CPUs). Um programa em execução é comumente chamado *processo*.

Para um ser humano, gerar um arquivo de programa codificando instruções de máquina, é uma tarefa árdua, tediosa e dispendiosa haja vista que para se produzir programas extremamente simples o tempo necessário seria consideravelmente grande. A solução para esse problema foi a concepção de linguagens com maior nível que a linguagem de máquina necessária para expressar instruções de máquina.

Os termos *linguagem* e *nível de linguagem* são discutidos a seguir. Uma linguagem é, de forma geral, um conjunto de regras utilizados para expressar ou registrar algum tipo de informação. Uma *linguagem de programação* (de computadores) é naturalmente aquela utilizada para projetar e construir programas. Quanto mais elaborada for uma linguagem, no sentido de propiciar melhor compreensão a um ser humano, maior será o seu nível. Ao longo da história das linguagens de programação o projeto e implementação de novas linguagens com nível cada vez mais alto se mostrou uma realidade. O leitor pode encontrar em Sebesta o histórico completo das principais linguagens de programação.

A sofisticação da modernas linguagens de programação tem como principal origem a produtividade de software pelas pessoas. Para os computadores, entretanto, a melhor e mais compreensível forma ainda é a em código de máquina. Na prática um programa, escrito numa dada linguagem de programação, precisa primeiramente ser *traduzido* em linguagem de máquina para que enfim possa ser executado. Essa tradução pode ser abordada genericamente de duas formas.

Na primeira forma um código em alto nível é completamente convertido em código de máquina. Esse processo é conhecido como *compilação*. Na compilação um arquivo de código (*código-fonte*) expresso em uma linguagem de alto nível é submetido a um programa denominado *compilador* e um novo arquivo em código de máquina (*executável*) é gerado como saída. O arquivo executável é de fato um novo programa cuja execução independe do código-fonte.

A segunda estratégia de tradução é conhecida como *interpretação*. Nessa estratégia não existe a criação de um novo arquivo em código de máquina. Ao invés disso a execução é realizada diretamente sobre o código-fonte por um programa denominado *interpretador*. À proporção que o interpretador avalia as linhas de código em linguagem de alto nível, ele as converte em

código de máquina e só então as executa. Na interpretação a execução só é viável se um interpretador está instalado no sistema.

Um código de máquina, oriundo de um processo de compilação, tem dependência direta com o hardware e o sistema operacional sobre o qual foi criado. Assim se um dado código-fonte F escrito numa linguagem de alto nível X, compilado num hardware H utilizando um sistema operacional S, só executará em em máquinas com a mesma arquitetura de H e com o sistema S. Como o executável é obtido através de uma tradução em código de máquina, então a linguagem X não impõe, no contexto avaliado, restrições ao programa obtido pela tradução de F. Se o hardware H é mantido e o sistema é mudado, digamos para S_2, então F poderá ser recompilado para S_2 gerando um novo executável.

Um compilador, que traduz código escrito numa linguagem de alto nível X, precisa de uma versão para cada sistema operacional em que se deseja escrever (utilizando X) programas que executem neste sistema. Assim, um código-fonte escrito numa linguagem X, que precisa ser traduzido em código de máquina para os sistemas S_1 e S_2, precisará de duas versões de compiladores C_1 e C_2, ambos da mesma linguagem X, para os respectivos sistemas.

Similar ao que acontece com compiladores, tradutores também requerem uma versão por sistema operacional. Entretanto, como não existe compilação prévia, o código-fonte pode ser executado diretamente em sistemas distintos desde que possuam instalado o interpretador apropriado.

2. A Linguagem Pascal

A linguagem de programação Pascal foi originalmente projetada por Niklaus Wirth na década de 1970. A linguagem tem absorvido desde essa época muitas contribuições que convergiram para a criação de diversos compiladores importantes. Atualmente é possível encontrar na Internet mais de 300 compiladores ou interpretadores da linguagem (alguns pagos, outros gratuitos e muitos descontinuados, ou seja, não possuem mais suporte). Veja www.Pascaland.org para uma lista completa de compiladores Pascal. As mais notáveis contribuições para o desenvolvimento do Pascal foram feitas pela Borland (empresa americana de desenvolvimento de software) com a criação do Turbo Pascal e posteriormente do Delphi Pascal.

A linguagem Pascal é originalmente estruturada, ou seja, os programas são construídos como um conjunto de pequenos subprogramas que se conectavam por *chamadas* (uma chamada de um determinado subprograma é a solicitação de execução de outros subprogramas a partir dele). A linguagem absorveu com o passar do tempo o paradigma da *orientação a objetos* (que se baseia nos conceitos de classes e objetos) dando origem ao Object Pascal.

O Turbo Pascal e o Delphi Pascal constituem a base de desenvolvimento do compilador *Free Pascal* abordado neste livro e recomendado aos leitores (veja www.freePascal.org). Trata-se de um compilador livre, de crescente popularidade e que reconhece códigos em Pascal puramente estruturado (como acontecia no Turbo Pascal), códigos em Object Pascal (como acontece com o Delphi) ou associação de código (parte estruturada e parte orientada a objetos)

Abordaremos neste livro apenas os elementos estruturados do Free Pascal (que incluem os fundamentos da linguagem, variáveis e constantes, operadores, expressões, controle de fluxo, *strings* e *arrays*, tipos de dados personalizados, ponteiros, modularização e arquivos) que permitirão ao leitor posterior extensão ao Object Pascal (que inclui ainda uso de classes e objetos, uso de interfaces, programação genérica, sobrecarga de operadores e manipulação de exceções).

Arquitetura do Processador: Define o leiaute e a capacidade de armazenamento ou processamento dos componentes de uma unidade central de processamento (CPU). As arquiteturas mais populares hoje em dia possuem de um a dois núcleos (em inglês, *single core* e *dual core*) e tamanhos de registradores (memória interna da CPU) de 32 ou 64 bits.

3. Utilizando um Compilador Pascal

O primeiro passo para se iniciar em programação é naturalmente instalar o compilador (ou o interpretador se for o caso). Para baixar o compilador Free Pascal visite www.freePascal.org/download.var. O site dispõe várias versões do compilador listados por arquitetura do processador e por sistema operacional. Na sessão *binaries* (binários) ficam os instaladores propriamente ditos. Na sessão *sources* (fontes) o código fonte do compilador (voltado para quem deseja contribuir com melhorias no Free Pascal). Por fim a sessão *documentation* (documentação) permite baixar a documentação oficial completa do compilador em vários formatos (PDF, HTML, PS, DVI e TXT).

Janela de Terminal: Uma janela de terminal é a forma mais primitiva que um sistema operacional oferece ao usuário para requisições de tarefas. Todas as ações são repassadas como instruções escritas denominadas de *linhas de comandos*. Para indicar que novas instruções podem ser digitadas, um terminal dispõe um indicador de entrada denominado *prompt* seguido de um cursor piscante. Para confirmar entradas o usuário pressiona ENTER após cada linha de comandos e aguarda o processamento que normalmente se encerra com a reexibição do prompt aguardando novas instruções.

Para testar se o compilador instalado está devidamente configurado, abra um terminal (no Windows isso pode ser feito indo em Iniciar|Executar, digite cmd, depois OK. Em sistemas Unix e relacionados, como Linux, caso o sistema já não esteja em modo terminal, há diversos aplicativos de simulação de terminal em modo X). Quando o prompt do terminal estiver visível simplesmente digite,

```
fpc
```

Se o compilador estiver devidamente instalado será impressa uma listagem das opções básicas de uso. A seguir mostramos um fragmento que antecede a listagem propriamente dita e que trás informações do compilador.

```
Free Pascal Compiler version 2.2.4-3 [2009/06/04] for i386

Copyright (c) 1993-2008 by Florian Klaempfl

/usr/lib/fpc/2.2.4/ppc386 [options] <inputfile> [options]
```

Para compilar um programa escrito em Pascal, via linha de comando (terminal), utilizando o Free Pascal, deve-se primeiramente mudar para a pasta que contém o arquivo com o código-fonte. Em seguida digite,

```
fpc teste.pas
```

neste caso teste.pas é o arquivo de código-fonte na pasta corrente. A saída em terminal da compilação anterior é algo semelhante a,

```
Free Pascal Compiler version 2.2.4-3 [2009/06/04] for i386

Copyright (c) 1993-2008 by Florian Klaempfl

Target OS: Linux for i386

Compiling teste.pas

Linking teste

3 lines compiled, 0.1 sec
```

Quando a compilação encerra, dois novos arquivos terão surgido: teste.o e teste (ou teste.exe, no Windows). O arquivo teste.o é chamado *arquivo objeto* e representa o programa sem ligações com o sistema operacional. Na última fase da compilação todos os arquivos objetos (há projetos com mais de um arquivo de código-fonte cada qual gerando um arquivo objeto) são submetidos a um processo denominado *linkagem* (vinculação) proporcionado por um programa chamado *linker* (vinculador). Este programa resolve as pendencias de conexão dos arquivos objeto com o sistema operacional e gera como saída o programa executável, neste caso teste. Para executar o novo programa no Windows simplesmente digite na linha de comando,

```
teste.exe
```

e no Unix e Linux:

```
./teste
```

Um compilador não pode corrigir os erros que lista porque, em sua abrangente visão, há muitas possibilidades de correção de um mesmo erro. Para ajudar então no trabalho do programador o compilador gera o máximo de documentação de erro que permite mais facilmente a solução do problema. O processo de resolução de erros num programa é conhecido como *depuração* (em inglês, *debug*).

Quando erros estão presentes no código-fonte o compilador reportará uma lista de todos eles (neste caso, no próprio terminal) indicando a linha onde cada um ocorre. Alguns erros são facilmente corrigíveis como *erros de sintaxe* (algo foi escrito erroneamente). Já outros requerem uma análise mais apurada do programador para serem corretamente diagnosticados e corrigidos.

Além de erros, um compilador pode gerar também *advertências*. Uma advertência (*Warning*) é um alerta que o compilador envia para indicar que algum recurso não está sendo apropriadamente utilizado. Erros impedem que o compilador crie o executável do programa ao contrário das advertências. Um programa compilado que gera somente advertências produz um binário. Na prática os efeitos de uma advertência para o programa que será executado pode ser imperceptível em alguns casos e prejudicial em outros. Consulte

o *guia do programador*, na documentação oficial do Free Pascal, para mais detalhes sobre erros e advertências de compilação.

Uma IDE (*Integrated development environment* ou ambiente de desenvolvimento integrado) é um programa utilizado para melhorar a produtividade de software. Eles constituem em geral de um editor de código-fonte com destacamentos de sintaxe, auto completação e uma série de ferramentas dispostas em menus e botões. Muitas IDEs são pagas e outras de uso livre. As IDEs mais populares para o desenvolvimento em linguagem Pascal e Object Pascal são *Delphi* e *Lazarus*. Entretanto diversos editores de texto elementares funcionam como IDEs sendo suficientes para escrever códigos fonte em Pascal.

Delphi ou Borland Delphi é na realidade a associação de um compilador (cuja linguagem é uma extensão do Object Pascal) e uma IDE orientada a desenvolvimento RAD (*Rapid Application Development* ou Desenvolvimento Rápido de Aplicação). Delphi era mantido inicialmente pela *Borland Software Corporation* (www.borland.com), mas atualmente a detentora é a *Embarcadero* (www.embarcadero.com). Delphi é direcionado a plataforma Windows e inclui atualmente versão para o framework .NET. Chegou a ser usado para desenvolvimento de aplicações nativas para Linux e Mac OS, através do Kylix. O desenvolvimento do Kylix foi descontinuado.

Lazarus (www.lazarus.freePascal.org) é uma IDE cuja interface é semelhante ao Delphi e que utiliza como compilador o Free Pascal. Trata-se de uma ferramente multiplataforma com versões para os sistemas Linux, Windows, OS/2, Mac OS, ARM, BSD, BeOS, DOS entre outros.

A exploração destas IDEs está fora do escopo deste livro. Ao invés disso apresentaremos brevemente a IDE que acompanha o próprio Free Pascal (*Free Pascal IDE*) e que pode ser executada via linha de comando digitando,

```
fp
```

Trata-se de uma IDE de interface modesta (sem recursos gráficos), mas de uso fácil e sem restrições de desenvolvimento, compilação ou execução de programas. O aspecto desta IDE é mostrado na Figura-1.1.

Figura 1 - IDE que acompanha o Free Pascal

Para criar, abrir ou salvar arquivos de código-fonte no Free Pascal IDE usa-se as opções new (novo), open (abrir), Save/Save as... (salvar/salvar como...) do menu File (arquivo) conforme Figura 2.

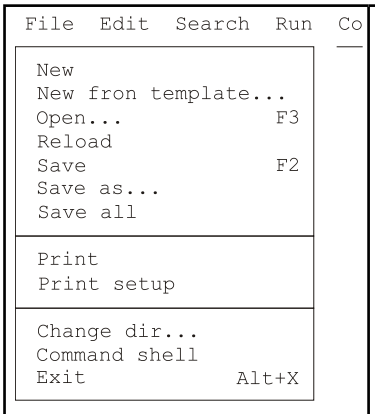


Figura 2 - menu File

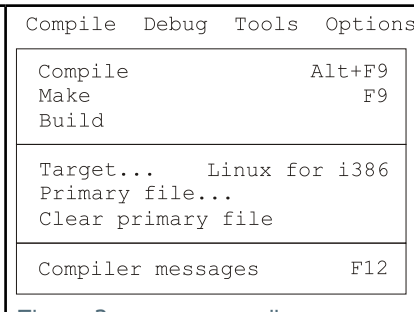


Figura 3 - menu compile

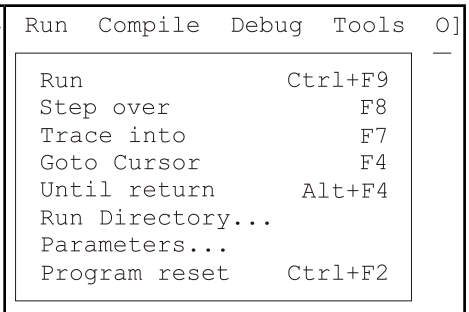


Figura 4 - Menu run

Para compilar o arquivo de código-fonte em edição use o menu Compile. Para compilação pura use Compile|Compile e para criar o executável use Compile|Build. A Figura 3 ilustra este menu. Para executar o programa na própria IDE utiliza o menu Run indo em Run|Run (Figura 4).

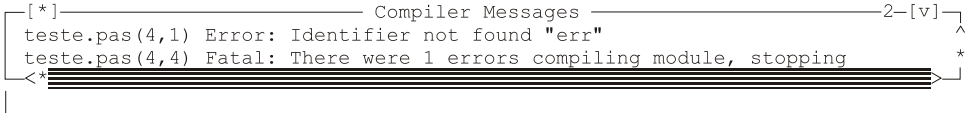


Figura 5 - Caixa de notificações do Free Pascal IDE

Quando erros e/ou advertências ocorrem durante a compilação, a IDE faz a listagem na caixa de notificação Compiler Messages (mensagens do compilador) que surge na parte inferior da IDE (Figura 5). Se erros não ocorrerem a IDE indica que a compilação obteve sucesso e exibe uma caixa como a mostrada na Figura 6.

```
─[*]────────── Compiling  (Normal mode) ──────────────┐
  Main file: /TEMP/TESTE/teste.pas                     │
  Done.                                                │
  Target: Linux for i386                               │
  Line number:      6     Total lines:           5     │
  Used memory:    128K    Allocated memory:    1504K   │
  Total errors:     0     Compile time:         0.0s   │
                                                       │
          Compile successful: Press any key            │
```

Para mais detalhes sobre a Free Pascal IDE consulte o guia do usuário (*user's guide*) na documentação do compilador.

4. Elementos Essenciais da Linguagem Pascal

Esta sessão tem por objetivo apresentar todos os elementos básicos construtivos da linguagem Pascal. Isso inclui regras elementares de sintaxe, variáveis, constantes, operadores, entrada e saída, controle de fluxo e uso básico da modularização. Este conteúdo é suficiente para a construção de diversos programas funcionais. Os capítulos retantes do livro tem por objetivo refinar ou potencializar os tópicos abordados nessa sessão.

4.1. Palavras Reservadas e Identificadores

Um código-fonte em Pascal é escrito pela associação dos chamados *tokens*. Tokens, por definição, são blocos léxicos básicos de código-fonte que representam as *palavras* da linguagem. Eles são formados pela combinação de caracteres de acordo com regras definidas na linguagem.

Palavras Reservadas são tokens com significado fixo numa linguagem de programação e desta forma não podem ser modificadas ou redefinidas. As palavras reservadas do Free Pascal somam aquelas provenientes da linguagem original (idealizada por Niklaus Wirth) mais aquelas oriundas das contribuições do Turbo Pascal e do Delphi Pascal. Ao invés de uma lista completa dessas palavras, apresentamos na Tabela-1.1 as palavras reservadas

utilizadas neste livro as quais são suficientes à programação estruturada aqui abordada.

and	for	not	string
array	function	of	then
asm	goto	on	to
begin	if	or	true
case	implementation	packed	type
const	in	procedure	unit
dispose	inline	program	until
div	interface	record	uses
do	label	repeat	var
exit	mod	set	while
false	new	shl	with
file	nil	shr	xor

Tabela 1 - Palavras reservadas da Linguagem Pascal

Identificadores são tokens definidos pelo programador que podem ser modificados e reutilizados. Identificadores são sujeitos às regras de escopo da linguagem.

Regras de Escopo: São regras que definem restrições de acessos entre partes distintas de um mesmo programa. Elas permitem, por exemplo, a implementação de programas grandes através de fragmentos menores fáceis de gerenciar.

Para escrever um identificador em Pascal deve-se seguir as premissas:

- O primeiro caractere (obrigatório) deve ser uma letra (intervalos a..z ou A..Z) ou um *underline* (_).
- Os demais caracteres (opcionais) podem ser letras, números (0..9) ou undelines.
- Não existe *sensibilidade de caixa*, ou seja, não há distinção entre maiúsculas e minísculas. Assim, por exemplo, os identificadores Casa, CASA e cASA são idênticos.

São todos exemplos de identificadores válidos em linguagem Pascal:

```
x a789    teste_56       dia_de_sol      __1987q _223    AxZ9    CARRO
```

4.2. Estrutura Básica de um Programa Pascal

O compilador Pascal precisa como entrada um ou mais arquivos de código para gerar um arquivo de código de máquina (executável). Estes arquivos de entrada são de fato arquivos texto podendo inclusive ser editados em editores de textos elementares. Como é usual em muitas IDEs a extensão destes arquivos é modificada para ".PAS" ou ".PP" sem comprometimento de tipo (ainda são de conteúdo texto). O menor código fonte compilável em Pascal é:

```
program teste; // linha de título
{ declarações }
begin
{ bloco principal }
end. {* fim do programa *}
```

Todo programa em linguagem Pascal deve iniciar com a palavra reservada program seguida de um identificador (nome do programa; neste caso, teste) seguido de ponto-e-vírgula. Logo abaixo devem ser colocadas, quando necessárias, as *declarações globais* do programa (global significa *disponível para todo programa*). Uma declaração é um código que viabiliza a utilização de um recurso do computador. As declarações em linguagem Pascal são dispostas em *sessões* (unidades, tipos, constantes, variáveis e módulos) que são estudadas ao decorrer do livro.

As respectivas palavras reservadas begin e end, representam os delimitadores do *bloco* ou *corpo principal* do programa: todo conteúdo entre begin (que significa *início*) e end (que significa *fim*) representa o programa propriamente dito a ser executado. A construção de um programa Pascal é enfim a implementação do código no bloco principal que eventualmente requer por recursos adicionais via declarações. O ponto (.) marcado após o end na última linha faz parte da sintaxe da linguagem e formaliza o término do programa.

Comentários são fragmentos de textos não compiláveis escritos conjuntamente ao código-fonte e cuja função é documentar o programa. Pode-se comentar conteúdo em Pascal colocando uma ou mais linhas entre chaves ou entre /* e */ ou prefixando cada linha com // (veja exemplos no código anterior).

Uma *sessão de unidades* declara bibliotecas conhecidas como *unidades* em Pascal. Unidades contêm sub-rotinas desenvolvidas por outros programadores que se tornam disponíveis nos programas onde são declaradas tais unidades (Várias unidades são distribuídas com o compilador Pascal e estão disponíveis após sua instalação).

Sub-rotina:
São trechos de código-fonte que funcionam como programas independentes. Programas e unidades em Pascal são de fato aglomerações de sub-rotinas. As sub-rotinas são também conhecidas como subprogramas. Adiante veremos que sub-rotinas em Pascal podem ser procedimentos ou funções e que por essa razão utilizamos o termo *módulo* para nos referirmos a ambos.

Listagem 1

```
01  program primeiro_exemplo;
02  uses crt, sysutils, nova_unidade;
03  begin
04  modulo1;
05  modulo2(34);
06  modulo3(12, 'teste');
07  end.
```

A Listagem 1 ilustra o uso de unidades em linguagem Pascal. A palavra reservada que abre uma sessão de declaração de unidades é uses. Todo conteúdo após uses e antes de outra sessão, ou do begin do programa principal, deverá ser de unidades devidamente separadas por vírgulas (quando há mais de uma). O final das declarações é sempre marcado com ponto-e-vírgula (;). Na Linha-02 vê-se a declaração de três unidades.

Cada módulo de uma unidade pode ser uma *função* ou um *procedimento* (veja Capítulo-5) o qual contém um nome (um identificador) que é usado para fazer uma *chamada* do módulo. Uma chamada de módulo representa sua invocação no programa principal através de uma linha que contém seu nome. Nas Linhas 04, 05 e 06, Listagem 1, vê-se três exemplos de chamadas a módulos. Na Linha 05 o módulo requer informação adicional que é repassada entre parênteses. Quando mais de uma informação é requerida pelo módulo, como na Linha-06, deve-se separá-las por vírgulas. Cada uma das informações fornecidas a um módulo é denominada de *argumento* ou *parâmetro* do módulo. As três chamadas de módulos na Listagem 1 possuem respectivamente zero, um e dois argumentos.

4.3. Variáveis

Uma *variável* é um artifício das linguagens de programação que permite o programador utilizar de forma iterativa a memória do computador. O programador define que variáveis o programa deverá utilizar e elas serão criadas na memória quando o programa for executado. Cada variável se vincula a uma célula de memória real de forma que as operações programadas em código afetarão a memória quando o programa estiver rodando.

Basicamente uma variável pode verificar ou modificar a célula de memória a qual está ligada. As operações de mudança e verificação de valor, por uma variável são chamadas respectivamente de operações de *escrita* e *leitura*. A leitura não afeta o conteúdo da varável ao passo que a escrita sobrescreve o valor que a variável contém com um novo valor (o valor sobrescrito é perdido).

A *sessão de declaração* de variáveis é a sessão do programa destinada a identificação (definição) de todas as variáveis que serão utilizadas. Tentar usar uma variável não declarada faz o compilador emitir um erro de sintaxe.

Cada nova variável precisa de um *nome* (que é um identificador) e um *tipo* (que é uma palavra reservada). O tipo dirá ao compilador que valores e operações serão suportados por aquela variável. Tipos podem ser *primitivos* e *derivados*. Os tipos primitivos são aqueles definidos nativamente pelo Pascal ao passo que os derivados são construídos pelo programador a partir de tipos primitivos (veja os Capítulos 2, 3 e 4 para mais detalhes).

A palavra reservada var marca o início de uma sessão de declaração de variáveis em Pascal. Um *grupo*, em uma declaração de variáveis, é um conjunto de nomes de variáveis que compartilham o mesmo tipo e que são sintaticamente separados por vírgulas. Cada lista de nomes de um grupo encerra em dois-pontos (:) seguida da palavra reservado ao tipo das variáveis deste grupo. Como a maioria das formações em Pascal, um grupo encerra com ponto e vírgula (;). Uma sessão pode possuir um ou mais grupos. Esquematicamente,

```
var
    <a_1>, <a_2>, ... ,<a_n> : <tipo_1>; // grupo de variáveis de tipo tipo_1
    <b_1>, <b_2>, ... ,<b_m> : <tipo_2>; // grupo de variáveis de tipo tipo_2
    ...
```

Os tipos primitivos da linguagem Pascal são subdivididos em cinco categorias,

- inteiros
- reais
- lógicos
- caracteres
- strings

Os tipos inteiros permitem a criação de variáveis para o armazenamento de números inteiros. Os tipos inteiros da linguagem Pascal distinguem-se basicamente pelo número de bytes e (consequentemente) pela faixa de valores que podem ser representados (quanto mais bytes, mais números podem ser representados por um tipo). A Tabela-1.2 apresenta as palavras reservadas para os dez tipos inteiros na Linguagem Pascal, bem como a faixa numérica dos valores cujas as variáveis podem representar e ainda o total de bytes que tais variáveis ocupam em memória quando o programa estiver em execução.

Tipo	Faixa	Tamanho (bytes)
Byte	0 .. 255	1
Shortint	-128 .. 127	1
Smallint	-32768 .. 32767	2
Word	0 .. 65535	2
Integer	smallint ou longint	2 ou 4
Cardinal	longword	4
Longint	-2147483648 .. 2147483647	4
Longword	0 .. 4294967295	4
Int64	-9223372036854775808 .. 9223372036854775807	8
QWord	0 .. 18446744073709551615	8

Tabela 2 - Tipos inteiros da Linguagem Pascal

A sessão a seguir declara algumas variáveis inteiras,

```
var
    a,b,c: integer; // grupo de varáveis integer de nomes a, b e c
    teste: word; // grupo de uma variável word de nome teste
    x: cardinal; // grupo de uma variável cardinal de nome x
```

Variáveis de tipo real armazenam números com possibilidade de casas decimais. Os tipos reais, similar aos inteiros, são tão representativos quanto maior for a quantidade de bytes associada. A diferença entre diferentes tipos reais reside na quantidade de bits dedicada a porção fracionária. A Tabela-1.3 lista as palavras reservadas aos seis tipos reais da linguagem Pascal bem como a faixa representativa e a quantidade de bytes (Conforme tabela, o tipo real depende da plataforma, ou seja, pode ter 4 ou 8 bytes conforme arquitetura de CPU utilizada).

Tipo	Faixa	Tamanho (bytes)
Real	depende da plataforma	4 ou 8
Single	1.5E-45 .. 3.4E38	4
Double	5.0E-324 .. 1.7E308	8
Extended	1.9E-4932 .. 1.1E4932	10
Comp	-2E64+1 .. 2E63-1	8
Currency	-922337203685477.5808 .. 922337203685477.5807	8

Tabela 3 - Tipos reais da Linguagem Pascal

A sessão a seguir declara algumas variáveis reais,

```
var
    angulo: real; // grupo de uma variável real de nome angulo
    pop, uni: extended; // grupo de duas variáveis extended de nomes pop e uni
```

Os tipos integer e real variam com a plataforma utilizada, ou seja, dependendo da arquitetura da CPU suas variáveis podem ser mais ou menos representativas. Recomenda-se que no caso do desenvolvimento de programas que devam rodar em mais de uma plataforma, esses tipos sejam evitados.

Tipos lógicos ou booleanos são utilizados para definir variáveis que assumam apenas os valores verdadeiro e falso. Estes valores são representados em Pascal respectivamente pelas palavras reservadas True e False. Uma variável lógica teoricamente é representada por um único bit, mas na prática existem quatro tipos booleanos em Pascal com pelo menos 1-byte, como mostrado na Tabela-1.4 (palavra reservada versus tamanho em bytes). O tipo boolean é o mais difundido sendo suficiente na maioria das aplicações que precisam de variáveis booleanas.

Tipo	Tamanho (bytes)
Boolean	1
ByteBool	1
WordBool	2
LongBool	4

Tabela 4 - Tipos lógicos da Linguagem Pascal

A sessão seguinte ilustra a declaração de variáveis booleanas,

```
var

    b, m, q: boolean; // grupo de três variáveis booleanas de nomes a, m e q
```

O tipo caractere, cuja a palavra reservada em Pascal é char, é uma variação do tipo inteiro byte. A faixa numérica 0..255 é substituída por 256 caracteres mapeados pelo equivalente byte. O tipo string, cuja palavra reservada em Pascal é string, refere-se a variáveis que devem armazenar um fragmento de texto. As strings constituem-se de sequências de caracteres e contam com um conjunto próprio de operações que as diferenciam dos demais tipos primitivos. Para saber mais sobre variáveis caracteres e strings, veja Capítulo 2.

Para determinar em Pascal o tamanho em bytes utilizado por uma variável utiliza-se o comando nativo de linguagem sizeof. Este comando pode receber tanto o nome da variável, como em,

```
sizeof(x);
```

como o nome de tipo,

```
sizeof(Integer);
```

No caso de tipos primitivos, sizeof possui aplicação importante na determinação dos tamanhos de integer e real haja vista dependerem da plataforma utilizada (hardware).

4.4. Constantes

Uma *constante* é um artifício das linguagens de programação que permite a manutenção de valores invariantes durante toda execução do programa. Uma constante pode aparecer num código como parte integrante de uma expressão (veja próxima sessão) ou vinculada a uma célula de memória independente (como nas variáveis), mas com suporte apenas a *verificação* (nunca a *escrita*). Nas expressões,

```
x*25

3.45-y

'a' + ch

'hoje foi' + st

b/100
```

x, ch, st e b são variáveis e 25, 3.45, 'a', 'hoje foi' e 100 são constantes. As variáveis podem mudar de valor em diferentes execuções ao passo que as constantes mantêm os mesmos valores. Constantes também possuem tipos. Nos exemplos anteriores, 25 e 100 são constantes de tipo inteiro, 3.45 de tipo real, 'a' de tipo caractere e 'hoje foi' de tipo string (constantes strings e caracteres são, em Pascal, sempre colocadas entre aspas simples. Para saber mais sobre caracteres e strings, veja Capítulo 2).

Para construir uma constante com célula própria de memória, deve-se construir uma *sessão de declaração de constantes*. A palavra reservada para início deste tipo de sessão é const. Cada nova constante possui a sintaxe fixa,

```
const
      <nome_da_constante> = <valor>;
```

A sessão a seguir ilustra a definição de cinco novas constantes em Pascal,

```
const
    cx = 25;                    // constante de tipo inteiro
    max = 6.76;          // constante de tipo real
    msg = 'ola mundo';  // constante de tipo string
    meu_ch = 'a';               // constante de tipo caractere
    my_bool = True;             // constante de tipo booleano
```

O compilador reconhece o tipo das constantes pelo valor na definição. Não é possível, em Pascal, definir constantes sem fornecer um valor.

O programador pode também forçar que uma constante seja de um tipo específico. São as chamadas *constantes tipadas*. Elas seguem a sintaxe fixa,

```
const
      <constante>: <tipo> = <valor>;
```

A sessão seguinte traz apenas constantes tipadas,

```
const
      q: integer = 11;
      ex: real = 3;
      st: string = 'x';
```

Observe que a tipagem de q não é necessário pois o compilador detectaria 11 como inteiro. Nos casos de ex e st a tipagem é necessária pois, sem elas, ex seria inteiro e st caractere.

4.5. Operadores e Expressões

O mecanismo utilizado para modificar o conteúdo de uma variável é denominado *atribuição explicita* ou simplesmente *atribuição*. Em linguagem Pascal uma atribuição é construída com o *operador de atribuição* (:=) seguindo a sintaxe,

```
<variável> := <expressão>;
```

Semanticamente a expressão do lado direito do operador é resolvida e o valor resultante é gravado na variável do lado esquerdo. Deve existir em Pascal apenas uma variável do lado esquerdo de uma atribuição.

Listagem 2

```
01  program atribuicoes;
02  var
03     i: integer;
04     x, info_2: real;
05     db: double;
06     b: boolean;
07  begin
08     i := 35;
09     x := 4.87;
10     info_2 := x * i;
11     db := x*x + 6.0*i + 45;
12     b := True;
13  end.
```

A Listagem 2 demonstra a construção de cinco atribuições com expressões diversificadas. O operador de atribuição é apenas um exemplo de operador. A linguagem Pascal disponibiliza diversos operadores divididos nas categorias,

- Operador de atribuição
- Operadores aritméticos
- Operadores relacionais
- Operadores lógicos
- Operadores de ponteiros
- Operadores bit-a-bit

Os operadores aritméticos da linguagem Pascal são listados na Tabela 5. Estes operadores são binários (possuem dois operandos) e infixos (colocados entre os operandos). Os operandos devem ser necessariamente valores numéricos e o resultado da operação é também um valor numérico. As operações de soma, subtração e produto, quando realizadas entre dois operandos inteiros, resultará num valor inteiro.

Entretanto, se qualquer um dos operandos for real (ou ambos), o valor final será real. O operador de divisão, /, sempre retornará um valor real, mesmo que os operandos sejam inteiros (ocorrerá um erro de execução se o divisor for zero). Os operador div (quociente de divisão) e mod (resto de divisão) só operam valores inteiros e o resultado da operação é também um inteiro. Usar estes operadores com variáveis reais ou qualquer outro tipo de variável, representa um erro de sintaxe (que o compilador acusará).

Operador	Função
+	Soma
-	Subtração
*	Produto
/	Divisão
div	Quociente de divisão
mod	Resto de divisão

Tabela 5 - Operadores Aritméticos em Pascal

Uma *expressão aritmética* pode ser um valor constante, uma variável ou ainda uma associação entre constantes, variáveis e operadores aritméticos (o uso de parênteses também é permitido). Numa atribuição o resultado final da expressão no lado direito deve ser compatível com o tipo da variável que aguarda do lado esquerdo. Na Listagem-1.2, o resultado de expressões aritméticas são atribuídos às variáveis i, x, info_2 e db. São outros exemplos legais de expressões aritméticas em Pascal,

```
12+x*(a-b*3.0) (x div 10)*10 beta mod 2 + 1 a+b*c div 2
```

Denomina-se *precedência de operações* a ordem que o compilador resolve as sub operações que constituem uma expressão. Numa expressão aritmética a sequencia de resoluções é a mesma definida pela matemática, ou seja, primeiro as multiplicações (*) e divisões (/, div ou mod), depois adições (+) e subtrações (-). Assim em, x+3*y, é feita primeira a multiplicação depois a soma. Se operadores de mesma precedência aparecem adjacentes numa mesma expressão, o compilador fará a avaliação seguindo a sequencia em que estão, da esquerda para a direita. Assim por exemplo a expressão, 20 mod 11 div 2, resulta em 4 porque mod é resolvido antes de div. De forma similar a expressão, 100 div 3 div 2 div 5, é reduzida pelo compilador na sequencia,

```
100 div 3 div 2 div 5

33 div 2 div 5

16 div 5

3
```

Para quebrar a ordem de precedência seguida nas avaliações de expressões, pode-se utilizar parênteses como em, (3+x)*y. Neste exemplo a soma é realizada primeiro.

Operadores relacionais são aqueles que comparam dois valores (que podem ser ou não numéricos) e resultam num valor booleano, ou seja, verdadeiro ou falso. Por exemplo a expressão,

```
x > 10
```

utiliza o operador relacional *maior que* (>) e retorna verdadeiro (se o valor em x é maior que 10) ou falso (se o valor em x é menor ou igual a 10). A tabela seguinte lista os operadores relacionais da linguagem Pascal e seus respectivos significados:

Operador Relacional	Significado
\	Maior que
Z	Menor que
\ [	Maior ou igual a
Z [	Menor ou igual a
Z \	Diferente de
[	Igual a

Tabela 6 - Operadores relacionais em Pascal

Operadores relacionais em Pascal são *não associativos*, ou seja, dois operadores relacionais não devem aparecer adjacentes na mesma expressão. Assim, por exemplo, a expressão, x>y>10, é ilegal em Pascal. Expressões formadas com operadores relacionais são denominadas expressões booleanas. De fato qualquer expressão que retorne um booleano como valor final é uma expressão booleana.

Operadores lógicos são operadores que modificam ou conectam entre si expressões booleanas. Por exemplo, na expressão,

```
(x>y) and (y>10)
```

o operador lógico and conecta duas expressões relacionais. Veja Tabela 7 com os operadores lógicos do Pascal.

O operador de conjunção, and, é binário, infixo e retorna verdadeiro apenas quando as duas expressões (que são seus operandos) são ambas verdadeiras. Os operadores de disjunção, or, e disjunção exclusiva, xor são

também binários e infixos. Na disjunção basta que um dos operandos seja verdadeiro para que a resultante também o seja. Na disjunção exclusiva a resultante é verdadeira quando apenas uma das expressões for verdadeira. O operador de negação, not, é unário (apenas um operando) e prefixo (disposto à esquerda do operando). A negação resulta no inverso do valor da expressão booleana.

Operador Lógico	Significado
`and`	Conjunção
`or`	Disjunção
`xor`	Disjunção exclusiva
`not`	Negação

Tabela 7 - Operadores lógicos em Pascal

Operadores lógicos possuem precedência de operações. O operador lógico de maior precedência é o not seguido por and, depois or e por fim, o de menor precedência, xor. Por exemplo, a expressão, not true or false and true, é avaliada na sequencia,

```
not true or false and true

false or false and true

false or false

false
```

Se operadores lógicos de mesma precedência aparecem adjacentes na mesma expressão o compilador efetuará a avaliação da esquerda para a direita.

A Tabela 8 ilustra a *tabela verdade* dos operadores lógicos em Pascal (A e B representam expressões booleanas)

A	B	A and B	A or B	A xor B	not A
TRUE	TRUE	TRUE	TRUE	FALSE	FALSE
TRUE	FALSE	FALSE	TRUE	TRUE	FALSE
FALSE	TRUE	FALSE	TRUE	TRUE	TRUE
FALSE	FALSE	FALSE	FALSE	FALSE	TRUE

Tabela 8 - Tabela verdade dos operadores lógicos em Pascal

Em Pascal, expressões booleanas conectadas com operadores lógicos, precisam estar entre parênteses. Isso acontece porque os operadores lógicos têm precedência sobre os operadores relacionais de forma que uma expressão do tipo, x>0 and x<100, sem parênteses, tem primeiramente avaliado o and, que neste caso significa operar a sub expressão, 0 and x. Mesmo que isso tivesse algum sentido para o programador ainda restariam por se re-

solver dois operadores relacionais adjacentes, o que é inviável. As expressões booleanas a seguir são todas legais em linguagem Pascal.

Expressão	**Verdadeira se o Valor em ...**
`x>10`	x é maior que dez
`y<=7`	y é menor ou igual a sete
`(x>11) and (x<=100)`	x é maior que onze e ao mesmo tempo menor ou igual a cem
`zeta <> mega`	zeta é diferente do em mega
`(a<100) or (b>500)`	a é menor que cem ou o em b é maior que quinhentos.
`(dia=seg) xor (dia=qua)`	dia não é segunda nem quarta
`not((x=y) and (a=b))`	x é diferente do em y ou o em a é diferente do em b

Operadores de ponteiros serão estudados no Capítulo 4. Operadores bit a bit não serão explorados neste livro.

4.6. Comandos de Entrada e Saída

A entrada de um programa é usualmente realizada através de um mouse e um teclado ao passo que a saída através da tela ou de uma impressora. Para facilitar o trabalho dos usuários é comum distribuir o programa com uma interface gráfica (uma janela com botões, menus, caixas de rolagem e etc). O desenvolvimento de interfaces gráficas (GUIs) requer mais que a programação estruturada apresentada neste livro. Para este nível de programação o leitor deverá estender seus estudos ao Object Pascal (www.freePascal.org) e depois explorar o uso de IDEs voltadas a criação de GUIs, como é o caso do Delphi (www.borland.com/br/products/delphi) que é comercial ou do Lazarus (www.lazarus.freePascal.org) que é livre.

Os detalhes de implementação de uma GUI são substituídos aqui por uma janela de terminal completamente orientada a conteúdo textual e onde são visualizadas todas as operações de entrada e saída (no Capítulo-6 estenderemos os mesmos conceitos a arquivos). Um programa executado num terminal via linha de comando (use o command no Windows ou um interpretador de linha de comando como o bash no Linux) imprime no próprio terminal tanto o conteúdo de entrada que é digitado em teclado quanto a saída gerada pelo programa.

A unidade em Pascal que contém módulos responsáveis por operações de entrada e saída é a unidade system. Seu uso é tão frequente que ela dispensa declaração, ou seja, um programa sem qualquer declaração de unidades ainda utiliza a unidade system. Outras categorias de módulos desta

unidade serão vistas ao decorrer do livro. Os módulos de entrada e saída em Pascal (comumente chamados *comandos de entrada e saída*) são,

write	Escreve na saída padrão o conteúdo de sua lista de argumentos. Podem haver um ou mais argumentos neste comando.
writeln	Funciona de forma similar a write acrescentando uma nova linha depois de escrever seus argumentos. Pode ser utilizado sem argumentos para deixar uma linha em branco.
read	Lê da entrada padrão (normalmente o teclado) um ou mais valores e os dispõe em seus argumentos obedecendo a sequencia de leitura. Os argumentos deste comando devem ser necessariamente variáveis.
readln	Funciona como read mas ainda pode ser utilizado sem argumentos simplesmente para aguardar o pressionar de tecla ENTER (quando o teclado é a entrada padrão).

Listagem 3

```
01  program entrada_e_saida;
02  var
03     x: integer;
04  begin
05     write('Fornecer valor de x: ');
06     read(x);
07     writeln('o dobro de x vale ', x*2);
08     readln;
09  end.
```

O programa da Listagem 3 ilustra a utilização dos comandos de entrada e saída em Pascal. Na Linha-05 o comando write é utilizado para escrever uma *string*, Uma string em Pascal é uma sequencia de caracteres entre aspas simples (‘ ‘) representando um trecho de texto. A impressão de uma string por write apenas reproduz seus caracteres na saída padrão. Veja Capítulo-2 para mais detalhes sobre strings em Pascal.

Na Linha-06 ocorre uma leitura de valor via teclado e o valor final digitado é armazenado na variável x. É importante observar que o comando read suspende momentaneamente a execução e aguarda que o usuário digite alguma coisa. Somente quando confirmada a entrada (normalmente com um ENTER) é que o programa prossegue suas execução. Na Linha-07 writeln recebe dois argumentos e os imprime na sequencia que aparecem (atente que a vírgula de separação de parâmetros não é impressa!).

O primeiro argumento é uma string e é impressa da mesma forma que aparece no código. O segundo argumento é uma expressão que deve ser resolvida pelo programa e o resultado (dobro do valor em x) impresso por writeln. Na Linha-08 o programa aguarda por um ENTER adicional antes de encerrar. A sequencia em terminal mostrada a seguir exemplifica a execução do programa da Listagem 3 (o underline, _, representa o cursor e o conteúdo entre < e > é apenas informativo e não consta na saída do programa),

1. Início da execução. Aguarda x:	**2. Valor de x fornecido:**
Fornecer valor de x: _	Fornecer valor de x: 32_ <enter>
3. Impressão da resposta:	**4. Encerramento:**
Fornecer valor de x: 32 o dobro de x vale 64 _	Fornecer valor de x: 32 o dobro de x vale 64 _

É possível modificar a formatação de valores numéricos impressos por write (ou writeln). Isso inclui alterar tanto o comprimento global do número (medido em caracteres) quanto o número de casas decimais utilizadas (no caso de valores reais). O modificador dois-pontos (:) formata a impressão de variáveis inteiras e reais pelas respectivas sintaxes,

```
<variável_inteira> : <comp>

<variável_real> : <comp> : <num_decs>
```

O valor de uma variável é impresso com o total de caracteres dado em *comp*, dos quais são reservados, para os decimais, o total de *num_decs* (caso a variável seja real). Se o comprimento comp é insuficiente, então é usado um valor de comprimento mínimo que inclua completamente a parte inteira do número. Se a representação requer menos que *comp* caracteres, então a saída será preenchida com espaços em banco à esquerda (alinhamento à direita). Se nenhuma formação é especificada, então o número é escrito usando seu comprimento natural, sem prefixação, caso seja positivo, e com um sinal de menos, caso seja negativo. Números reais não formatados são escritos em notação científica. A seguir alguns exemplos de formatação,

Var.	Tipo	Valor	Exemplo	Saída
a	integer	81	writeln('tenho',a,'anos');	tenho81anos
b	real	37.92	write(b);	3.79200000000000E+001
x	integer	35	write('Carros: ', x:6)	Carros: 35
y	real	6.7541	Writeln('>',y:7:2,' gramas');	> 6.75 gramas

4.7. Tomando Decisões

Dependendo do estado de uma variável ou do resultado de uma expressão um programa poderá tomar uma *decisão* relativa a um caminho ou outro que deva seguir. Na prática isso significa desviar-se de ou ir direto para um trecho de código específico. Em termos de implementação uma decisão requer que o programador codifique duas ou mais possibilidades de direções que um programa pode seguir num dado instante de sua execução. A direção escolhida só é conhecida em tempo de execução e pode mudar entre execuções distintas.

A linguagem Pascal suporta três formas de estruturação de decisões: *Decisão Unidirecional*, *Decisão Bidirecional* e *Decisões Múltiplas*.

4.7.1. Decisões Unidirecional e Bidirecional

Uma decisão unidirecional é aquela que deve decidir entre executar um bloco de código ou não fazer nada. Sua estrutura fundamental em Pascal é dada por,

```
if <teste> then
    <bloco>;
```

As palavras reservadas if (que significa *se*) e then (que significa *então*) marcam o início de um processo de decisão. Elas reservam entre si uma expressão booleana, representada aqui por <*teste*>. Quando o resultado é verdadeiro o conteúdo de <bloco> é executado. Do contrário o processo se encerra e o programa continua sua execução. O ponto-e-vírgula (;) marca o fim sintático da gramática de decisão de forma que se <teste> é falso então o programa desvia sua execução para logo após esta marcação.

Uma decisão bidirecional é aquela que deve decidir entre executar um de dois blocos sem possibilidade de execução de ambos. Sua estrutura fundamental em Pascal é dada por,

```
if <teste> then
   <bloco_1>
else
   <bloco_2>;
```

Quando <teste>, na decisão bidirecional, é verdadeiro então o código em <bloco_1> é executado. Do contrário o código em <bloco_2> o será. A nova palavra reservada else (que significa *senão*) é que provoca o desvio para <bloco_2> quando o teste falha. A marcação de ponto-e-vírgula (;) após <bloco_2> é obrigatória pois indica o final sintático da gramática de decisão, mas não ocorre após <bloco_1> devido a estrutura não ter se concluído ainda.

Os blocos ilustrados nas estruturas de decisão anteriores podem possuir uma ou mais linhas de comando. Define-se *linha de comando* como uma instrução de programa cuja sintaxe é localmente independente. Cada linha de comando em Pascal encerra num ponto-e-vírgula (;).

Quando existe apenas uma linha de comando numa decisão, ela deve ser escrita logo após o if..then. Para o compilador é indiferente se isso for feito na mesma linha de texto, como em,

```
if x>10 then writeln(x);
```

valor em `x` impresso somente se for maior que 10.

ou em linhas de texto adjacentes, como em,

```
if (angulo<3.14) then
write('Ola Mundo!');
```

a string `'Ola Mundo'` só é impressa se o valor em `angulo` for inferior a 3.14.

Expressões booleanas completas podem fazer parte de uma estrutura de decisão como em,

```
if (y<=80) and (x>76) then
readln(beta);
```

Se o valor em `y` for menor ou igual a 80 e ao mesmo tempo o valor em `x` superior a 76, então o programa suspende provisoriamente para ler um valor e armazená-lo em `beta`.

O compilador, entretanto, não reconhece, como pertencente à decisão, mais de uma linha de comando adjacente como em,

```
if x<>100 or y<>67 then
write(x+y)
write('Oi');
```

Caso o valor em `y` seja diferente de 100 ou o em `y` diferente de 67 então a soma deles será impressa. A impressão da mensagem `'Oi'`, apesar do alinhamento enganoso, *independe* do teste pois não faz parte do bloco de decisão.

No exemplo a seguir um else está presente (decisão bidirecional) e cada bloco possui apenas uma linha de comando.

```
If p div 2 <> 0 then
writeln(p)
else
writeln(p/2);
```

Se o valor em `p` for ímpar ele será impresso integralmente, do contrário será impresso metade de seu valor.

Se uma sequência formada por mais de uma linha de comando precisa de execução condicional, então a formação do bloco deverá incluir as palavras chaves begin e end. Estas são respectivamente dispostas antes e depois das linhas de comando formando o novo bloco da estrutura if..then. As estruturas finais, nas decisões unidirecional e bidirecional, envolvendo blocos de linhas de comando, são respectivamente:

```
if <teste> then
begin
<linhas de comando>
end;
```

```
if <teste> then
begin
<linhas de comando>
end else begin
<Linhas de comando>
end;
```

Decisões podem ocorrer uma dentro da outra, ou seja, dentro de um bloco de decisão pode haver uma outra decisão. Este mecanismo é conhecido como *aninhamento de decisões* e teoricamente não possui restrições. Para ilustrar o aninhamento de decisões observe o esquema a seguir,

```
1  if <teste1> then
2     if <teste2> then
         <bloco1>
2     else
         <bloco2>
1  else
3     if <teste3> then
         <bloco3>
```

O if..then..else marcado com 1 é bidirecional e aninha, no primeiro bloco, o if..then..else bidirecional marcado com 2 e no segundo bloco o if..then unidirecional marcado com 3. Dessa forma $bloco_1$ só executa se $teste_1$ e $teste_2$ forem verdadeiros; $bloco_2$ só executa se $teste_1$ for verdadeiro e $teste_2$ for falso; e por fim $bloco_3$ só executa se $teste_1$ é falso e $teste_3$ verdadeiro.

4.7.2. Decisões Múltiplas

Uma decisão múltipla em Pascal é aquela que lida com diversas possibilidades de caminhos a se seguir. Cada possibilidade é representada por um bloco de código e apenas um destes blocos terá o conteúdo executado. Como as expressões booleanas só encaminham para no máximo duas rotas, elas não são a base de construção de uma decisão múltipla em Pascal. Ao invés disso utilizam-se expressões aritméticas que resultem em valores inteiros. Desta forma cada bloco de decisão fica atrelado a um valor inteiro distinto. A estrutura de decisão múltipla em Pascal tem sintaxe,

```
case <expressao_aritmetica_inteira> of
   <rotulo_1>: <bloco_1>;
   <rotulo_2>: <bloco_2>;
   …
   <rotulo_n>: <bloco_n>;
   else <bloco_alternativo>;
end;
```

As palavras reservadas case..of (que significam *caso..seja*) formam a coroa da estrutura de decisão múltipla que necessariamente deve ser encerrada em end mais ponto e vírgula. A expressão aritmética entre case e of deve resultar num dos valores inteiros rotulados no código do corpo da estrutura. Cada rótulo deve ser seguido de dois pontos (:) e do bloco de decisão (encerrado em

ponto e vírgula). Um bloco pode ser uma linha de comando ou um conjunto delas encapsuladas por begin/end. Se o valor da expressão avaliada não se igualar a nenhum dos rótulos, será executado o *bloco alternativo* que é marcado por um else. O compilador não exige a presença de um bloco alternativo em decisões múltiplas. Além de case..of..end; e da expressão aritmética, a estrutura exige no mínimo um rótulo e seu bloco.

A Listagem 4 ilustra o uso de decisões múltiplas. Dependendo do valor de op fornecido pelo usuário, um de cinco blocos será executado. Atenue ao bloco de rótulo 4 que encapsula uma decisão unidirecional demonstrando a flexibilidade de associação de estruturas em Pascal.

Listagem 4

```
01 program operacoes_basicas;
02 var op: integer;
03     a, b: real;
04 begin
05     writeln('Menu');
06     writeln('1-- somar');
07     writeln('2 - subtrair');
08     writeln('3 - multiplicar');
09     writeln('3 - dividir');
10     readln(op);
11     writeln('valores:');
12     readln(a, b);
13     case op of
14        1: writeln('soma: ', a+b:0:3);
15        2: writeln('subtracao: ', a-b:0:3);
16        3: writeln('multiplicacao: ', a*b:0:3);
17        4: if abs(b)>1.0e-5 then
18              writeln('divisao: ', a/b:0:3);
19        else writeln('opcao invalida');
20     end;
21 end.
```

Se mais de uma valor resultante da expressão de entrada na estrutura case..of precisar acionar o mesmo bloco, poderá se usar a *listagem de rótulos* como ilustrada a seguir,

```
case x of
   1,2,3: write('ok');
   4: write('tchau!');
   else write('invalido!');
end;
```

O primeiro bloco deste exemplo poderá ser executado quando a variável x conter 1, 2 ou 3. Uma listagem de rótulos usa vírgulas para separar os rótulos que devem conduzir a um mesmo bloco.

4.8. Utilizando Laços

Laços, ou *loops*, são estruturas que permitem a repetição da execução de um mesmo trecho de código. Eles possibilitam a automatização de tarefas que necessitam ser processadas diversas vezes. Denomina-se *iteração* cada uma das repetições de um laço. A *entrada de uma iteração* é o estado das variáveis que participam do código do laço. A entrada de uma dada iteração pode, ou não, afetar a entrada da iteração posterior. Uma ou mais linhas de comando podem ser repetidos por um laço.

Há três estruturas de construção de laços em Pascal: *laços com contador*, *laços de pré-teste* e *laços de pós-teste*.

4.8.1. Laços com Contador

Um laço com contador é aquele que utiliza uma variável auxiliar denominada *contador*, dois valores limites denominados *limite inferior* e *limite superior* e um valor de *passo*. O laço inicia configurando o contador com o limite inferior. Em cada iteração o contador é adicionado do passo. O laço encerra quando o contador se iguala ou extrapola o limite superior. Cada iteração se destaca por ser processada com seu próprio valor do contador.

Em Pascal o contador é necessariamente uma variável de tipo inteiro. Os limites devem possuir também valores inteiros e o passo vale sempre 1 (*laço crescente*) ou -1 (*laço decrescente*). Em laços crescentes o limite inferior deve ser menor ou igual ao superior ou então nenhuma iteração se procederá. Em laços decrescentes ocorre exatamente o inverso. A estrutura de um laço de contagem crescente em Pascal é a seguinte:

```
for <contador> := <limite_inferior> to <limite_superior> do
   <bloco_de_repetição>
```

Esta estrutura utiliza as palavras reservadas for (que significa *para*), to (que, neste contexto, significa *até*) e do (que significa *faça*). O operador de atribuição (:=) é parte integrante também da estrutura indicando que o contador sofre uma nova atribuição em cada iteração que se inicia. A estrutura de um laço decrescente se distingue do crescente apenas pela substituição de to pela palavra reservada downto, ou seja:

```
for <contador> := <limite_inferior> downto <limite_superior> do
   <bloco_de_repetição>
```

O trecho a seguir ilustra o uso de laços de contagem crescentes,

```
for i := 1 to 100 do
   write(i, '');
```

Os valores de 1 a 100 são impressor separados por espaços em branco. A variável `i` é inteira.

Para gerar a mesma saída em ordem decrescente faz-se:

```
for i := 100 downto 1 do
   write(i, ' ');
```

Os valores de 1 a 100 são impressor separados por espaços em branco e em ordem decrescente.

Quando mais de uma linha de comando está presente no mesmo laço, deve-se utilizar um bloco begin/end similar àquele utilizado em estruturas de decisão. Esquematicamente tem-se,

```
for <contador> := <limite_inferior> to/
downto <limite_superior> do
begin
   <linhas_de_comando>
end;
```

O trecho a seguir ilustra o uso de laços de contagem com mais de uma linha de comando para repetição,

```
for i := 1 to 100 do
begin
   k := 101 - i;
   write(k, ' ');
end;
```

Os valores de 1 a 100 são impressos separados por espaços em branco e em ordem decrescente. Como o laço é crescente utilizou-se a variável `c` , inteira, para calcular o valor de impressão.

É possível acoplar decisões a laços fazendo *toda* estrutura de decisão se tornar o bloco da estrutura de repetição. Este mecanismo é bastante flexível e permite a implementação de algoritmos mais sofisticados. O exemplo a seguir ilustra o acoplamento entre um laço de contagem e uma decisão unidirecional,

```
for i := 1 to 1000 do
   if (i mod 7 = 0) and
(i mod 3 <> 0) then
      write(i, ' ');
```

Imprime inteiros entre 1 e 1000 que são divisíveis por 7 (o resto da divisão por 7 é zero, ou `i mod 7 = 0`) *e* não divisíveis por 3 (o resto da divisão por 3 é não nulo, ou seja, `i mod 3 <> 0`)

Laços podem ser *aninhados* em Pascal. Aninhar significa construir laços dentro de outros. Não há restrições da quantidade de aninhamentos numa mesma construção. Cada laço em um aninhamento representa um *nível* deste aninhamento. No exemplo,

```
for i:= 1 to 10 do
   for j := 1 to 10 do
      write(i*j);
```

há dois níveis de aninhamento. Diz-se que o segundo laço for está aninhado no primeiro. Semanticamente cada iteração do primeiro laço for provoca a execução das dez iterações do segundo laço for. Assim a instrução write é executada um total de 100 vezes. No exemplo,

```
for i:= 1 to 10 do begin
   for j := 1 to 10 do write(i+j);
   for j := 1 to 10 do write(i+j);
end;
```

o primeiro laço aninha outros dois laços que por sua vez *não* se aninham. Assim existem apenas dois níveis de aninhamento neste exemplo. O comando write é executado 200 vezes.

4.8.2. Laços com Pré-teste

Expressões booleanas podem ser utilizadas para montar laços. A mais usual é o laço com pré-teste no qual uma expressão é testada no início de cada iteração. Enquanto o valor da expressão resultar em verdadeiro o laço deve

prosseguir. Do contrário ele se encerra. A sintaxe básica de um laço pré-teste em Pascal é dada por,

```
while <expressao> do
   <bloco>
```

As palavras reservadas while (que significa *enquanto*) e do (que significa *faça*) encapsulam a expressão booleana, <expressao>. Enquanto o valor desta expressão for verdadeiro o código em <bloco> é repetido. Este bloco pode ser formado por uma linha de comando ou por um conjunto delas desde que encapsuladas pelas palavras reservadas begin/end;.

É importante observar que o estado das variáveis que formam a expressão devem ser modificadas dentro do bloco (ou por um agente externo ao programa), ou do contrário o laço se tornará infinito (ou nunca será executado). No exemplo,

```
j := 1;
while j < 100 do
begin
   write(j, ' ');
   j := j+1;
end;
```

os inteiros de 1 a 99 são impressos em saída padrão.

a variável j age como contador que é iniciado fora do laço. A atribuição, j := j+1, permite que a expressão booleana de entrada do laço, j<=100, diferencie sua avaliação de uma iteração para outra. Na primeira iteração do laço, j contém 1 e é este o valor impresso por write. Nas demais iterações a incrementação de j geram novas impressões. Quando 99 é impresso o valor em j é incrementado para 100 na linha seguinte. Quando o teste de entrada executar na próxima vez (e ele ainda o será), o avaliador encontrará a instrução 100<100 que resulta falso: neste ponto o laço é encerrado. Assim 99 é o último valor impresso.

O laço anterior pode ser facilmente substituído por um laço de contagem. Há situações entretanto que os laços de contagem não podem resolver, como em,

```
x := 145;
while x > 0 do begin
   write(x mod 2);
   x := x div 2;
end;
```

Imprime a representação bin[aria de 145, de trás para frente, ou seja, 10001001.

4.8.3. Laços com Pós-Teste

Laços de pós teste também são baseados em expressões booleanas para controle de fluxo. Neste caso entretanto o teste fica *à saída* da estrutura. Em Pascal a semântica de um laço pós-teste é a seguinte: repetir um bloco de código *até que determinada condição seja satisfeita*. A sintaxe básica de um laço pós-teste em Pascal é dado por,

```
repeat
   <bloco>
until <expressao>;
```

As palavras reservadas repeat (que significa *repita*) e until (que significa *até*) encapsulam o bloco de código que deve ser repetido. Esta sintaxe, ao contrário das apresentadas até então, dispensa o uso das palavras reservadas begin/end mesmo quando várias linhas de comando constituem <bloco>. O valor da avaliação de <expressao> precisa ser *falso* para garantir a próxima iteração. No momento em que essa expressão é avaliada como *verdadeira*, o laço se encerra (*repita até que isso seja verdade*). Num laço pós-teste a primeira iteração sempre acontece (o que não é verdade para laços pré-teste). No exemplo,

```
j := 1;
repeat
   write(j, ' ');
   j := j+1;
until j>100;
```

os valores de 1 a 100 são impressos na saída padrão. O teste de parada é aquele onde j vale 101 e por essa razão o 100 também é impresso (lembre que no exemplo análogo apresentado com laço pré-teste, o 100 não era impresso).

A obrigatoriedade de execução da primeira iteração num laço pós-teste pode ser interessante em algumas situações como a mostrada na Listagem 5. Este programa lê uma quantidade indefinita de inteiros e exibe a média ao final. Para executar esta tarefa, um readln é utilizado dentro de um laço pós-teste (Linha-08) que recebe através de x cada um dos valores fornecidos pelo usuário. Quando um valor zero for fornecido o laço é encerrado (o zero não entra na média). O teste da Linha-09, que também está dentro do laço, faz com que o valor lido através de x, numa dada iteração, seja acumulado em s (Linha-11). Esta variável (previamen-

te anulada fora do laço, Linha-04) conterá, no final das iterações, a somatória dos valores fornecidos pelo usuário (excluindo o zero). A variável auxiliar n permite a contagem de valores lidos. Ela é anulada antes do laço (Linha-05) e incrementada em uma unidade (Linha-12) cada vez que um novo valor (não nulo) é lido através de x. A média final é naturalmente s/n e é impressa na Linha-16 desde que o valor em n não seja nulo (verificação feita pelo teste da Linha-15). Se n contém zero, a mensagem da Linha-18 é impressa ao invés da média.

Listagem 5

```
01  program media;
02  var s, x, n: integer;
03  begin
04     s := 0;
05     n := 0;
06     repeat
07        write('Valor: ');
08        readln(x);
09        if x <> 0 then
10        begin
11            s := s+x;
12            n := n+1;
13        end;
14     until x=0;
15     if n>0 then
16        write('media: ', s/n :0:2)
17     else
18        write('nenhum valor fornecido!');
19  end.
```

4.8.4. Break e Continue

Quebrar um laço significa suspender seu fluxo de execução e continuar a execução do programa a partir da primeira instrução que sucede o corpo estrutural do laço quebrado. Para quebrar um laço (seja com contador, de pré-teste ou de pós-teste) a linguagem Pascal conta com uma instrução específica cuja palavra reservada é break. Utilizando break é possível, por exemplo, reescrever o laço da Listagem-1.5 utilizando while..do conforme segue,

```
while True do begin
   write('Valor: ');
   readln(x);
   if x = 0 then break;
   s := s+x;
   n := n+1;
end;
```

Este laço é teoricamente infinito porque a expressão booleana de teste é simplesmente True. Entretanto, quando zero é fornecido, o teste do if se torna verdadeiro e break é acionado, quebrando o laço. Observe que os incrementos de s e n não são mais protegidos por um corpo de decisão simplesmente porque não serão mais executados após a quebra do laço. O efeito deste laço é o mesmo daquele na Listagem-1.5.

Break tem efeito apenas sobre o laço em que está embutido. Numa situação onde laços estão aninhados, break afeta o laço que o executa. Por exemplo em,

```
while <expressao_1> do begin
   ...
   while <expressao_2> do begin
      ...
      if <expressao_3> then break;
   end;
end;
```

break quebra o segundo while (se <expressao_3> é verdadeira). Podem haver várias quebras, dependendo do total de iterações do primeiro while. Já no caso seguinte,

```
while <expressao_1> do begin
   ...
   while <expressao_2> do begin
      ...
   end;
   if <expressao_3> then break;
end;
```

a execução de break quebra o primeiro while e impede novas iterações do segundo while.

Há situações em que é necessário executar parcialmente o código de um bloco de um laço e passar para a iteração seguinte (sem quebra). Pascal possui uma instrução para isso de palavra reservada continue. Neste exemplo,

```
for i := 1 to 100 do begin
   if (i mod 2 = 0) continue;
   write(i, ' ');
end;
```

apenas os números ímpares, entre 1 e 100, são impressos. Isso acontece porque quando o teste i mod 2 = 0 é verdadeiro, continue é executado obrigando o laço a ignorar as instruções restantes do bloco (neste caso apenas o write) e passar para a iteração seguinte. A expressão i mod 2, que é o resto de divisão por 2 do valor em i, só retorna zero quando i contém um valor par o que justifica a impressão apenas de valores ímpares.

4.8.5. Desvio Incondicional

O *desvio incondicional* é um artifício que permite ao programador desviar o fluxo de execução para outras partes do programa. Este artifício não tem uso recomendado sequer existindo em outras linguagens. Pascal constrói desvio incondicional com uma instrução de palavra reservada goto. A sintaxe de goto é,

```
goto <rotulo>:
```

onde <rotulo> é um identificador declarado pelo programador numa *sessão de rótulos* e repetido no ponto do programa para onde o fluxo de execução deverá ser desviado. Esquematicamente,

```
label <rotulo>; // sessão de rótulos
begin
   <rotulo>:            // destino do desvio
   ...
   goto <rotulo>;
end.
```

A palavra reservada label inicia uma sessão de rótulos em Pascal. Declarar um rótulo significa simplesmente definir para ele um identificador nesta sessão. Se mais de rótulo existir, eles deverão ser separados por vírgulas. O local do programa para onde o desvio deve conduzir deve conter o rótulo pós-fixado de dois pontos (:).

Listagem 6

```
01  program desvio;
02  var
03     n: integer;
04  label
05     rot;
06  begin
07     n := 1;
08     rot:
09     write(n, ' ');
10     n := n+1;
11     if n<100 then goto rot;
12  end.
```

Na Listagem 6 o rótulo para desvio incondicional, rot, é declarado na Linha-05 e utilizado na Linha-08. O goto na Linha-11 provoca o desvio que, neste caso, está condicionado pelo if na mesma linha. O efeito da associação destes elementos é um laço: a variável n funciona como contador sendo configurada como 1 antes do rótulo (Linha-07) e incrementada da unidade (Linha-10) depois dele; além disso if provoca desvio para rot enquanto o valor em n for menor que 100. o resultado final é a impressão dos valores entre 1 e 99 (pelas sucessivas chamadas a write na Linha-09).

4.9. Funções e Procedimentos

Módulos são normalmente distribuídos em coleções chamadas *unidades*. Módulos de uma unidade têm sintaxe e funcionalidade de um programa e por essa razão são também denominados de *subprogramas*. Subprogramas podem ser implementados em arquivos de unidades (veja Capítulo-5 para mais detalhes sobre construção de unidades) ou no próprio arquivo principal do programa. Módulos codificados no arquivo principal do programa são disponíveis apenas para este programa. Cada módulo Pascal possui seu próprio nome e corpo de código que deve ser disposto antes do bloco principal e logo após as sessões de declaração (apenas aquelas que os módulos precisam).

Existem dois tipos de módulos em Pascal: *procedimentos* e *funções*. Tanto um quanto o outro podem requerer por informação adicional no momento em que são requisitados, mas apenas as funções devolvem (obrigatoriamente) um resultado tangível (ou seja, que pode ser armazenado numa variável). Toda informação adicional requerida por um módulo é recebida através de variáveis auxiliares denominadas de *argumentos* ou *parâmetros* do módulo.

Um módulo (procedimento ou função) pode contar com zero ou mais argumentos. A requisição de um módulo é denominada de *chamada do módulo* e o resultado de uma função de *retorno da função*. A chamada de um módulo é construída utilizando o nome do módulo pós-fixando os argumentos (quando existentes) entre parênteses e separados por vírgulas. Se não há argumentos, apenas o nome do módulo efetua sua chamada. Se uma função é chamada ela deve estar no lado direito de uma atribuição ou fazer parte de uma expressão.

O exemplo,

```
teste(23, x);
```

constrói uma chamada ao módulo teste que requer dois argumentos de entrada. Já em,

```
teste;
```

a chamada não requer argumentos e assim parênteses não são necessários. O módulo teste nesses exemplos pode ser tanto um procedimento quanto uma função. Caso seja função, o retorno é *perdido* por ausência de um agente que resgate o retorno. Um exemplo de resgate de retorno de uma função pode ser escrito assim,

```
y := teste(23, x);
```

ou integrada numa expressão como,

```
y := 23*teste*(23, x) + h*4.2;
```

onde y atua como resgatador direto no primeiro caso e indireto no segundo. Caso teste seja um procedimento nestes últimos exemplos, o compilador acusará erro de sintaxe. Vale enfatizar que todas essas construções devem encerrar em ponto-e-vírgula.

Para construir um procedimento sem argumentos em Pascal deve-se utilizar a seguinte sintaxe,

```
procedure <identificador_do_nome>;
{ declarações }
begin
{ corpo do procedimento }
end;
```

Procedimentos em Pascal devem iniciar com a palavra chave procedure seguida de um identificador (nome do módulo). A linha contendo a palavra reservada procedure é chamada de *protótipo* ou *interface* do procedimento. Sessões de declaração são suportadas, mas os recursos declarados são de uso único do procedimento. O bloco begin/end, restringe o corpo de código do procedimento e deve ser encerrado em ponto-e-vírgula. O programa da Listagem 7 ilustra o uso de procedimentos Pascal sem argumentos.

Listagem 7

```
01  program usando_procedimento;
02     procedure meu_teste;
03     begin
04        writeln('chamada feita com sucesso');
05     end;
06  begin
07     meu_teste;
08  end;
```

Na Listagem 7 o procedimento meu_teste compreende as Linhas 02 a 05 e sua chamada implica a impressão de uma mensagem fixa (Linha-04). O corpo principal do programa é constituído unicamente por uma chamada ao procedimento meu_teste (Linha-07) e dessa forma a saída do programa deverá ser,

```
chamada feita com sucesso
```

Quando o procedimento possui argumentos a sintaxe possui aspecto,

```
procedure <identificador_do_nome>(<grupo_1>;
<grupo_2>;...;<grupo_n>);
{ declarações }
begin
{ corpo do procedimento }
end;
```

Similar a uma sessão de declaração de variáveis, o par de parênteses pós-fixo ao nome do procedimento encapsula grupos de variáveis (argumentos ou parâmetros) separados por ponto-e-vírgula. Um *grupo de argumentos* é uma listagem de variáveis, de mesmo tipo separadas por vírgulas (quando mais de uma está presente) e encerrada com :<*tipo_do_grupo*>.

Listagem 8

```
01 program teste_de_procedimentos;
02 var
03    n: integer;
04 { módulos }
05 procedure print(n: integer);
06 begin
07    writeln('O quadrado de ', n, ' vale ', n*n);
08 end;
09
10 procedure listar(m, n: integer);
11 var i: integer;
12 begin
13    for i := m to n do
14       print(i);
15 end;
16 { programa principal }
17 begin
18    readln(n);
19    listar(0, n);
20    listar(2*n, 3*n);
21 end.
```

O programa da Listagem 8 trás dois procedimentos, print e listar, cujos protótipos estão respectivamente nas Linhas 05 e 10. O procedimento print imprime uma mensagem formatada baseada no valor do argumento n. O procedimento listar usa um laço para imprimir várias linhas de texto formatadas impressas por chamadas a print.

A chamada de um módulo a partir de outro é legal em Pascal desde que o módulo chamado seja definido antes do módulo chamador. O procedimento listar possui dois argumentos que recebem os valores dos limites inferior e superior do laço de impressão (Linhas 13 e 14). Por fim o programa principal faz duas chamadas independentes a listar (Linhas 19 e 20) cujos valores de repasse aos argumentos são calculados com base do valor de n lido na Linha-18. Um exemplo de saída deste programa é,

```
3

O quadrado de 0 vale 0

O quadrado de 1 vale 1

O quadrado de 2 vale 4

O quadrado de 3 vale 9

O quadrado de 6 vale 36

O quadrado de 7 vale 49

O quadrado de 8 vale 64

O quadrado de 9 vale 81
```

O mecanismo de *passagem de argumentos* funciona da seguinte forma: quando um módulo é chamado ele deve receber um total de entradas (separadas por vírgulas) igual ao número de argumentos presentes em seu protótipo. Essas entradas podem ser valores constantes, variáveis, expressões ou mesmo chamadas de outros módulos. Cada entrada é resolvida antes da execução do módulo e gera um valor que é atribuído ao argumento correspondente no módulo. Esse tipo de atribuição é denominada *implícita* por não utilizar o operador de atribuição (:=). Assim, no programa da Listagem-1.8, Linha-20, as expressões 2*n e 3*n são avaliadas e seus valores atribuídos respectivamente aos parâmetros m e n do procedimento listar.

Para construir uma função em Pascal deve-se utilizar a seguinte sintaxe,

```
function <ident_do_nome> (<grupo_1>,<grupo_2>,…,
<grupo_n>): <tipo_de_retorno>;
{ declarações }
begin
{ corpo da função }
end;
```

Funções em Pascal devem iniciar com a palavra chave function seguida de um identificador (nome do módulo). O protótipo de uma função inclui ainda o tipo da informação que a função retorna (*tipo de retorno*) marcado com dois-pontos seguido da palavra reservada do tipo. A estrutura de sessões de declaração e de código do corpo de uma função são idênticos aos dos procedimentos. As funções ainda contam com uma variável interna, cujo nome é o mesmo da função e o tipo é o de retorno, e cujo valor armazenado antes do término da função é justamente o valor de retorno desta função.

O programa da Listagem 9 possui protótipos de uma função (fat) na Linha-05 e de um procedimento (listar) na Linha-13. A função fat calcula o fatorial do valor recebido pelo argumento n. A terminação :integer no protótipo indica que a função retorna um valor inteiro. Para calcular o fatorial, a variável interna fat recebe inicialmente, na Linha-08, 1 (elemento neutro da multiplicação). O laço da Linha-09 evolui o valor em fat por sucessivas multiplicações até que esta variável contenha o valor do fatorial do valor em n.

Quando a função encerra, o último valor em fat é o do fatorial procurado e é justamente esse o valor retornado. O procedimento listar faz diversas chamadas à função fat (através do laço na Linha-16) para imprimir uma lista dos fatoriais entre zero e o valor no argumento de entrada max. A chamada à função fat é realizada diretamente dentro do writeln que recebe e imprime o valor do fatorial. Apesar de não ser necessário neste contexto, uma variável intermediária poderia receber o valor da chamada deixando explicito que *alguém* está de fato manipulado a saída da função. Neste caso a função listar se reescreveria assim,

```
procedure listar(max: integer);
var i, res: integer;
begin
   for i := 0 to max do begin
      res := fat(i);
      writeln('Fatorial de ', i, ' = ', res);
   end;
end;
```

onde res é a variável intermediária. Um exemplo de saída do programa é,

```
5
Fatorial de 0 = 1
Fatorial de 1 = 1
Fatorial de 2 = 2
Fatorial de 3 = 6
Fatorial de 4 = 24
Fatorial de 5 = 120
```

Listagem 9

```
01  program teste_de_funcoes;
02  var
03     n: integer;
04
05  function fat(n: integer): integer;
06  var i: integer;
07  begin
08     fat := 1;
09     for i := 1 to n do
10        fat := fat*i;
11  end;
12
13  procedure listar(max: integer);
14  var i: integer;
15  begin
16     for i := 0 to max do
17        writeln('Fatorial de ', i, ' = ', fat(i));
18  end;
19
20  begin
21     readln(n);
22     listar(n);
23  end.
```

Funções sem argumento, assim como procedimentos, são possíveis em Pascal. O protótipo,

```
function teste: integer;
```

pertence a uma função que não recebe parâmetros e retorna um valor inteiro. Esse tipo de função, devido a ausência de argumentos, aparentemente retorna sempre o mesmo valor. Entretanto elas podem usar outras informações do próprio programa ou mesmo do sistema para gerar diferentes saídas.

Variáveis declaradas em módulos, sejam eles procedimentos ou funções, são denominadas de *variáveis locais*. São variáveis locais tanto os argumentos de entrada quanto aquelas declaradas na sessão interna de declaração do módulo. Na Listagem 9 o procedimento listar possui duas variáveis locais, max e i, ao passo que a função fat possui três, n, i e fat. Variáveis locais são de uso exclusivo do módulo que as declara e por essa razão o mesmo nome pode ser utilizado para batizar variáveis que estão em módulos diferentes.

Variáveis declaradas na sessão de declarações do programa são denominadas de *variáveis globais*. Elas podem ser utilizadas por todo o programa incluindo os módulos e o programa principal. A restrição a essa globalidade é a posição da sessão de declaração do programa que pode inclusive ficar entre dois módulos, como em,

```
function teste_1(x: integer): real;
begin
   teste_1 := x*x;
end;

var
   y: real;

procedure teste_2;
begin
   writeln('olá');
end;
```

Em casos como esse a variável global (neste exemplo, y) só é visível nos módulos que sucedem a declaração e no programa principal. Assim teste_1 não visualiza y, apenas teste_2.

Quando uma variável local possui o mesmo nome de uma variável global (como n na Listagem 9), o módulo ignora a variável global e qualquer operação afeta apenas a variável local.

As regras que definem o acesso das variáveis num programa, brevemente discutidas nesta sessão, são denominadas de *regras de escopo*. Para entender mais sobre procedimentos, funções em Pascal, veja Capítulo 5.

5. Estudos de Caso

Os estudos de casos a seguir aplicam a sintaxe básica da linguagem Pascal estudada neste capítulo.

5.1. Conversão Celsius em Fahrenheit

A expressão que relaciona temperaturas em Celsius (ºC) e Fahrenheit (ºF) é a dada por:

$$\frac{C}{5} = \frac{(F - 32)}{9}$$

onde C é a temperatura em Celsius e F em Fahrenheit. Isolando F é fácil construir um programa de conversão de Celsius para Fahrenheit como se vê na Listagem 10.

Listagem 10

```
01  program Celsius_para_Fahrenheit;
02  var
03     C, F: real;
04  begin
05     write('Valor de temperatura em Celsius: ');
06     read(C);
07     F := C*9/5 + 32;
08     writeln(C:0:1, 'oC = ', F:0:1, 'oF');
09  end.
```

Na Linha-07, vê-se F isolado do lado esquerdo da atribuição e uma expressão sem parênteses do lado direito. O primeiro e o terceiro argumento do writeln, Linha-08, fazem as temperaturas serem escritas com uma casa decimal e o mínimo de caracteres (o zero em :0:1 causa este efeito). A saída final do programa após o usuário fornecer uma temperatura de 27^{0}C é a seguinte:

```
Valor de temperatura em Celsius: 27

27.0oC = 80.6oF
```

5.2. Análise do IMC

O IMC, ou *índice de massa corpórea*, de uma pessoa, é determinado pela equação:

$$IMC = \frac{p}{h^2}$$

onde *p* é o peso (em quilogramas) e *h* a altura (em metros). Quando o IMC é menor que 25, diz-se que o indivíduo está magro; se o valor é maior que 25, diz-se que está gordo; por fim um IMC no intervalo fechado de 20 a 25 acusa peso ideal. O programa da Listagem-1.11 resolve o problema do cálculo e análise do IMC.

Listagem 11

```
01  program Calculo_IMC;
02  var
03     peso, altura, imc: real;
04  begin
05     write('Fornecer altura em metros: ');
06     readln(altura);
07     write('Fornecer peso em kg: ');
08     readln(peso);
09     imc := peso/(altura*altura);
10      write('IMC = ', imc:0:1, '. ');
11     if (imc<20) then
12        writeln('Magro!')
13     else
14        if (imc>25) then
15           writeln('Gordo!')
16        else
17           writeln('Normal!');
18  end.
```

No programa da Listagem 11 a decisão na Linha-11 divide o processo entre as pessoas que são magras (imc é menor que 20) e as que não são (imc maior ou igual a 20). Quando a entrada não é de uma pessoa magra, o bloco do else (Linha-13) entra em atividade e dispara a decisão bidirecional na Linha-14. Este novo processo se divide entre os gordos (sucesso do teste da Linha-14) e os não gordos. Como a primeira condição imposta foi a de não magros, então a impressão da Linha-17 ocorre quando simultaneamente nem imc é menor que 20 e nem maior que 25.

Um exemplo de saída do programa da Listagem 11 É mostrado a seguir:

```
Fornecer altura em metros: 1.85
Fornecer peso em kg: 90
IMC = 26.3. Gordo!
```

5.3. Maior de Três Valores

Listagem 12

```
01  program Maior_De_Tres;
02  var
03     a,b,c: integer;
04  begin
05     write('Fornecer numeros: ');
06     readln(a,b,c);
07     write('Max(',a,',',b,',',c,') = ');
08     if a>b then
09        if (a>c) then
10        writeln(a)
11        else
12           writeln(c)
13     else
14        if b>c then
15           writeln(b)
16        else
17           writeln(c);
18  end.
```

A Listagem 12 resolve o problema de determinação do maior de três números inteiros fornecidos. Este programa possui três processos bidirecionais de decisão estando os dois últimos aninhados no primeiro. O if..then.. else do primeiro processo está nas Linha 08 e 13 dividindo o processo de decisão entre a situação em que o valor em a é maior que o em b e a situação em que o valor em a é menor ou igual ao em b. Quando a primeira situação ocorre o segundo processo de decisão (entre as Linhas 09 e 12) é acionado. Do contrário o terceiro processo de decisão (Linhas 14 a 17) tomará lugar. Apenas uma entre quatro chamadas de impressão (Linhas 10, 12, 15 ou 17) ocorrerá de fato. Um exemplo de saída do programa da Listagem-1.12 é o seguinte:

```
Fornecer numeros: 5 11 3

Max(5,11,3) = 11
```

5.4. Ordenação de Três Números

Seja o problema de escrever em ordem crescente três valores inteiros dados. Uma solução é utilizar variáveis para armazenar os valores dados e depois efetuar *trocas* convenientes de forma que os valores nelas fiquem em ordem crescente (na ordem de apresentação das variáveis). Uma troca de valores entre duas variáveis envolve necessariamente uma terceira variável (*auxiliar*) e ocorre em três fases: (i) a variável auxiliar copia o conteúdo da primeira variável; (ii) a primeira variável copia o conteúdo da segunda variável; (iii) a segunda variável copia o conteúdo da variável auxiliar. Sem variável auxiliar perder-se-ia o valor de uma das variáveis de entrada. O programa da Listagem 13 resolve o problema da ordenação de três valores inteiros.

Listagem 13

```
01  program Ordenacao_Tres_Numeros;
02  var
03     a, b, c, t: integer;
04  begin
05     write('Fornecer numeros: ');
06     readln(a,b,c);
07     if a>b then begin
08        t := a;
09        a := b;
10        b := t;
11    end;
12     if a>c then begin
13        t := a;
14        a := c;
15        c := t;
16     end;
17     if b>c then begin
18        t := b;
19        b := c;
20        c := t;
21     end;
22     writeln('Ordenacao: ',a,', ',b, ', ',c);
23  end.
```

No programa da Listagem 13 possui três decisões *não* aninhadas. Cada uma delas representa uma possível troca entre duas variáveis. Por exemplo, a decisão na Linha-07, quando obtém êxito, troca o conteúdo das variáveis a e b. A variável auxiliar é t e o bloco com begin-end é necessário porque três linhas de comando estão presentes. O algoritmo por trás das trocas é explicado a seguir.

Para ordenar a sequencia deve-se primeiramente fazer a conter o menor dos três valores. Para isso esta variável deve ser comparada com b (Linha-07) e c (Linha -12) e, possivelmente, trocada com elas (trocas das Linhas 08 a 10 e Linhas 13 a 15). Uma vez que o menor valor agora está em a, o valor intermediário estará em b ou c, impasse facilmente resolvido por um único teste, neste caso, o da Linha-16. Finalizado os três testes, e feitas as devidas trocas (nenhuma troca pode ocorrer!), a, b e c conterão nesta ordem os valores de entrada em ordem crescente. Abaixo ilustra-se um exemplo de saída do programa da Listagem 13,

```
Fornecer numeros: 32 11 9

Ordenacao: 9, 11, 32
```

5.5. Somando Quadrados

Seja o problema de somar o quadrado dos inteiros entre 0 e um limite superior dado. O programa da Listagem 14 sugere uma solução.

Listagem 14

```
01  program Somatorio_Quadratico;
02  var
03     i, max, S: integer;
04  begin
05     S := 0;
06     write('Fornecer limite superior: ');
07     readln(max);
08     for i := 0 to max do
09        S := S + i*i;
10     writeln('Soma de quadrados entre 0 e ', max, ' = ', S);
11  end.
```

Na Linha-05 a variável S, que conterá ao final do processamento o somatório desejado, recebe zero marcando o início do somatório. O laço da Linha-08 tem como limite superior a própria variável max indicando que o somatório vai de fato até o valor lido na Linha-07. A expressão do lado direito da atribuição na Linha-09 contém a própria variável que sofrerá a atribuição (S).

Como um processo de atribuição processa da direita para a esquerda então o valor da expressão é resolvido antes da atribuição propriamente dita desassociando os papéis que S desempenha em cada lado. A rigor o valor em S é atualizado com um novo valor calculado a partir de seu anterior. Como se trata de um laço, então S será atualizada várias vezes. Em cada iteração o quadrado do valor em i (i*i) é adicionado ao valor corrente de S e este novo valor atribuído a S causando um efeito de acúmulo. No final o valor impresso na Linha-10 será a somatória quadrática entre zero e o valor em max. Um exemplo de saída do programa é mostrado a seguir,

```
Fornecer limite superior: 100
Soma de quadrados entre 0 e 100 = 10670
```

5.6. Somatório Condicional

Seja o problema de somar todos os números inteiros positivos menores do que 500 que sejam divisíveis por 7 mas que não sejam divisíveis por 3. O programa da Listagem 15 sugere uma solução.

Listagem 15

```
01  program Somatorio_Condicional;
02  var
03     n, S: integer;
04  begin
05     S := 0;
06     for n := 0 to 500 do
07           if (n mod 7 = 0) and (n mod 3 <> 0) then
08              S := S + n;
09        writeln('Soma = ', S);
10  end.
```

A variável S é utilizada para armazenar a soma de inteiros pretendidos. Por essa razão ela é previamente configurada como zero (Linha-05) indicando que nada ainda foi somado a ela. O laço na Linha-06, associado a decisão na Linha-07, condiciona a atribuição na Linha-08. A expressão S := S + n representa dois estágios de S, um antes (lado direito da atribuição) que soma seu valor na iteração anterior ao valor em n; e um depois (lado esquerdo da atribuição) que sobrepõe em S o novo valor obtido da expressão.

O efeito destes acoplamentos é um contínuo acúmulo em S dos valores em n que suprem o teste na Linha-07. Este teste condiciona que o resto da divisão por 7 deve ser nula (ou seja, divisível por sete) e simultaneamente que o resto da divisão por 3 não deve ser nula (ou seja, não divisível por 3). A saída do programa da Listagem 15 é,

```
Soma = 12096
```

5.7. Números Primos

Números primos são números naturais que são divisíveis apenas por um e por si mesmos (o menor número primo é 2). Para testar se um natural *n* é primo, deve-se dividi-lo pelos naturais no intervalo |2;Ψ|, onde $\Psi^2 \leq n$. Se todas as divisões forem inexatas (não divisibilidade), *n* será primo. Se entretanto, neste processo de divisões, uma delas for exata, o restante do intervalo (se ainda houver) deverá ser ignorado e *n* considerado não primo.

Listagem 16

```
01  program primos;
02
03  function e_primo(n: integer): boolean;
04  var d: integer;
05      res: boolean;
06  begin
07      d := 2;
08      res := true;
09      while d*d <= n do begin
10          if (n mod d) = 0 then begin
11              res := false;
12              break;
13          end;
14          d := d + 1;
15      end;
16      e_primo := res;
17  end;
18
19  var i, cnt: integer;
20  begin
21      i := 2;
22      cnt := 0;
23      while  cnt < 20 do begin
24          if e_primo(i) then begin
25              write(i, ' ');
26              cnt := cnt+1;
27          end;
28          i := i+1;
29      end;
30  end.
```

A função e_primo na Listagem 16 recebe um inteiro e determina se ele é ou não primo. Para tanto o retorno da função é um booleano (true indica que n contém um primo e false um não primo). A função e_primo funciona conforme explicado a seguir. O laço da Linha-09 faz o valor em n ser dividido por todos os valores do intervalo de teste (veja parágrafo anterior).

O controle deste intervalo é feito com a variável d: o limite inferior (menor primo) é definido na Linha-07 e o superior testado pela expressão booleana do laço (a incrementação é feita na Linha-14). A variável res controla o estado (primo ou não primo) do valor em n e dessa forma, como primeira hipótese, recebe true (*um número é primo até que prove o contrário*).

O teste na Linha-10 verifica a divisibilidade do valor em n pelo valor corrente em d (numa dada iteração de while). Quando o valor em n é primo, todos estes testes falham, o laço principal se desenvolve normalmente e o valor retornado (Linha-16) é true (como esperado). Se um destes testes ob-

tém sucesso, significa que o valor em n não é primo e desta forma o laço não precisa mais continuar. Por essa razão res é redefinido para false (Linha-11) e o laço quebrado (Linha-12).

O programa principal na Listagem 16 utiliza a função e_primo para imprimir na saída padrão os primeiros 20 números primos. A variável i inicia com valor 2 (linha-21) e é incrementado continuamente na Linha-28 por ação do laço while (Linha-23). Cada vez que o valor em i é primo (teste da Linha-24) seu valor é impresso (Linha-25) e o contador de primos, cnt (previamente anulado na Linha-22), incrementado de uma unidade (Linha-26). Como o teste de while é baseado em cnt, o efeito final será a impressão de uma quantidade fixa de primos (20, neste exemplo). A saída do programa é,

```
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71
```

Atividades de avaliação

1. Faça um programa para o computador calcular o quadrado de um número.
2. Faça um programa para o computador determinar o valor máximo entre dois números inteiros A e B.
3. Faça o programa para calcular a média das notas de uma turma de 20 alunos.
4. Faça um programa que receba as dimensões de uma caixa d'água e calcule o seu volume.
5. Modifique o programa da questão 04 de forma que a saída seja a quantidade em metros quadrados de material necessário a construção da caixa. Considere a presença de tampa.
6. Dois alunos fizeram três provas cada um cujas notas são conhecidas Faça um programa que diga qual deles tem a maior média.
7. Construa um programa que receba dois valores inteiros **a** e **b** e retorne como saída $\mathbf{a}^{b}$.
8. Construa um programa que receba os lados de um triângulo e indique se ele existe ou não. Em caso afirmativo imprimir também: a) se ele é retângulo, acutângulo ou obtusângulo; b) valor de sua área.
9. Faça um programa para calcular o fatorial de um número N inteiro dado.

10. Faça um programa que determine se um número é divisível ou não por outro.

11. Faça um programa para calcular a soma de todos os números inteiros menores que 500 e que não sejam divisíveis por 3 e terminem no dígito 2.

12. Considere o problema do polinômio de segundo grau. Construa programa que receba como entrada os três coeficientes destes polinômios, exiba suas raízes se existirem, senão informe isso como saída.

13. Considere uma moeda que contenha cédulas de 50,00, 10,00, 5,00 e 1,00. Considere ainda que caixas eletrônicos de um banco operem com todos os tipos de notas disponíveis, mantendo um estoque de cédulas para cada valor. Os clientes deste banco utilizam os caixas eletrônicos para efetuar retiradas de uma certa quantia. Escreva um programa que, dado o valor da retirada desejada pelo cliente, determine o número de cada uma das notas necessárias para totalizar esse valor, de modo a minimizar a quantidade de cédulas entregues.

14. Construir programa que receba os valores naturais **a** e **M** e tenha como saída o valor da seguinte somatória:

Capítulo 2

Strings e Arrays

1. Caracteres e Strings

A maior parte de toda informação gravada em mídias digitais, incluindo aquela disponível via internet, está em formato texto. Textos, em computadores, são representados por *strings* que por sua vez são representadas pela união sequencial (*concatenação*) de unidades gráficas de informação denominadas *caracteres*. A presença de recursos de manipulação de strings numa linguagem de programação é essencial. Tais recursos são normalmente disponibilizados através de bibliotecas ou de comandos nativos (previstos no projeto original da linguagem e implementados juntamente com ela). Pascal conta com recursos nativos para a manipulação de texto.

1.1. Caracteres

Pascal possui o tipo de dados primitivo char para representar caracteres independentes (que não fazem parte de uma string). Assim,

```
var
    ch: char;
```

declara um caractere em Pascal. Em Pascal os caracteres utilizam 1-byte e seguem a codificação ASCII. Em ASCII os caracteres são representados por sete bits e são baseados no alfabeto inglês (o oitavo bit é não representativo). O total de caracteres é 128 sendo 33 não imprimíveis (como o caso dos caracteres de controle utilizados nos bastidores de softwares processadores de texto).

ASCII: abreviação de *American Standard Code for Information Interchange* que significa Código Padrão Americano para o Intercâmbio de Informação.

cod	ASCII	cod	ASCII	cod	ASCII	cod	ASCII	cod	ASCII
32		42	*	52	4	62	>	72	H
33	!	43	0	53	5	63	?	73	I
34	“	44	0	54	6	64	@	74	J
35	#	45	0	55	7	65	A	75	K
36	$	46	.	56	8	66	B	76	L
37	%	47	/	57	9	67	C	77	M
38	&	48	0	58	:	68	D	78	N
39	‘	49	1	59	;	69	E	79	O
40	(	50	2	60	<	70	F	80	P
41	)	51	3	61	=	71	G	81	Q

cod	ASCII	cod	ASCII	cod	ASCII	cod	ASCII	cod	ASCII
82	R	92	\	102	f	112	p	122	z
83	S	93	]	103	g	113	q	123	{
84	T	94	^	104	h	114	r	124	|
85	U	95	_	105	i	115	s	125	}
86	V	96	`	106	j	116	t	126	~
87	W	97	a	107	k	117	u		
88	X	98	b	108	l	118	v		
89	Y	99	c	109	m	119	w		
90	Z	100	d	110	n	120	x		
91	[	101	e	111	o	121	y		

Tabela 9 - Tabela de código ASCII

Uma constante caractere deve ser especificada em Pascal entre aspas simples, como nos exemplos,

```
'a'     'A'     '$'     '7'     '_'     ' '
```

Conforme padrão ASCII, há distinção entre caracteres maiúsculos e minúsculos, no caso de letras, e *não* há correspondência entre caracteres numéricos e constantes numéricas (assim, por exemplo, '7' é diferente de 7).

Cada caractere ASCII possui um código (valor inteiro) exclusivo denominado *código ASCII* do respectivo caractere. A Tabela 9 Ilustra os códigos ASCII dos caracteres imprimíveis cuja faixa é 32 a 126 (128 possíveis com 7-bits menos os 33 não imprimíveis).

Um caractere Pascal pode ser representado em código também pela prefixação do símbolo # ao código ASCII. Assim a expressão #65 é equivalente a 'A'. Quando o caractere não é imprimível, essa notação torna-se uma opção à representação em código do caractere. Alguns importantes caracteres não

imprimíveis possuem larga aplicação em programação possuindo inclusive teclas de acionamento próprias (no teclado do computador). São exemplos,

```
#10     ENTER   // concatenado a #13 em alguns sistemas

#27     ESC

#9      TAB

#8      BACKSPACE
```

Para permutar entre representação numérica e gráfica, os caracteres em Pascal contam com os comandos nativos ord e chr. O comando ord recebe um caractere e retorna seu código ASCII ao passo que chr recebe um código ASCII e retorna o caractere equivalente. Assim, por exemplo, ord('A') retorna 65 e chr(65) retorna 'A'.

Caracteres podem ser impressos por write/writeln desde que sejam imprimíveis. Caracteres não imprimíveis causam outros efeitos (não necessariamente na saída padrão) que dependerão da plataforma utilizada. São exemplos de impressão de caracteres imprimíveis,

```
write('A');                         // A

writeln('B', '%');                  // B%

write(#65, ord('X'), chr(35));      // A98#
```

O resultado de impressão de um caractere imprimível, seja fornecido com a notação de aspas simples, seja com a notação de # (ou do comando chr), é sempre simbólica. A impressão da saída de uma chamada a ord é sempre um valor inteiro (valor do código ASCII).

Caracteres são dados ordinais dispostos na sequência de seus respectivos códigos ASCII. Dessa forma podem ser comparados com operadores relacionais ou utilizados como contadores em laços for. Para construir, por exemplo, uma tabela de letras minúsculas e seus respectivos códigos ASCII, implementa-se,

```
var

    ch: char;

begin

    for ch := 'a' to 'z' do

        writeln(ch, ':  ', ord(ch));

end.
```

Aritmética envolvendo caracteres pode ser executada indiretamente pelo uso de seus respectivos códigos ASCII. Um exemplo interessante é a conversão de um caractere letra minúscula para maiúscula. A ideia é subtrair o código de um dado caractere letra minuscula pelo código do caractere 'a'

(107) e somar resultado ao do código do caractere 'A' (75). Como as letras, tanto maiúsculas como minúsculas, são sequencias contínuas (veja Tabela-2.1), este procedimento deve fazer retornar um código ASCII conveniente para a versão maiúscula. A função maiuscula na Listagem 17 recebe um caractere, verifica se é uma letra minuscula (Linha-03) e, caso seja, retorna sua versão maiúscula (calculada pela expressão na Linha-04), do contrário retorna o próprio caractere de entrada (Linha-06).

Listagem 17

```
01  function maiuscula(ch: char): char;
02  begin
03      if (ch>='a') and (ch<='z') then
04          maiuscula := chr( ord(ch) - 32 );   // -32 é 75-107
05      eles
06          maiuscula := ch;
07  end.
```

1.2. Strings Curtas e Longas

Uma string é uma concatenação de caracteres. Em Pascal a palavra reservada string denota o tipo para declaração de strings. Strings possuem especificação opcional de tamanho, ou seja, não é obrigatório informar quantos caracteres a string deverá conter. No exemplo a seguir uma string sem especificação de tamanho é declarada,

```
var
    st1: string;
```

Neste outro exemplo a declaração é feita com especificação de tamanho,

```
var
    st2: string[120];
```

O tamanho, quando especificado, é definido entre colchetes, [ ], e pós-fixado à palavra reservada string. Strings de tamanho definido em Pascal só suportam até 255 caracteres. Especificar um valor maior que esse causa um erro de compilação. A restrição de tamanho imposta por este tipo de construção classifica as strings declaradas como *strings curtas*. Tais strings podem ser alternativamente declaradas pela palavra reservada shortstring (sem especificação de tamanho). Exemplo,

```
var sst: shortstring;
```

As *strings longas* em Pascal correspondem às strings sem restrições de tamanho. Elas permitem a concatenação de qualquer quantidade de caracteres. São declaradas utilizando-se o tipo ansistring. Exemplo,

```
var lst: ansistring;
```

Uma string longa é processada mais lentamente que as curtas devido a natureza de sua implementação. Por uma questão de performance, sempre que possível, é aconselhável utilizar strings curtas.

O compilador Pascal poderá interpretar uma declaração de string sem especificação de tamanho como sendo uma string curta ou longa dependendo do acionamento da diretiva {$H}. A diretiva {$H-} instrui o compilador a considerar uma declaração de string sem definição de tamanho como uma string curta de tamanho 255 ao passo que {$H+} instrui tratar-se de uma string longa. Se nenhuma destas duas diretivas estiver presente o padrão {$H-} será aplicado. Logo existe equivalência tanto entre os quatro códigos,

Diretivas de Pré-processamento ou simplesmente Diretivas: tratam-se de códigos escritos entre chaves, { }, pré-fixados pelo símbolo $ e que servem normalmente para ligar, desligar ou modificar recursos do compilador. Exemplo, {$I+}.

```
program teste;
var
    x: string;
{…}
```

```
program teste;
{$H-}
var
    x: string;
{…}
```

```
program teste;
var
    x:
string[255];
{…}
```

```
program teste;
var
    x:
shortstring;
{…}
```

quanto nos dois que seguem,

```
program teste;
{$H+}
var
    x: string;
{…}
```

```
program teste;
var
    x: ansistring;
{…}
```

1.3. Operações com Strings

Em Pascal, atribuições podem ser diretamente realizadas em strings longas e curtas pelo operador de atribuição, :=. As atribuições a seguir são válidas,

```
var
    a: string;
    b: ansistring;
begin
    a := 'este e um teste';
    b := 'este e um novo teste';
end.
```

Constantes de tipo string são dispostas entre aspas simples, ‘’. No exemplo anterior o conteúdo de duas strings constantes são copiados para as variáveis string a e b. A primeira atribuição feita a uma variável string é conhecida como *inicialização da string*.

Variáveis de tipo string podem ser atribuídas diretamente a outras mesmo quando seus **comprimentos** são distintos. Quando uma string recebe o conteúdo de outra string, seu conteúdo é primeiramente eliminado da memória e em seguida sua referência (o ponto da memória ao qual a string se refere) é mudado para o mesmo da string atribuída. Por exemplo em,

O *tamanho* de uma string equivale ao máximo de caracteres que ela pode suportar. O **comprimento** de uma string representa o total de caracteres que ela utiliza em determinado momento. O comprimento é sempre menor ou igual ao tamanho.

```
A := ‘minha casa’;
B := ‘meu trabalho’;

B := A;
```

faz a string ‘meu trabalho’ ser eliminada da memória e a string ‘minha casa’ passar a ser referenciada tanto por A quanto por B. A priori acredita-se que qualquer mudança em A afetará B (e vice-versa) numa nova atribuição. Mas isso não acontece. Se algum código mais adiante tentar modificar A ou B (não apenas ler, mas modificar!), uma cópia da string será feita antes da modificação e atribuída a quem solicitou a mudança (A ou B). Novamente as strings possuirão referências distintas (uma mantida e outra modificada). Exemplo,

```
A := 'minha casa';              // A no endereço 1
B := 'meu trabalho';            // B no endereço 2
B := A;                 // A e B no endereço 1 e nada mais em 2

A := 'minha vida';              // B no endereço 1 e A no endereço 3
```

Os comandos write e writeln podem conter strings nas suas respectivas listas de impressão sendo possível tanto imprimir constantes strings quanto o conteúdo de variáveis strings. Exemplo,

```
fulano := ‘beltrano’;

writeln(‘Meu nome e ‘, fulano);
```

Os comandos read e readln podem ser usados para ler strings curtas e longas sem adicional de sintaxe como em,

```
read(fulano);
```

onde fulano é uma string (curta ou longa). Os argumentos string de read/readln precisam ser variáveis. Caso a string repassada *não caiba* na variável, ocorrerá um erro em tempo de execução.

A concatenação ou fusão de strings em Pascal é construída com o operador de concatenação, +, cujo símbolo é o mesmo da soma aritmética mas a semântica (comportamento) é diferente. Podem ser concatenadas constantes strings, strings em variáveis ou associações. Exemplos,

```
x := 'ola' + ' !';                    // ola!
y := x + 'mundo';                     // ola!mundo
z := x + y;                           // ola!ola!mundo
```

onde x, y e z são variáveis e podem ser tanto strings longas como curtas.

Concatenações geram novas strings. Por exemplo em,

```
x := x + 'fim';
```

onde x é uma string. Uma nova string é criada do lado direito da atribuição pela concatenação do conteúdo de x e 'fim'. Com a atribuição, a antiga string inicial em x é eliminada da memória e x passa a se referenciar à nova string oriunda da concatenação.

Strings podem ser também concatenadas com caracteres, como em,

```
x := 'Caractere de ASCII 38 e: ' + #38; // Caractere de ASCII 38 e: &
```

Strings não podem ser concatenadas a tipos numéricos diretamente. Para inserir numa string valores que estão armazenados em inteiros ou reais deve-se primeiramente converter os valores numéricos em strings e por fim efetuar a concatenação. A unidade system possui o comando str para a conversão de inteiros ou reais em strings. A sintaxe básica deste comando é,

```
str(V, S);
```

onde V denota o valor (inteiro ou real) para ser convertido e armazenado na string S (curta ou longa). A Listagem 18 ilustra o uso de str. As conversões nas Linhas 09 e 10 colocam respectivamente nas variáveis si e sp as versões string do inteiro em idade e do real em peso. Observe que é possível converter a parte do valor real que realmente interessa utilizando a notação de dois-pontos (:0:2 denota todas as casas antes da vírgula e apenas duas depois). A string final é gerada pela concatenação da Linha-11, armazenada na variável s e impressa na Linha-12.

Listagem 18

```
01  program usando_str;
02  var
03      s, si, sp: string;
04      idade: integer;
05      peso: real;
06  begin
07      idade := 25;
08      peso := 60.5;
09      str(idade, si);
10      str(peso:0:2, sp);
11      s := 'Ela tem '+si+' anos de idade e pesa '+sp+'
12  quilos';
13      writeln(s);
    end.
```

Números também podem ser convertidos em strings. O comando em system para essa tarefa é val cuja sintaxe básica é,

```
val(S, V, P)
```

onde S representa a string a ser convertida em número e V a variável numérica (inteira ou real) que receberá o resultado da conversão. A conversão de valor numérico em string é susceptível a erro e por essa razão existe um terceiro argumento de controle denotado pela variável inteira P. Se a conversão ocorre normalmente, o valor final de P será zero, do contrário conterá um valor maior que zero (posição na string S onde ocorreu o erro de conversão).

A Listagem 19 Ilustra o uso do comando val. O programa lê um valor da entrada padrão e imprime seu quadrado. Para assegurar que o valor digitado é realmente um inteiro criou-se a função inteiro_valido que possui um laço infinito (Linhas 08 a 16) quebrado na Linha-14 somente quando um inteiro valido é dado via entrada padrão. Em cada iteração deste laço a string entrada é usada para receber uma informação aleatória do teclado (Linha-10).

Na Linha-11, val tenta executar a conversão da string em entrada utilizando a variável interna inteiro_valido como receptáculo e err como controle de conversão. O teste da Linha-12 decide entre a impressão de uma mensagem de erro (err tem um valor diferente de zero) ou pela quebra do laço (err contém zero). Neste último caso o ultimo valor da variável interna inteiro_valido, antes da quebra do laço, é coerente. Como a função termina logo após o fim do laço e o último valor da variável interna inteiro_valido é o retorno da função, então x, no programa principal, recebe necessariamente um inteiro válido. Um exemplo de saída deste programa é,

```
Valor inteiro> 90u

inteiro invalido!

Valor inteiro> 7.8

inteiro invalido!

Valor inteiro> /

inteiro invalido!

Valor inteiro> 7

O quadrado vale 49
```

Listagem 19

```
01  program valor_seguro;
02
03  function inteiro_valido: integer;
04  var
05     entrada: string;
06     err: integer;
07  begin
08   repeat
09     write('Valor inteiro> ');
10     readln(entrada);
11     val(entrada, inteiro_valido, err);
12     if err<>0 then
13      writeln('inteiro invalido!')
14        else
15            break;
16    until False;             // Laço infinito
17  end;
18
19  var x: integer;
20  begin
21   x := inteiro_valido;
22   writeln('O quadrado vale ', x*x);
23  end.
```

Os caracteres de uma string podem ser acessados de forma individual utilizando o *operador de indexação*, [ ]. Este operador é pós-fixado à variável string e deve conter um inteiro (*índice*) com valor entre 1 e o comprimento da string. No exemplo,

```
s := 'abcdefg';
write(s[3], s[2]);                    // cb
ch := s[7];                           // g
```

onde s e ch são respectivamente uma string e um caractere, write imprime o terceiro e o segundo caracteres de s ao passo que ch copia o sétimo.

O comando length da unidade system recebe uma string como entrada e retorna seu comprimento. No caso de strings curtas com tamanho predefinido o comprimento consiste apenas dos caracteres gravados e não do valor definido na declaração. Exemplo,

```
var
    s: string[100];                    // string curta de tamanho predefinido
begin
    s := 'abcdefg';
    writeln( length(s) );              // 7
end.
```

Apesar de s ter comprimento predefinido 100, o valor impresso por writeln é 7. O comprimento predefinido representa a quantidade máxima de caracteres que a string pode suportar. É denominado *comprimento teto*. Já o valor retornado por length é denominado *comprimento utilizado*. O comando length funciona normalmente com cadeias longas e curtas com ou sem comprimento predefinido.

O programa da Listagem 20 ilustra o uso de length. Neste programa uma string, lida via entrada padrão, é reescrita de trás para a frente. O laço de contagem decrescente na Linha-09 varre todos os índices da string s de trás para frente (inicia em length(s) e vai até 1) e faz write na Linha-10 imprimir o caractere correspondente a posição visitada. O efeito é a impressão da string de entrada em ordem reversa. Um exemplo de saída do programa é,

```
Entrada> abcdefg
Inverso> gfedcba
```

Listagem 20

```
01  program de_tras_pra_frente;
02  var
03      s: string[100];
04      i: integer;
05  begin
06      write('Entrada> ');
07    readln(s);
08      write('Inverso> ');
09      for i := length(s) downto 1 do
10          write( s[i] );
11      writeln;
12  end.
```

O acesso individual a caracteres proporcionado pelo operador de indexação é leitura/escrita, ou seja, pode ser usado tanto para ler os caracteres quanto para modificá-los. Um exemplo interessante é a conversão de uma string em sua versão maiúscula. Utilizando a função maiuscula da Listagem 17, pode-se escrever,

```
read(s);
for i := 1 to length(s) do
    s[i] := maiuscula( s[i] );              // ver Listagem-2.1
writeln(s);
```

onde s pode ser uma string curta ou longa. O laço de contagem crescente varre a cadeia completa e modifica cada caractere para a saída da chamada da função maiuscula (que por sua vez recebe o caractere da cadeia de entrada).

Strings podem ser comparadas com operadores relacionais. O critério de comparação adotado pelo compilador Pascal é denominado *critério lexicográfico*. É exatamente o mesmo critério utilizado para fazer comparações e então ordenar uma lista, por exemplo, de nomes de pessoas. Dadas duas strings, a comparação lexicográfica executa comparações entre caracteres de uma e de outra na mesma posição.

As comparações partem do início de cada string e se mantêm enquanto os caracteres forem idênticos ou uma das cadeias não extrapolar seu índice máximo. Quando surge no processo dois caracteres distintos, a comparação termina e será lexicograficamente maior a string cujo caractere comparado tiver maior código ASCII. Exemplos,

```
'casar' > 'casado'              // TRUE pois ASCII de 'r' maior que o de 'd'

'MEDALHA' > 'MEDICO'            // FALSE pois ASCII de 'A' menor que o de 'I'

'mata' <= 'pata'                // TRUE pois ASCII de 'm' menor que o de 'p'

'cantar' <> 'cantora'           // TRUE pois encontrou 'a' diferente de 'o'
```

Quando a menor string é prefixo da maior então não existe caractere de desempate. Neste caso a de maior comprimento é considerada lexicograficamente maior. Exemplos,

```
'casa' > 'casado'               // FALSE pois a primeira é prefixo da segunda

'folhagem' > 'folha'            // TRUE pois a segunda é prefixo da primeira

'cantor' <= 'cantora'           // TRUE pois a primeira é prefixo da segunda
```

Duas strings são idênticas em Pascal deste que tenham mesmo comprimento e mesma sequencia de caracteres.

Para obter uma **substring** de uma dada string, utiliza-se o comando copy, unidade system, cuja sintaxe básica é,

```
copy(S, I, cont);
```

onde S denota uma string, curta ou longa, da qual se deseja obter a substring, I denota a posição dentro da string onde deve iniciar a substring e cont a quantidade de caracteres que devem ser extraídos. A nova string gerada por copy dve ser atribuída a uma variável string, fazer parte de uma expressão ou ser repassada a como argumento de um módulo. Exemplos,

Substring:
String formada por uma subsequencia contínua dos caracteres de uma outra string. Na string 'abcdefghij' são exemplos de substrings 'abc', 'cdefg' e 'ij'.

```
s := 'ABCDEFG';

t := copy(s, 3, 4);                    // CDEF

writeln( copy(t, 2, 2)  );             // DE

s := s + copy(s,4,4);          // ABCDEFGDEFG
```

onde s e t são variáveis strings. A primeira extração é atribuída a t, a segunda é repassada como parâmetro de writeln e a terceira é componente de uma expressão.

Quebrar uma string significa subdividi-la em substrings cuja concatenação origina a string original. Por exemplo, a string 'abcxyzw' pode ser quebrada no conjunto {'abc', 'xyzw'} ou em {'ab', 'cxy', 'yz', 'w'}.

Strings podem ser inseridas em outras, ou seja, é possível, utilizando um único comando, **quebrar uma string** W em W_1 e W_2 e depois fazer W se tornar $W_1 + X + W_2$ onde X é a string inserida. Para inserir uma string em outra, utiliza-se o comando insert, unidade system, seguindo a sintaxe básica,

```
insert(S1, S2, p);
```

onde S1 representa a string que será inserida e S2 a string que sofrerá a inserção. A posição de inserção em S2 é dada pelo inteiro p. No exemplo,

```
s := '123';

tex := 'abcdef';

insert(s, tex, 4);             // abc123def
```

a string s é inserida na string tex. A inserção quebra tex em 'abc' e 'def' e depois atribui a tex o resultado da concatenação 'abc' + '123' + 'def'. Insert não possui retorno (procedimento) mas modifica a string passada como segundo argumento.

Para remover caracteres de uma string utiliza-se o comando delete, unidade system, de sintaxe básica,

```
delete(S, p, cont);
```

onde S denota a string que sofrerá retração, p a posição onde deve principiar a deleção e cont a quantidade de caracteres que deverão ser eliminados. Internamente delete efetua a quebra da string e concatena as partes que devem ser mantidas. Exemplo,

```
s := 'abcdefgh';

delete(s, 3, 4);

writeln(s);                    //  abgh
```

onde s é uma string. A deleção ocorre a partir da posição 3 e elimina 4 caracteres. Delete quebra S nas substrings {'ab'. 'cdef'. 'gh' }, concatena a primeira com a terceira e atribui o resultado a S. O comando delete, assim como insert, não possui valor de retorno mas modifica a string que recebe como argumento.

2. Arrays

Uma *array* é uma estrutura de dados que permite o armazenamento sequencial de dados de mesmo tipo. Cada unidade constituinte de uma array é denominada *célula* e o tipo de dados comum a todas as células é denominado *tipo base*. Uma array é referenciada em código por uma única variável e suas células por uma associação desta variável, do operador de indexação, [ ], e de um ou mais valores de índices.

Strings são exemplos de arrays cujo tipo base é char mas comumente não recebem o título (de array) por possuírem propriedades próprias não presentes em arrays de outros tipos base. Manteremos a convenção distinguindo strings de arrays. Arrays não podem ser diretamente concatenadas, comparadas, impressas por write/writeln ou lidas por read/readln. Veremos alguns códigos que permitirão gerar comportamentos equivalentes em arrays.

O *comprimento* de uma array é igual ao total de células que a constitui e pode também ser determinado pelo comando length que receberá, neste caso, a variável array como argumento. Arrays não fazem distinção entre comprimento teto e comprimento utilizado como nas strings. Dessa forma length retorna sempre o total de células alocadas em memória para a array, independente do conteúdo.

Arrays em Pascal podem ser *estáticas* e *dinâmicas*. Uma array estática é aquela cujo comprimento é predefinido na declaração e não pode ser mais modificado em tempo de execução. Arrays dinâmicas são aquelas cujo comprimento pode ser modificado a qualquer momento da execução (tanto expandindo como contraindo).

As células de arrays, sejam estáticas ou dinâmicas, suportam leitura/escrita deste que o índice utilizado esteja na faixa apropriada. A faixa apropriada em arrays estáticas pode ser definida pelo programador tendo comumente como limite inferior o valor 1. O índice inferior de arrays dinâmicas é sempre zero.

Arrays podem possuir mais de uma *dimensão*. Uma array unidimensional representa, por exemplo, uma lista e necessita de um único índice para referenciar suas células. Uma array bidimensional representa, por exemplo, uma tabela ou uma matriz e precisa de dois índices. Uma array tridimensional utiliza três índices e assim por diante. Não há restrições em Pascal no número de dimensões que uma array pode possuir.

2.1. Arrays Unidimensionais

Para declarar uma array estática unidimensional em Pascal deve-se utilizar a seguinte sintaxe básica,

```
var
    <variavel> : array[<indice_inferior>..<indice_superior>] of <tipo_base>;
```

As palavras reservadas array e of significam *arranjo* e *de*. Os dois pontos sequenciais, .., separam os dois valores limites de índice sendo o primeiro o inferior e o segundo o superior. Colchetes, [], devem necessariamente conter a faixa de índices. O tipo base após of é também obrigatório. Para criar uma array de 100 valores inteiros pode-se escrever,

```
var
    x: array[1..100] of integer;
```

neste exemplo os índices limites são 1 e 100. Para alterar uma dada célula de x usa-se,

```
x[34] := 152;
```

neste caso a trigésima quarta célula recebe o valor 152. Para fazer x receber os 100 primeiros ímpares, pode-se escrever,

```
for i := 1 to length(x) do
    x[i] := 2*i - 1;;
```

onde i é um inteiro. Para programadores de linguagem C, acostumados com arrays de índice inicial zero, a declaração de x pode alternativamente ser,

```
var
    x: array[0..99] of integer;
```

Observe que essa alteração de índices mantem o comprimento 100 da array x de forma que para construir um laço de varredura completa deve-se fazer o limite inferior zero e o superior length(x)-1, como em,

```
for i := 0 to length(x)-1 do
    x[i] := 2*i + 1;            // o sinal deve mudar de - para +
```

Os comandos low e high, unidade system, retornam respectivamente os limites inferior e superior dos índices de uma array. Utilizando tais comandos o exemplo anterior pode ser reescrito para,

```
for i := low(x) to high(x) do
    x[i] := 2*( i-low(x) ) + 1;            // i-low(x) inicia sempre
em zero
```

A memória ocupada por uma variável array é a somatória das memórias de suas células e pode ser determinada pelo comando nativo sizeof. No exemplo,

```
var
    y: array[1..20] of Longint;        // usa 80-bytes
```

a quantidade de bytes de y é determinada por sizeof(y). Isso é equivalente ao produto `length(y)*sizeof(Longint)`.

Uma array estática pode ser atribuída diretamente para outra desde que possuam *tipos compatíveis*. Duas arrays possuem tipos compatíveis quando possuem o mesmo comprimento, a mesma faixa de índices e o o mesmo tipo base. Assim no exemplo,

```
var
    A: array[1..100] of integer;
    B: array[1..100] of integer;
    C: array[0..99] of integer;
```

A e B são compatíveis e as atribuições entre elas, utilizando um único :=, são permitidas. Entretanto, apesar do mesmo comprimento e tipo base, C não é compatível com A e B. Tentativas de atribuições diretas entre arrays não compatíveis são reportadas como erro na compilação.

Listagem 21

```
01  program arrays_compativeis;
02  var
03      i, n: integer;
04      A: array[1..5] of integer;
05      B: array[1..5] of integer;
06  begin;;
07      n := 257;
08      for i := low(A) to high(A) do begin
09          A[i] := n;
10          n := n div 2;
11      end;
12      B := A;
13      for i := low(B) to high(B) do B[i] := B[i]*2;
14      for i := 1 to 5 do
15          writeln(A[i], ' ', B[i]);
16  end.
```

A Listagem 21 Ilustra a atribuição direta entre arrays compatíveis. A é inicializada por um laço de contagem entre as Linhas 08 e 11 e seu novo conteúdo é diretamente atribuído (copiado) para B na Linha-12. O laço na Linha-13 duplica os valores nas células da array B recém carregada. O laço entre as Linhas 14 e 15 imprime o conteúdo das duas arrays. A saída deste programa tem aspecto,

```
257 514

128 256

64 128

32 64

16 32
```

Este exemplo também mostra a possibilidade de modificar e escrever arrays utilizando laços. Nestas situações cada célula recebe tratamento individual numa dada iteração de um laço que efetua varredura completa. Nas Linhas 14 e 15 do programa da Listagem - 21 foi inclusive possível processar duas arrays no mesmo laço.

Arrays unidimensionais constantes podem ser construídas em Pascal. A sintaxe adicional, em complementação à declaração de constantes e de variáveis arrays, são parênteses separando os valores invariáveis, como no exemplo,

```
const

    x: array[1..10] of integer = (2, 4, 9, 3, 11, 13, 7, 5, 21, 19);
```

O acesso às células de constantes arrays, como x, é feito exatamente da mesma forma que variáveis arrays. Entretanto o acesso é somente leitura (os valores nas células não podem ser modificados). Toda constante array em Pascal precisa ser tipada.

2.2. Arrays Multidimensionais

Arrays multidimensionais são suportadas em Pascal. Para declarar, uma array bidimensional estática pode-se tanto escrever,

```
var

    x: array[1..100] of array[1..50] of byte;          // primeira sintaxe
```

quanto,

```
var

    x: array[1..100, 1..50] of byte;                   // segunda sintaxe
```

sendo a segunda sintaxe preferencial.

Cada dimensão de uma array multidimensional estática em Pascal possui sua própria faixa de índices encapsuladas pela palavra reservada array e por colchetes individuais (primeira sintaxe) ou separadas por vírgulas e envolvidas num colchete comum (segunda sintaxe). Para acessar uma célula de uma array multidimensional, seja para leitura seja para modificação, deve-se informar os índices na quantidade que forem as dimensões e nas faixas previstas na declaração. No caso de x tem-se por exemplo,

```
x[20, 17] := 65;
```

que altera para 65 o conteúdo da célula da array x cujo índice da primeira dimensão é 20 (deve estar entre 1 e 100) e da segunda dimensão é 17 (deve estar entre 1 e 50).

Em caso de arrays bidimensionais, como x, é comum fazer a analogia com matrizes (matemática) dizendo ser a primeira dimensão a das *linhas* e a segunda a das *colunas*. Assim x possui 100 linhas e 50 colunas. Abstraindo a array seguinte como uma matriz quadrada de ordem cinco,

```
var
    mat: array[1..5, 1..5] of real;
```

então um código para carregá-la com a **matriz identidade** é,

```
for lin := 1 to 5 do
    for col := 1 to 5 do
        if col = lin then
            mat[lin, col] := 1.0
        else
            mat[lin, col] := 0.0;
```

Matriz Identidade: Na matemática é a matriz quadrada (número de linhas igual ao de colunas) cujos elementos são todos nulos, exceto aqueles da diagonal principal (diagonal cujo índice da linha e da coluna são idênticos), que valem 1.

onde col e lin são inteiros. O aninhamento dos laços neste exemplo permite varrer para cada linha (primeiro laço) as células em cada coluna (segundo laço). O total de iterações é portanto o número de células da array (25 neste caso). Este número de iterações é também o de decisões tomadas com if.

São exemplos de arrays de quantidade de dimensões superior a dois,

```
var
   A: array[1..5, 1..10, 1..100] of boolean;
   Teste: array[1..2, 1..2, 1..2, 1..5] of double;
```

Na prática arrays unidimensionais e bidimensionais são suficientes para resolução da maioria dos problemas.

Internamente arrays estáticas, independentemente da quantidade de de dimensões que possuam, são representadas em memória por sequencias

contíguas de células sendo o aspecto multidimensional um recurso da linguagem para melhorar a capacidade de abstração.

2.3. Arrays Dinâmicas

Uma *array dinâmica* em Pascal é uma array especial cujo comprimento é definido somente em tempo de execução. A sintaxe básica para declaração de uma array dinâmica é idêntica a de uma estática, exceto pela ausência de índices limites. A sintaxe básica de declaração é,

```
var
    <variavel>: array of <tipo_base>;
```

Exemplo,

```
var
    x: array of integer;
```

onde x é uma array dinâmica de inteiros.

Após a declaração, uma array dinâmica ainda não está disponível ao uso. É necessário antes *alocar memória* para ela. A unidade system fornece o comando setlength para este fim. Sua sintaxe básica, para arrays unidimensionais, é,

```
setlength(<array_dinamica>, <comprimento>);
```

onde o primeiro argumento é a variável array dinâmica que solicita a alocação de memória e o segundo o total de células que a array deverá ter após a chamada de setlength. Para alocar em x memória de 100 inteiros utiliza-se,

```
setlength(x, 100);
```

Em outras linguagens de programação funções de uso similar recebem a quantidade de bytes para alocar e não o número de células como em setlength. A memória de fato alocada é o valor do segundo argumento de entrada multiplicado pelo tamanho do tipo base (no caso de x, integer). O comando length retorna zero antes da primeira chamada a setlength e o valor exato de comprimento, depois. Assim a saída do código,

```
var x: array of integer;
begin
    write( length(x) );
    setlength(x, 25);
    writeln( ' ', length(x) );
end.
```

deve ser,

```
0 25
```

Se outras chamadas a setlength ocorrerem no mesmo programa para uma mesma array dinâmica, ocorrerá a chamada *realocação de memória* sendo tanto possível retrair quanto expandir memória. Na realocação com expansão outro espaço em memória é alocado e o conteúdo do antigo é copiado para ele. A variável array, que solicitou a realocação, passa então a se referenciar a nova memória alocada e a antiga é limpa. Na realocação com retração, dados são perdidos. Realocação em excesso compromete desempenho do programa.

Quando uma array dinâmica *perde o contexto* (o programa termina ou um módulo que a criou retorna) então ela é eliminada automaticamente da memória.

O índice inferior de uma array dinâmica é sempre zero. Assim para carregar uma array dinâmica com os primeiros 100 números ímpares deve-se escrever,

```
var
    i: integer;
    w: array of integer;
begin
    setlength(w, 100);
    for i := 0 to length(w)-1 do
        w[i] := 2*i + 1;
end.
```

Listagem 22

```
01  program arrays_dinamicas;
02  var
03      i: integer;
04      x, y: array of integer;
05  const
06      max = 5;
07  begin
08      setlength(x, max);
09      for i := 0 to max-1 do
10          x[i] := 2*i+1;
11      y := x;
12      for i := 0 to max-1 do
13          y[i] := 2*y[i];
14      for i := 0 to max-1 do
15          writeln( x[i], '  ', y[i] );
16  end.
```

A atribuição direta de uma array dinâmica a outra *não* gera uma nova array dinâmica. Ao invés disso passam a existir duas referências ao mesmo local de memória. Isso é ilustrado no programa da Listagem - 22. A array x é alocada na Linha-08 e recebe os valores dos primeiros cinco ímpares (Linhas 09 e 10).

A atribuição direta na Linha-11 cria uma nova referência para a memória de x, ou seja, x e y agora se referem a mesma array dinâmica. Nas Linhas 12 e 13 o conteúdo de cada célula da array dinâmica y é multiplicado por dois o que é equivalente a dobrar os valores das células de x. Assim as impressões nas linhas 14 e 15 geram a saída,

```
2 2

6 6

10 10

14 14

18 18
```

Para declarar arrays dinâmicas com mais de uma dimensão utiliza-se a primeira sintaxe de declaração de arrays estáticas, mas sem índices limite. Exemplos,

```
var

   A: array of array of integer;               // array dinâmica bidimensio-
nal

   B: array of array of array of integer;      // array dinâmica tridimensio-
nal
```

Arrays dinâmicas multidimensionais são também alocadas com setlength. Surge nele um argumento adicional para cada comprimento de dimensão novo. Exemplos,

```
setlength(A, 2, 5);         // aloca comprimentos 2 e 5 para as duas dimensões
da array A

setlength(B, 5, 3, 4);  // aloca comprimentos 5, 3 e 4 para as três dimensões
da array B
```

Utilizando arrays dinâmicas bidimensionais é possível generalizar o exemplo da matriz identidade. A declaração geral se torna,

```
var

    mat: array of array of real;
```

e a alocação,

```
readln(n);

setlength(mat, n, n);
```

onde n é um inteiro fornecido via entrada padrão. A inicialização fica portanto,

```
for lin := 0 to length(mat)-1 do
    for col := 0 to length( mat[0] )-1 do
        if col = lin then
            mat[lin, col] := 1.0
        else
            mat[lin, col] := 0.0;
```

onde lin e col são inteiros. Observe que na segunda chamada a length o argumento é mat[0]. De fato se mat tivesse, por exemplo, cinco dimensões, o comprimento delas seriam calculados pelas respectivas instruções,

```
length( mat );
length( mat[0] );
length( mat[0][0] );
length( mat[0][0][0] );
length( mat[0][0][0][0] );
```

3. Estudos de Caso

3.1. Palíndromos Numéricos

Um palíndromo é qualquer sequencia de informação cuja disposição dos elementos na ordem reversa (de trás para a frente) origina a mesma sequencia. As strings 'asa', 'abbba' e 'tomamot' são exemplos de palíndromos. Palíndromos numéricos são inteiros cuja sequencia de dígitos são palíndromos. Exemplos são 2, 121, 5665 e 10101.

A função e_palind da Listagem 23 determina se seu argumento de entrada é ou não um palíndromo numérico. A primeira hipótese da função é de que seja e por essa razão a variável interna e_palind recebe true na Linha-05. O laço entre as Linhas 08 e 11 determina quantos dígitos, t, possui o valor da entrada em n. Para isso uma cópia deste valor é feito para x e t iniciado com zero.

Cada iteração do laço reduz o valor em x para o quociente de sua razão por dez (Linha-10) e o laço encerra quando zero é alcançado. Como t incrementa da unidade em cada iteração, esta variável conterá no final das iterações a quantidade de dígitos do valor em n. Na Linha-12 a array dinâmica digs é alocada para registrar os dígitos do valor em n e o laço entre as Linhas 13 e 16 carrega os dígitos. O laço é decrescente para preencher os dígitos de trás para frente (porque é nesta ordem que eles são extraídos). A expressão, n mod 10, Linha-14, retorna o último dígito do valor em n ao passo que, n := n div 10, Linha-15, força sua extração.

Utilizando a array digs, agora convenientemente carregada, o laço entre as Linhas 17 e 21 determina se existe um palíndromo. Para tanto o conteúdo das células de índices entre 0 e (t div 2)-1 (esta expressão representa exatamente o maior valor de índice da primeira metade da array excluindo o valor central em caso de arrays de comprimento ímpar) são comparadas com suas imagens (os índices da segunda metade da array calculados neste caso pela expressão, t-i-1).

Para que digs contenha um palíndromo, todas as células da primeira metade (posições i) devem possuir o mesmo conteúdo que suas células imagem (posições t-i-1). Basta uma única equivalência não ocorrer (decisão da Linha-18) para o status de e_palind ser revertido para false (Linha-19) e o laço ser quebrado (Linha-20). Entretanto, se n contiver um palíndromo numérico, esta quebra nunca sucederá e o retorno da função (valor de e_palind) será o da hipótese inicial (true).

Listagem 23

```
01 function e_palind(n: longint): boolean;
02 var digs: array of byte;
03     x, t, i: integer;
04 begin
05     e_palind := true;
06     x := n;
07     t := 0;
08     while x>0 do begin
09         t := t + 1;
10         x := x div 10;
11     end;
12     setlength(digs, t);
13     for i := high(digs) downto low(digs) do begin
14         digs[i] := n mod 10;
15         n := n div 10;
16     end;
17     for i := 0 to (t div 2)-1 do
18         if digs[i] <> digs[t-i-1] then begin
19             e_palind := false;
20             break;
21         end;
22 end;
```

3.2. Ordenação por Seleção

Seja o problema de classificar em ordem crescente os valores armazenados numa array de inteiros. Há muitos programas propostos para a resolução deste problema. Apresentaremos aqui a *ordenação por seleção*.

Seja *A* uma array com limites inferior e superior *1* e *n* respectivamente. A ordenação por seleção consiste em determinar, para cada posição *i* em A,, a posição *k* do menor valor da subarray de *A*, com limites *i* e *n*, e trocar o conteúdo das posições *i* e *k*. O total de trocas realizadas está diretamente ligado ao esforço de ordenação.

Subarray:
Array obtida a partir de uma subsequência de células continuas de uma array dada.

O programa da Listagem 24 implementa a ordenação por seleção em Pascal. O laço entre as Linhas 10 e 13 carrega (e imprime) a array dat com dez valores aleatórios obtidos com o comando random. O comando random, unidade system, retorna um valor aleatório entre 0 e o valor do argumento de entrada subtraído de um. Para uso apropriado deve-se chamar previamente o comando randomize.

A ordenação por seleção de dat ocorre por ação dos laços entre as Linhas 14 e 21. A laço externo varre todas as células no intervalo 1..n-1 ao passo que o laço interno busca, através de k, pela célula de menor valor no intervalo i..n. A primeira hipótese de k em cada iteração do laço externo é o valor em i (Linha-15). As trocas entre as células de índices i e k são feitas pelas Linhas 18 a 20 usando x como variável auxiliar. O aspecto de saída deste programa será,

```
94      17      31      50      20      73      64      84      69      59

17      20      31      50      59      64      69      73      84      94
```

Listagem 24

```
01  program ordenacao_selecao;
02  const
03      n = 10;
04  var
05      dat: array[1..n] of integer;
06      i, j, k, x: integer;
07  begin
08      randomize;
09      writeln;
10      for i:= 1 to n do begin
11          dat[i] := random(100);
12          write( dat[i], #9 );
13      end;
14    for i := 1 to n-1 do begin
15          k := i;
16          for j := i+1 to n do
17              if dat[j] < dat[k] then k := j;
18          x := dat[i];
19          dat[i] := dat[k];
20          dat[k] := x;
21      end;
22      writeln;
23      for i:= 1 to n do
24          write( dat[i], #9 );
25      writeln;
26  end.
```

3.3. Hiper Fatorial

O fatorial de um número inteiro positivo é igual ao produto dele por todos seus antecessores maiores que zero (sendo ainda definido como 1 o fatorial de zero). O tipo primitivo inteiro que pode representar o maior inteiro é int64 e mesmo assim tal limite representativo é insuficiente para calcular fatoriais de números acima de 20. A solução proposta neste estudo de caso permite, através da utilização de arrays, determinar o fatorial de qualquer inteiro.

A tarefa de determinação do fatorial de um inteiro qualquer possui quatro etapas: (I) determinar o tamanho (em dígitos) do fatorial de um valor *N* dado; (II) criar uma array de dígitos com o comprimento determinado na etapa I e carregá-lo com zeros, exceto a última posição que deverá conter 1; (III) reprogramar a operação de multiplicação de forma a ser possível fazer o produto entre um escalar e a array criada em II; (IV) repetir a etapa III para todos os inteiros entre 2 e *N*. A array criada em II representa na verdade o número procurado expresso como uma array de dígitos. Os zeros seguidos de um 1 denotam a representação do número 1 que é o valor que deve ter a variável acumuladora antes dos produtos.

Variável Escalar: São todas as variáveis que não são arrays nem strings nem estruturas heterogêneas (Veja Capítulo-3).

Para resolver a primeira etapa utilizamos a equação,, que determina a quantidade de dígitos *t* na base *b* de um inteiro *n*. Assim a quantidade de dígitos do fatorial de *N* na base 10 é,

$$t = 1 + \log_{10}(N!) \qquad (1)$$

Como o logaritmo do produto é a soma dos logaritmos, a Equação-2.1 pode ser reescrita como,

$$t = 1 + \sum_{j=1}^{n} \log_{10}(j) \qquad (2)$$

Ao contrário da Equação-2.1, a Equação-2.2 é facilmente computável.

Na segunda etapa uma array de dígitos *data* é alocada, carregada e deve assumir aspecto como o esquema,

```
0  0  0  0  0  0  0  0  0  1
```

A terceira etapa, da multiplicação de um escalar *d* pela array *data*, funciona da seguinte forma. Um valor inteiro de correção *ac* é definido e iniciado com zero. Uma série de produtos de *d* por cada célula de *data* ocorrem varrendo *data* de trás para frente. O valor de cada produto é somado a *ac* para obter um valor *p*. Este valor potencialmente contém mais de um dígito sendo o último deles aquele que deve sobrepor a célula em *data* multiplicada.

Para resolver isso registra-se em *data* apenas o resto da divisão de *p* por 10 (que é seu último dígito) e guarda-se em *ac* o quociente *ac/10*. No próximo produto o novo valor de *ac* entrará na computação. De fato *ac* pode ser não nulo ao final de vários estágios sendo sempre aproveitado no estágio posterior. Para ilustrar essa etapa considere que *data* seja uma array de 3 dígitos contendo os valores {1, 3, 2} (representa o número 132) e deva ser multiplicado pelo escalar 7. Os estágios de *data*, *ac* e *p* são mostrados a seguir em série,

```
1 | 3 | 2       ac: 0  p: 2 x 7 = 14
1 | 3 | 4        ac: 1 p: 3 x 7 = 21
1 | 1 | 4        ac: 2 p: 1 x 7 = 7
7 | 1 | 4        ac: 0
```

Listagem 25

```
01  program hiper_fatorial;
02  var s: real;
03      n, t, i, d, ac, p: integer;
04      data: array of integer;
05  begin
06      write('Valor> ');
07      readln(n);
08      write('Fatorial> ');
09      s := 1.0;
10      for i := 1 to n do
11          s := s + ln(i)/ln(10.0);
12      t := round(s);
13      setlength(data, t);
14      for i := 0 to t-1 do data[i] := 0;
15      data[t-1] := 1;
16      for d := 2 to n do begin
17          ac := 0;
18          for i := t-1 downto 0 do begin
19              p := data[i] * d + ac;
20              data[i] := p mod 10;
21              ac := p div 10;
22          end;
23      end;
24      for i := 0 to t-1 do
25          write( data[i] );
26      writeln;
27  end.
```

O programa da Listagem 25 Implementa o programa de determinação de fatoriais descrito anteriormente. A primeira etapa ocorre entre as Linhas 06 e 12. A variável real s é responsável pela contabilização da quantidade de dígitos do fatorial (Equação-2.2) da entrada n. Devido a soma de logaritmos, s deve ser real. Seu valor inicial é 1.0 (Linha-09) para contabilizar a primeira parcela do lado esquerdo da Equação-2.2. O laço nas Linhas 10 e 11 contabilizam a s os logaritmos calculados por ln(i)/ln(10.0). O comando ln, unidade system, determina o logaritmo natural de um valor. O arrendondamento do valor em s (comando round, unidade system), registrado em t, é a quantidade de dígitos procurada.

O logaritmo de um número *a* na base *b*, ou seja, $\log_b(a)$, pode ser reescrito como a razão ln(*a*)/ln(*b*), onde *ln* é o logaritmo natural.

A segunda etapa ocorre entre as Linhas 13 e 15 onde ocorre alocação da array data e seu preenchimento apropriado. A terceira etapa, Linhas 17 a 22, está aninhada na quarta etapa (Laço iniciado na Linha-16). Cada valor de

d (contador do laço) é multiplicado pela array data e por essa razão ac é sempre reiniciado em zero (Linha-17). O valor calculado com p em cada iteração do laço interno usa a i-ésima célula de data que é varrido de trás para frente. Os operadores mod (na Linha-20) e div (na Linha-21) são respectivamente responsáveis pela extração do novo dígito que irá sobrepor aquele na posição i corrente em data e pela atualização de ac. As Linhas 24 e 25 imprimem a saída obtida dígito a dígito, desta vez varrendo data na sequência crescente. Um exemplo de saída do programa é,

```
Valor> 50
Fatorial> 30414093201713378043612608166064768844377641568960512000000000000
```

Síntese da Parte 1

Foram apresentadas no Capítulo-1 as estruturas básicas de construção de um programa em Pascal incluindo sintaxe básica, utilização de variáveis e constantes, uso de operadores e expressões, entrada e saída padrões, controle de fluxo e fundamentos sobre procedimentos e funções. No Capítulo-2 foram apresentadas as estruturas homogêneas sequenciais da linguagem, ou seja, strings – que permitiram a representação e manipulação de informação textual – e as arrays – que possibilitaram o sequenciamento de dados dos mais diversos tipos nativos existentes.

Atividades de avaliação

1. Criar função que receba um caractere e, caso seja uma letra, retorne sua versão maiúscula, quando a entrada for minúscula, e minúscula quando a entrada for maiúscula. Se a entrada não é uma letra, o mesmo caractere deve ser retornado.
2. Escreva uma função que receba uma string s e um caractere ch e retorne o número de vezes que ch aparece em s.
3. Escreva programa que receba uma string da entrada padrão e imprima quantas *palavras* ela contém. Considere palavras como sendo pedaços da string separados por espaços em branco.
4. Construa programa que receba uma array de inteiros e imprima como saída o maior e o menor valor contidos nela.

5. Construa programa que busque numa array os dois maiores valores contidos.

6. Construa um programa que receba uma array e altere seus valores de forma a ficarem em ordem inversa.

7. Construir programa que carregue em uma array de comprimento n os primeiros n valores primos.

8. Rotação é o procedimento de deslocamento dos itens de uma array para casas vizinhas. Numa *rotação a direita* todos os itens migram para posições logo a frente, exceto o último que vai para a posição 1, ex: se v = {1,2,3,4}, uma rotação a direita modifica v para {4,1,2,3}. De forma análoga uma *rotação a esquerda* de v geraria {2,3,4,1}. Construir programas para os dois processos de rotação.

9. Construa um programa que calcule a média dos números menores que 5000 que sejam divisíveis por 7 mas não sejam divisíveis por 3.

10. Considere uma array V que contenha os coeficientes de um polinômio P(x) de grau n. Construa programa que receba V, n e um valor de x_0 e imprima $P(x_0)$.

11. Seja uma lista L formada por n valores reais. Dado que: $\sigma = \sqrt{\sum_{i=1}^{n} (x_m - x_i)^2}$, onde x_m é a média dos elementos de L, x_i é cada um dos n elementos de L (com i = {1..n}), implementar programa que receba L e n e calcule σ.

12. Um CPF tem nove dígitos e mais dois para verificação. A verificação se dá da seguinte forma: Primeiro, multiplica-se cada um dos nove dígitos por um peso, como no esquema:
d1 d2 d3 d4 d5 d6 d7 d8 d9 dígitos

X

10	9	8	7	6	5	4	3	2	pesos
a1	a2	a3	a4	a5	a6	a7	a8	a9	

Depois se calcula a soma S1 de todos os números resultantes. O décimo dígito, d10, será 11 – (S1 mod 11),ou zero se esta conta der mais que nove. Para calcular d11, faz-se como antes, mas leva-se em conta também d10:
d1 d2 d3 d4 d5 d6 d7 d8 d9 d10

X

11	10	9	8	7	6	5	4	3	2
a1	a2	a3	a4	a5	a6	a7	a8	a9	a10

Somam-se agora todos os números obtidos. O dígito d11 será 11 – (S2 mod 11), ou zero se esta conta der mais que nove. Faça um programa que verifique se um CFP está correto.

13. Escreva programa que determine a somatória dos dígitos do valor 100!.

14. Escreva dois programas um para a conversão de números inteiros decimais em sua forma binária e outro para a operação inversa.

15. O número 585 é palíndromo nas bases 10 e base 2 (1001001001). Construir programa que determine a soma de todos os números com esta propriedade que sejam menores que um milhão.

16. Dado uma array *V* ordenada de comprimento *n* no qual se busca o valor *x*, a *busca binária* se procede como segue: (1) *inf* recebe 1 e *sup* recebe *n*; (2) se *inf* for maior que *sup*, encerrar (valor não encontrado); (3) *med* recebe a média de *inf* e *sup*; (4) *x* é comparado a *V[med]* de forma que se este valor é igual a *x* o procedimento encerra (valor encontrado), se for maior que *x* o valor de *sup* passa a ser *med-1*, senão se for menor que *x*, *inf* passa a ser *med+1*; (5) voltar a 2. Construir programa para busca binária

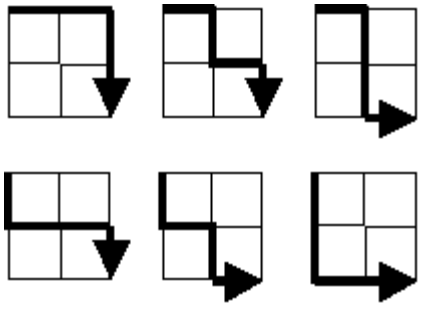
17. Considere uma lista L de valores inteiros não repetidos contendo n elementos. Desenvolva um programa que selecione m elementos aleatórios entre os n existentes de forma que também sejam todos diferentes. O valor m deve ser sempre menor que n.

18. Temos uma array com 20 números. Queremos reorganizar o array de forma que dois números pares não sejam vizinhos. Faça um programa que reorganize o array desta forma, ou diga que não é possível.

19. Construa um programa que tenha como entrada duas matrizes quadradas de ordem n e ofereça as operações algébricas de soma, subtração e multiplicação.

20. Construa um programa que receba uma matriz quadrada de ordem n e calcule seu determinante.

21. Considere uma grade 2X2 como a desenhada abaixo na qual se parte do canto superior à esquerda até o inferior à direita: há um total de 6 trajetórias possíveis. Construir programa que receba N e retorne o total de trajetórias numa grade NXN.



Referências

SEBESTA, Robert W., **Concepts of Programming Languages,** Addison Wesley; 9 edition (April 2, 2009)

CANNEYT, Michaël Van, **Reference guide for Free Pascal**, Document version 2.4 , March 2010

http://www.freepascal.org/docs-html/

Parte 2

Tipos de Dados Personalizados

Capítulo 3

Tipos de Dados Personalizados

Objetivos:

- Esta unidade tem por objetivo introduzir estruturas sofisticadas da linguagem Pascal. No Capítulo-1 são apresentadas estruturas que permitem abstrair e operar modelos ligados a objetos enumeráveis, descritos por campos ou associados a conjuntos. No Capítulo-2 é estudado o poderoso recurso dos ponteiros que tanto potencializam construções nativas do Pascal quanto permitem um mais flexível gerenciamento da memória do computador. Além das vantagens são também estudados os problemas com ponteiros. Um estudo de caso prático procura aplicar as principais estruturas abordadas no capítulo.

1. Definindo Novos Tipos

Um *tipo de dados* define um conjunto de regras utilizadas para representar e operar dados através de variáveis e constantes. Ao definir uma variável como smallint, por exemplo, ela poderá representar valores inteiros no intervalo de -128 a 127, estar presente em expressões aritméticas em operações de soma, subtração, multiplicação e divisão bem como ser alvo de uma atribuição. Todas essas características foram definidas na implementação da Linguagem Pascal e os tipos de dados construídos sobre tal condição são definidos como *nativos da linguagem*. Entre os tipos nativos estão os tipos *primitivos* (todos os numéricos, caractere e booleanos) estudados nos Capítulos 1 e 2, as *sequencias homogêneas* (strings e arrays) estudadas no Capítulo-2, as *enumerações*, *estruturas heterogêneas* e os *conjuntos*. Os três últimos serão abordados neste capítulo.

Em Pascal uma sessão rotulada com type é utilizada para definir *tipos personalizados*. Um tipo personalizado é uma novo tipo de dados construído a partir de tipos nativos e com operabilidade definida, em alguns casos, pelo programador. A forma mais simples de tipo de dados personalizado é aquela que define um novo nome para um tipo já existente. No exemplo,

```
type
        inteiro = integer;
```

o identificador inteiro passa a funcionar, após essa definição, como o tipo primitivo integer. A declaração,

```
var x: inteiro;
```

é legal neste contexto.

Cada nova definição da sessão type tem sintaxe geral,

```
type
        <identificador_do_novo_tipo> = <tipo_de_dados>;
```

onde a igualdade, =, apesar da mesma representação do operador relacional de comparação (Capítulo-1), possui semântica completamente diferente, ou seja, nomeia um tipo personalizado. O identificador que nomeia o novo tipo segue as regras de identificadores da Linguagem Pascal.

Um tipo *faixa* em Pascal representa outro exemplo de tipo personalizado que é construído numa sessão type. Trata-se de um tipo que representa uma subsequencia de números inteiros oriunda de um tipo inteiro primitivo e cuja construção é feita a partir dos valores limites numéricos inferior e superior conforme sintaxe,

```
type
        <identificador> = <limite_inferior>..<limite_superior>;
```

No exemplo,

```
type
        faixa1 = 10..20;
var
        x: faixa1;
```

um tipo denominado faixa1 é definido para que suas variáveis (tais como o exemplo x) possam representar inteiros entre 10 e 20. Atribuições explícitas de valores fora da faixa definida (como, x := 89, neste exemplo) fazem o compilador emitir uma advertência, mas não um erro.

O tamanho em bytes de uma variável de tipo faixa é o menor valor inteiro *n* tal que 2^{8n} seja maior ou igual ao total de inteiros cobertos pelo tipo. Por exemplo na faixa 10..20, são cobertos 11 inteiros e o menor valor de *n* é 1-byte pois $2^{81} = 2^8 = 256 > 11$. Para a faixa 10..1000, que cobre 991 inteiros, *n* vale 2-bytes pois $2^{82} = 2^{16} = 65536 > 991$. Vaiáveis de tipo faixa são operadas normalmente como qualquer outra variáveis inteira.

Tipos arrays estáticas podem também ser definidos numa sessão type. No exemplo,

```
type
        tarr = array[1..10] of integer;
```

O tipo tarr pode definir variáveis e constantes que comportam sequencialmente dez inteiros. Nas declarações,

```
const
        c: tarr = (5,4,1,3,2,0,9,7,8,6);
var
        x: tarr;
```

a constante c e a variável x definem duas arrays de dez inteiros.

O compilador Pascal reconhece atribuições entre dados de um tipo array estático personalizado. Assim, neste último caso, a atribuição,

```
x := c;
```

pode ser utilizada para inicializar x. As dez células de c têm o conteúdo copiado para as células de x sem necessidade de um laço. O mesmo acontece com strings, como no exemplo,

```
type
    mensagem = string[100];
const
    erro_1: mensagem = 'disco falhou!';
var
    s: mensagem;
begin
    s := erro_1;
    writeln(s);
end.
```

Listagem 26

```
01  program tipo_array_dinamico;
02  type
       tdyarr = array of integer;
03  var
       x, y: tdyarr;
04     i: integer;
05  begin
       setlength(x, 20);
06     for i := low(x) to high(x) do x[i] := i*i;
07     y := x;
       for i := low(x) to high(x) do y[i] := y[i]*2;
08     for i := low(x) to high(x) do write(x[i], ' ');
09  end.
10
11
12
13
```

No caso de arrays dinâmicas o tipo personalizado pode ser definido na sessão type com a mesma sintaxe das definições anteriores. Entretanto as atribuições diretas não são capazes de copiar os dados efetivamente (veja arrays dinâmicas no Capítulo - 2). No exemplo da Listagem 26, a array dinâmica x, de tipo tdyarr, é alocada (Linha-08) e ilicializada com os dez primeiros quadrados perfeitos (Linha-09). A atribuição direta para y entretanto só torna esta variável uma nova referência ao bloco de memória alocado na Linha-08. Isso tanto é verdade que, ao dobrar o valor de cada célula da array y (Linha-11), o efeito é sentido por x (impressão da Linha-12).

2. Enumerações

Uma enumeração é um tipo faixa especial cujos inteiros cobertos possuem cada um seu próprio identificador. A sintaxe geral de um tipo enumeração Pascal é a seguinte,

```
type
       <nome> = (<ident_1>, <ident_2>, <ident_3>, {...}, <ident_n>);
```

Tipos enumerações são utilizados para modelar dados que seguem algum critério numérico de ordenação e que comummente são manipulados por nomes próprios. Um exemplo comum são os dias da semana que podem ser representados por variáveis da enumeração seguinte,

```
type
       dia = (seg, ter, qua, qui, sex, sab, dom);
```

Os identificadores entre parênteses não podem ser utilizados adiante no mesmo programa, por exemplo, para representar constantes ou variáveis. Atrubuições a variáveis de tipo enumeração são construídas repassando-se diretamente um dos identificadores definidos no tipo. Exemplo,

```
var
        hoje: dia;
begin
        hoje := dom;
end.
```

Quando um valor enumeração é impresso por write/writeln, uma versão string do identificador é repassada a saída padrão. No exemplo,

```
write('Hoje e ', hoje);
```

onde hoje contém dom, é impresso *hoje* e *dom*.

Como enumerações são casos especiais de faixas, seus identificadores possuem equivalentes numéricos. No padrão Pascal o primeiro elemento de uma enumeração equivalente a *zero*, o segundo a *um* e assim por diante. Para determinar o equivalente numérico de um identificador de uma enumeração utiliza-se o comando ord. No exemplo,

```
writeln( ord(hoje) );
```

onde hoje vale dom, é impresso 7.

É possível modificar a sequência numérica de equivalência de uma enumeração. Para tanto basta modificar o equivalente numérico de um dos identificadores e os dos demais corresponderão a valores sucessores ou antecessores, dependendo do caso. Para modificar o equivalente numérico de um identificador deve-se atribuir a ele um valor inteiro com o operador de atribuição diretamente na definição da enumeração. Na redefinição da enumeração dia,

```
type
        dia = (seg=27, ter, qua, qui, sex, sab, dom);
```

o primeiro identificador tem seu equivalente numérico modificado para 27 e os demais respectivamente para 28, 29, 30, 31, 32 e 33. Neste outro exemplo,

```
type
        dia = (seg, ter, qua=27, qui, sex, sab, dom);
```

a nova sequência de equivalentes numéricos é 25, 26, 27, 28, 29, 30 e 31.

Os comandos low e high, da unidade system, podem ser utilizados para determinar os identificadores de um tipo enumeração respectivamente de me-

nor e maior equivalentes numéricos. Estes comandos recebem apenas um argumento que pode ser uma variável enumeração ou o nome de um tipo enumeração. No exemplo,

```
writeln( low(dia), ' a ', high(dia) );
```

imprime em saída padrão *seg a dom*.

Enumerações podem constituir expressões relacionais sendo as comparações efetuadas de acordo com os equivalentes numéricos. No exemplo,

```
var a, b: dia;
begin
    a := ter;
    b := qui;
   if a > b then writeln('passou o prazo') else writeln('ainda esta em tempo');
end.
```

é impresso *ainda esta em tempo* na saída padrão pois o equivalente numérico de ter é menor que o de qui.

Contadores de laços de contagem podem ser de tipo enumeração. O exemplo,

```
type dia = (seg, ter, qua, qui, sex, sab, dom);
var d: dia;
begin
    for d := seg to dom do
        writeln( d, ' equivale a ', ord(d) );
end.
```

Imprime em saída padrão,

```
seg equivale a 0
ter equivale a 1
qua equivale a 2
qui equivale a 3
sex equivale a 4
sab equivale a 5
dom equivale a 6
```

Utilizando os comandos low e high, o laço anterior pode ser reescrito como,

```
for d := low(dia) to high(dia) do
        writeln( d, ' equivale a ', ord(d) );
```

Uma enumeração pode ser anônima, ou seja, definida diretamente com as variáveis numa sessão var. No exemplo seguinte,

```
var
        mes: (jan, fev, mar, abr, mai, jun, jul, ago, stb, otb, nov, dez);
```

a variável mes é de um tipo enumeração anônimo que modela os meses do ano.

3. Estruturas Heterogêneas

Uma *estrutura heterogênea* em Pascal é um tipo de dados construído pelo encapsulamnto de outros tipos de dados que podem ser nativos ou pernonalizados. O encapsulamento funciona pela definição de identificadores e seus tipos cada qual responsável por uma fatia dos dados que uma variável ou constante desta estrutura deverá conter. Cada par identificador/tipo de uma estrutura heterogênea é denominado *campo*. A sintaxe geral de um tipo estrutura heterogênea em Pascal é,

```
type
    <nome> = record
        <campo_1>: <tipo_1>;
        <campo_2>: <tipo_2>;
        <campo_3>: <tipo_3>;
        ...
        <campo_n>: <tipo_n>;
    end;
```

onde a palavra reservada record (que significa *registro*) forma com end o bloco encapsulador de campos. O exemplo,

```
type
    aluno = record
        nome: string[25];
        idade: byte;
        media: real;
    end;
```

defiine um tipo de dados estrutura heterogênea chamado aluno que contém três campos denominados respectivamente de nome, idade e media. O novo tipo é utilizado para representar, através de cada variável que definir, informações gerais de um aluno no final do semestre letivo. O campo nome é usado para armazenar o nome completo com até 25 caracteres, idade o valor da

idade e media a média final. A declaração de variáveis de tipo estrutura heterogênea segue a mesma sintaxe já apresentada. No exemplo,

```
var
        x: aluno;
```

define uma variável de tipo aluno.

Os campos de uma variável estrutura heterogênea podem ser acessados diretamente como se fossem variáveis independentes. Este acesso é contruído com o operador ponto (.) posto entre a variável estrutura heterogênea e o identificador de um de seus campos. Por exemplo, para inicializar a variável x no exemplo anterior pode-se escrever,

```
x.nome := ‘pedro da silva’;
x.idade := 25;
x.media := 8.7;
```

De fato, em qualquer momento do programa, os campos de uma variável estrutura heterogênea podem ser modificados ou lidos independentemente. Para imprimir em saída padrão o conteúdo de x pode-se implementar,

```
writeln(‘Nome: ‘, x.aluno, ‘ Idade: ‘, x.idade, ‘ Media: ‘, x.media:0:1);
```

Uma forma alternativa de ler ou modificar os campos de uma variável estrutura heterogênea, num mesmo contexto, é utilizando *referências elípticas*. Uma referência elíptica é um recurso que permite modificar os campos da variável diretamente através de seus identificadores sem uso do operador ponto. Para tando toda os acessos (tanto em expressões como em atribuições) devem ocorrer dentro de um bloco with. A sintaxe de um bloco with Pascal é,

```
with <variavel_estrutura_heterogenea> do
begin
    { corpo }
end;
```

onde o *corpo* contém código Pascal que utiliza os campos da variável disposta entre with e do. Estas palavras reservadas significam *com* e *faça* respectivamente. No exemplo,

```
with x do begin
    nome := 'pedro da silva';
    idade := 25;
    media := 8.7;
    writeln('Nome: ', aluno, ' Idade: ', idade, ' Media: ', media:0:1);
end;
```

a variável x é inicializada e impressa através de referências elípticas.

Estruturas heterogêneas podem conter campos de mesmo tipo desde que possuam identificadores distintos. A estrutura seguinte, por exemplo,

```
type ponto = record
    x, y: real;
end;
```

possui dois campos de nomes x e y e representa um **ponto no plano**.

Ponto no Plano: Os pontos no plano cartesiano são representados por uma coordenada no eixo x (abcissas) e outra no eixo y (ordenadas).

Utilizando uma estrutura é possível abstrair uma entidade que facilite o trabalho do programador. Por exemplo, aplicando a ideia de ponto no plano da estrutura anterior é possível escrever funções e procedimentos que executem tarefas de **geometria analítica plana**. Neste contexto uma função que calcule a distância entre dois pontos no plano pode ser implementada como segue,

Geometria Analítica Plana: Parte da matemática que trata da resolução de problemas da geometria plana através do equacionamento de entidades primitivas como pontos, retas e cônicas. A geometria analítica não depende de desenhos (apesar de ajudarem) como a geometria plana pura.

```
function dist(p, q: ponto) : real;
begin
        dist := sqrt( sqr(p.x-q.x) + sqr(p.y-q.y) );
end;
```

onde sqrt e sqr são funções da unidade system que retornam respectivamente a raiz quadrada e o quadrado de um número. Os argumentos de entrada p e q, de tipo ponto, permitem uma codificação mais elucidativa sobre o problema da distância entre dois pontos no plano.

Um tipo estrutura heterogênea pode ser utilizada como saída de uma função para permitir que mais de uma informação possa ser retornada. Por exemplo, no problema da equação do segundo grau, duas raízes precisam ser devolvidas, quando existirem. Um modelo de estrutura heterogênea proposta para resolução do problema é a seguinte,

```
type root = record
        x1, x2: real;
        erro: integer;
end;
```

Os dois primeiros campos, x1 e x2, de tipo real, representam as raízes reais do polinômio (se existirem) ao passo que erro, de tipo integer, indica se houve ou não erro operacional. Quando nenhum erro ocorre o valor de erro é zero e x1 e x2 contêm valores válidos de raízes. Quando erro não é nulo então um erro ocorreu e os valores em x1 e x2 são inconsistentes.

Os demais códigos de erros considerados são apenas dois: 1, quando o polinômio não é de segundo grau e 2, quando não existem raízes reais. A função eq2g na Listagem-3.2 implementa o modelo. Na Linha-01 são recebidos os coeficientes da equação do segundo grau () via argumentos a, b e c. Na Linha-04 faz-se a hipótese de que raízes reais de fato existem até que se prove o contrário (o código do erro é zero).

O teste da Linha-05 avalia o quanto o valor em a está próximo de zero (a função abs, unidade system, retorna o valor numérico absoluto). Estando muito próximo de zero então o processamento desvia para a Linha-13 onde o código de erro é mudado para 1 e a função encerrada (um valor de a nulo, ou muito próximo disso, descaracteriza a equação que passa a ser de primeiro grau).

Do contrário o valor de delta é calculado (Linha-06) e testado (Linha-07). Caso este seja negativo então o processamento é desviado para a Linha-11 onde o código de erro é mudado para 2 (delta negativo significa raízes complexas) e a função encerrada. Do contrário as Linhas 08 e 09 processam os valores das raízes corretamente. Atente que neste caso o código de erro não muda pois a primeira hipótese é verdade.

Listagem 27

```
01  function eq2g(a,b,c: real): root;
02  var delta: real;
03  begin
04       eq2g.erro := 0;
05       if abs(a)>1.0e-5 then begin
06           delta := sqr(b) - 4*a*c;
07           if delta>=0 then begin
08               eq2g.x1 := (-b+sqrt(delta))/(2*a);
09               eq2g.x2 := (-b-sqrt(delta))/(2*a);
10           end else
11               eq2g.erro := 2;
12       end else
13           eq2g.erro := 1;
14  end;
```

A utilização da função eq2g da Listagem 27 pode ser ilustrada por,

```
var
       a,b,c: real;
       r: root;
begin
       readln(a,b,c);
       r := eq2g(a,b,c);
       case r.erro of
       0: writeln('Raizes: ', r.x1:7:3, r.x2:7:3);
       1: writeln('Polinomio invalido');
       2: writeln('Raizes complexas');
       end;
end.
```

Estruturas heterogêneas podem agrupar arrays, enumerações ou mesmo outras estruturas heterogêneas. No exemplo,

```
type estudante = record
        nome: string[30];
        notas: array[1..4] of real;
        turma: record
                serie: byte;
                letra: char;
                turno: (manha, tarde, noite);
        end;
end;
```

O campo notas pode registrar quatro notas de um estudante em uma mesma variável. Para modificar tais notas faz-se a associação das sintaxes já apresentadas, por exemplo,

```
x.notas[1] := 7.6;
x.notas[2] := 8.7;
x.notas[3] := 8.0;
x.notas[4] := 6.2;
```

onde x é do novo tipo estudante. O campo turma é de um tipo estrutura heterogênea anônima com três campos por se configurar sendo o terceiro de um tipo enumeração, também anônimo. Usando referências elípticas pode-se, por exemplo, escrever,

```
with x.turma do begin
        serie := 8;
        letra := 'C';
        turno := manha;
end;
```

ou equivalentemente sem referências elípticas,

```
x.turma.serie := 8;
x.turma.letra := 'C';
x.turma.turno := manha;
```

Neste caso dois níveis de chamada ao operador ponto ocorrem devido a hierarquia dentro da estrutura.

Estruturas heterogêneas anônimas podem ser aplicadas diretamente em varáveis como no exemplo,

```
var y: record
        check: boolean;
        lote: integer;
end;
```

Nestes casos o novo tipo fica *ancorado* e de uso exclusivo das variáveis definidas juntamente com ele. O acesso a campos descrito anteriormente funciona da mesma forma.

Para definir uma constante de estrutura heterogênea utiliza-se a sintaxe geral seguinte,

```
const
     <nome>: <tipo_estrutura_heterogenea> =
        (<campo1>:<valor1>;
         <campo2>:<valor2>;
         <campo3>:<valor3>;
         ...
         <campon>:<valorn>);
```

O exemplo a seguir ilustra a definição de algumas constantes de estruturas heterogêneas declaradas anteriormente,

```
const
    um_aluno: aluno = (nome: 'pedro'; idade: 25; media: 8.5);
    p: ponto = (x: 10.0; y: 30.0);
    my_root: root  = (x1: 0.0; x2: 0.0; erro: 0).
    aplicado: estudante =
       ( nome: 'raul';
         notas: (9.5, 10.0, 8.5, 9.4);
         turma: (serie: 8; letra: 'C'; turno: manha) );
```

É possível definir uma array cujas células são de um tipo estrutura heterogênea. Elas seguem a mesma sintaxe das arrays comuns. No exemplo,

```
var
        lista: array[1..100] of ponto;
```

a array lista comporta 100 dados de tipo ponto.

O acesso individual aos campos de cada uma das células de uma array de estrutura heterogênea é construído também com o operador ponto que neste caso é posto entre o segundo colchete do operador de indexação e o identificador do campo. Por exemplo, para carregar lista com 100 pontos da função, com *x* no intervalo [5: 50], pode-se escrever,

```
for i := 1 to 100 do begin
    lista[i].x := 5.0 + 45.0*(i-1)/99;
    lista[i].y := sqr( lista[i].x );
end;
```

ou ainda utilizando referências elípticas,

```
for i := 1 to 100 do
    with lista[i] do begin
        x := 5.0 + 45.0*(i-1)/99;
        y := sqr( x );
    end;
```

4. Conjuntos

As três opções mais comuns de estilos de formatação de texto são o negrito, o itálico e o sublinhado. Tais estilos podem ocorrer ao mesmo tempo, aos pares, apenas um ou mesmo nenhum deles fazendo um total de oito possibilidades. Aos estilos de formatação denominamos de *domínio de representação* do problema e às possibilidades combinatórias de *conjuntos*. De forma geral um domínio com *n* elementos rende um total de 2^n conjuntos.

A linguagem Pascal suporta a implementação de conjuntos. Um conjunto em Pascal é construído sobre um tipo inteiro, faixa ou enumeração (denominado *tipo base*) e registra quaisquer uma das combinações que os elementos deste tipo podem formar. A sinaxe básica de definição de um tipo conjunto Pascal é,

```
type
        <nome> = set of <tipo_base>;
```

onde as palavras reservadas set of significam *conjunto de*. Para modelar o exemplo dos estilos de texto ilustrado no início da sessão, pode-se implementar,

```
type
        estilo = (negrito, italico, sublinhado);
        estilos = set of estilo;
```

Neste exemplo o tipo estilo define uma enumeração de estilos de texto ao passo que estilos define um tipo conjunto de estilos de texto.

As operações básicas suportadas por conjuntos em Pascal são,

- Atribuição
- Inclusão

- Remoção
- Verificação

Numa atribuição a variável conjunto recebe integralmente os itens que deverá conter. Se ela contém outras informações, estas são sobrepostas. A sintaxe geral de uma atribuição para uma variável conjunto é,

```
<variavel_conjunto> := [ <lista_de_itens> ];
```

São obrigatórios os colchetes e a lista de itens pode inclusive ser vazia. Quando mais de um item esta presente, devem ser separados por vírgulas. As seguintes atribuições,

```
s := [];

s := [italico];

s := [negrito, sublinhado];

s := [negrito, italico, sublinhado];
```

são todas válidas e inicializam s, de tipo estilos, em várias possibilidades.

A ordem que os identificadores aparecem nas atribuições não é importante. A repetição de um mesmo identificador numa dada atribuição gera um erro de compilação.

Uma operação de inclusão é aquela que adiciona um ou mais itens a uma dada variável de tipo conjunto. Caso o conjunto já contenha algum ou alguns dos itens que se almeja inserir então são inseridos apenas os que ainda não estão presentes. Uma inclusão é construída com uma soma (operador +) e uma atribuição (operador :=). Os novos itens incluídos devem estar sempre entre colchetes (separados por vírgulas). Nos exemplo,

```
s := s + [negrito];

t := t + [sublinhado, negrito];
```

caso ainda não estejam presentes, a opção negrito é incluída na varável conjunto s e as opções negrito e sublinhado na variável conjunto t.

Uma operação de remoção exclui de uma variável conjunto um ou mais itens se eles estiverem presentes. A construção é similar a inclusão, mas substitui o operador + por -. No exemplo,

```
s := s - [negrito, sublinhado];
```

as opções de negrito e sublinhado são removidas da variável conjunto s caso estejam presentes. Se apenas uma estiver presente, será removida. Se nenhuma estiver presente, nada ocorrerá.

A operação de verificação em conjuntos consiste na checagem da presença de um determinado item. Ao contrário da atribuição, inclusão e remoção, a verificação só pode ser executada item a item. A Linguagem Pascal

possui o operador in exclusivo para verificação de itens em conjuntos. Trata-se de um operador relacional posto entre o item que se deseja verificar a presença e a variável conjunto. Sintaticamente,

```
<identificador_do_item> in <conjunto>
```

Por exemplo, para verificar se a variável conjunto s, de tipo estilos, contém a opção negrito, pode-se usar,

```
if negrito in s then
        writeln('Texto em negrito!');
```

Sematicamente a expressão em if pergunta se negrito é parte do conjunto s.

Para verificar simultaneamente mais de um item deve-se usar operadores lógicos, como em,

```
if (italico in s) and (sublinhado in s) then
        writeln('Texto em italico e sublinhado');
```

Atividades de avaliação

1. Construa uma estrutura heterogênea que modele uma data incluindo dia, mês e ano. Use uma enumeração para representar os meses.
2. Utilizando o modelo de data do problema-1 construa programa que leia duas datas e diga qual delas está a frente da outra.
3. São considerados anos bissextos todos os anos múltiplos de 400 ou os múltiplos de 4 e não múltiplos de 100. Utilizando essas informações, uma estrutura heterogênea para datas e uma enumeração para os dias da semana, construir programa que determine em que dia da semana cai determinada data.
4. Utilizando estruturas heterogêneas montar um modelo para números complexos e implementar funções que efetuem as operações de soma, subtração, multiplicação e divisão de dois complexos dados.
5. Expandir o modelo da questão anterior e criar uma função que retorne raízes reais ou complexas de um polinômio de segundo grau.
6. Vetores no espaço possuem três componentes referentes aos eixos x, y e z. Construir modelo para vetor e implementar função que retorne o módulo de um vetor recebido como argumento. O módulo de um vetor é calculado pela raiz quadrada da somatória dos quadrados de suas componentes.

7. O produto vetorial de dois vetores no espaço de componentes $\langle a_x, a_y, a_z \rangle$ e $\langle b_x, b_y, b_z \rangle$ é dado pelo determinante $\begin{bmatrix} a_x & a_y & a_z \\ b_x & b_y & b_z \\ 1 & 1 & 1 \end{bmatrix}$. Criar programa que, utilizando o modelo de vetores do problema-4, determine o produto vetorial de dois vetores dados.

8. Construir modelo, utilizando arrays e estruturas heterogêneas, que represente uma lista de partículas no plano. Uma partícula deve conter informações sobre coordenadas e sua massa. O *centro de massa* de um sistema de partículas é um ponto cujas coordenadas são calculadas poronde (x_i, y_i), m_i e M são respectivamente a coordenada e massa da i-ésima partícula e a massa do sistema. Construir função que, utilizando o modelo construído, receba uma lista de partículas e retorne a coordenada do centro de massa equivalente.

9. Num jogo de computador deve-se clicar numa área na tela com o botão esquerdo do mouse para se ver o que há por trás. Se houver uma bomba o jogo se encerra. Se a área estiver trancada ela não poderá ser aberta desta forma. Uma área é trancada e destrancada com o clicar do botão direito do mouse. Crie um conjunto que defina o estado desta área (minado, aberto, trancado e detonado) e construa uma função que simule uma tentativa de clique.

Capítulo 3

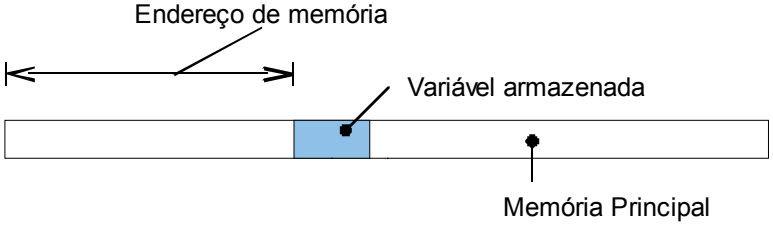
Ponteiros

1. Fundamentos

Ponteiro é um tipo de varável utilizada para armazenar endereços de outras variáveis. Assim a informação registrada em um ponteiro é um **endereço de memória**. Como a forma de endereçamento pode mudar entre arquiteturas distintas, então o conteúdo de um ponteiro vai depender do hardware utilizado e do sistema operacional instalado.

Mas convenientemente, num mesmo sistema em operação, qualquer referência a memória é feita com a mesma nomenclatura e cada variável instanciada tem um endereço distinto. Dificilmente um programador que trabalha com uma linguagem de alto nível, como Pascal, terá diretamente que tratar com valores de endereços das variáveis de seus programas. Como se verá neste capítulo o uso de ponteiros é quase sempre para uma operação indireta.

A título de visualização, abstraindo-se a memória como uma estrutura linear, é possível conceber um modelo que define o endereço como a quantidade de bytes que antecede a variável endereçada (Figura - 7).



Ponteiros possuem duas aplicações práticas,

- Referenciamento Indireto.
- Gerenciamento dinâmico de memória.

O *referenciamento indireto* é o mecanismo pelo qual um ponteiro, utilizando o endereço de variável que armazena, lê ou modifica o conteúdo desta

variável. O referenciamento indireto de ponteiros é também conhecido como *desreferenciamento*.

Gerenciar dinamicamente a memória significa, em tempo de execução, requerer blocos de memória e devolvê-los quando não mais são necessários. Este requerimento é feito diretamente ao sistema operacional sobre o qual o programa executa. O sistema responde repassando, para cada solicitação, um valor de endereço de uma área livre de memória com o tamanho solicitado. Para cada solicitação o programa deverá manter um ponteiro que receberá o valor de endereço repassado pelo sistema.

Este processo de requisição de memória é conhecido como *alocação dinâmica de memória*. Utilizando o desreferenciamento e aritmética de endereços (que será estudada mais adiante) o programa poderá efetuar operações de leitura e escrita na nova área de memória alocada. Antes que o programa se encerre, toda memória dinamicamente alocada deverá ser formalmente devolvida (*desalocação dinâmica*). Quando isso não ocorre a área de memória mantém o status de ocupada (memória ociosa) enquanto o programa executa.

O gerenciador de memória do sistema divide a memória entre memória de alocação estática (*stack*) e memória de alocação dinâmica (*heap*). Todas as variáveis declaradas explicitamente num programa, ou seja, que possuem um nome, são alocadas em *stack*, ao passo que toda alocação dinâmica requer memória em *heap*.

Na alocação estática todas as variáveis possuem nome e a desalocação é automática. Na alocação dinâmica os blocos de memória alocados são também variáveis, mas sem nome e por essa razão necessitam de ponteiros para poderem ser manipuladas devidamente. Num processo de alocação dinâmica duas variáveis estão presentes: o ponteiro e a variável sem nome cujo endereço ele armazena.

Diz-se que ponteiros *apontam* para outras variáveis. Assim, o valor apontado por um ponteiro corresponde ao conteúdo da variável endereçada e não ao conteúdo do ponteiro em si. Se um desreferenciamento com atribuição ocorre, o conteúdo do ponteiro não muda, mas sim o da variável apontada. Se o conteúdo do ponteiro é mudado por alguma razão, ele deixa de ter acesso à variável que endereçava e passa a atuar sobre o novo endereço que recebeu.

O conteúdo para o qual um ponteiro aponta pode diversificar em tipo e tamanho. Um ponteiro pode apontar para um escalar (integer, real, char e etc), para uma estrutura heterogênea (record), para uma array ou mesmo para outro ponteiro. Já o conteúdo de um ponteiro, ou seja, o valor de endereço que ele armazena, possui para uma dada arquitetura de CPU sempre o mesmo tamanho em bits não importando a natureza do conteúdo apontado.

Uma arquitetura cujos endereços de memória têm *n*-bits, pode representar um total de 2^n endereços distintos. Haja vista que cada endereço refere-se a 1-byte distinto (veja Figura-4.1) então o tamanho máximo de memória principal gerenciável deve ser exatamente 2^n-bytes. Para se ter uma ideia, em arquiteturas populares onde *n* é 32-bits, este total é de 2^{32} bytes, ou seja, 4-GB. Quando a memória máxima gerenciável é ultrapassada no hardware, o sistema não reconhece a memória excedente.

2. Declaração e Inicialização de Ponteiros

Existem duas categorias de ponteiros em linguagem Pascal: os ponteiros *tipados*, que conhecem o tipo das variáveis para onde apontam, e os *não-tipados*, que não conhecem o tipo da variável para onde apontam, mas que em contra partida, podem apontar para variáveis de quaisquer tipo.

A sintaxe para declarar um ponteiro tipado em Pascal é,

```
var
        <ponteiro>: ^<tipo_base>;
```

onde o circunflexo (^) é denominado de *operador de declaração de ponteiro*. Ele deve ser pré-fixado a um tipo de dados indicando que o ponteiro declarado apontará para uma variável deste *tipo base*. O tipo base pode ser um tipo primitivo como,

```
var
    p: ^integer;
    q: ^real;
    pt: ^boolean;
```

Diz-se que `p` é um ponteiro para inteiro, `q` para real e `pt` para booleano.

pode ser um tipo estrutura heterogênea,

```
type
        TReg = record
                x, y, z: Byte;
        end;
var
        reg: ^TReg;
```

pode ser de um tipo array,

```
type
        MyVec = array[1..3] of integer;
var
        pVec: ^MyVec;
```

ou mesmo ser de um tipo ponteiro,

```
type
        MyPtInt = ^integer;
var
        pp: ^MyPtInt;
```

Note que a formação de um tipo ponteiro, como no exemplo de MyPtInt, segue a mesma sintaxe de declaração, exceto pela igualdade (=) exigida em definições de tipos personalizados (veja Capítulo-3).

Para declarar ponteiros não-tipados deve-se utilizar a palavra chave pointer como tipo (*sem* o operador de inicialização, ^). Esquematicamente,

```
var
        <variavel>: pointer;
```

Um ponteiro declarado e não inicializado pode conter um valor descontextualizado de endereço. Diz-se, neste caso, que o ponteiro armazena um endereço *não seguro*. Operar endereços não seguros pode causar efeitos colaterais de funcionamento ou mesmo suspensão de execução do programa (isso ocorre quando o sistema operacional se defende do uso inapropriado da memória feito pelo programa). Há três maneiras em Pascal de inicializar um ponteiro com um valor *seguro* de endereço,

- Fazer o ponteiro *apontar para lugar nenhum.*
- Atribuir diretamente ao ponteiro o endereço de uma variável já existente.
- Alocar dinamicamente um bloco de memória e delegá-lo ao ponteiro.

Analisaremos os dois primeiros tópicos nessa sessão. O terceiro tópico será estudado na Sessão- 4.5 .

Um ponteiro aponta para *lugar nenhum* quando seu conteúdo não conduz a uma posição real da memória. A palavra chave nil, em Pascal, é utilizada para se referir a este valor inerte de endereço. Atribuir diretamente nil a quaisquer ponteiros (tipados ou não), fazem-nos apontar para lugar nenhum. Assim,

```
p := nil;
```

faz com que o ponteiro p aponte para parte alguma.

Para atribuir diretamente o endereço de uma variável a um ponteiro, utiliza-se o operador @. Este operador determina o endereço de uma variável qualquer à qual está pré-fixado. O resultado deve ser atribuído a um ponteiro. Exemplo,

```
p := @i;
```

Neste exemplo, i é um inteiro e p é um ponteiro de inteiro. Aqui p recebe o endereço de i (apenas uma cópia do endereço de i e não o conteúdo de i). Passam a existir duas referências para a mesma posição na memória como se pode visualizar na Figura - 8.

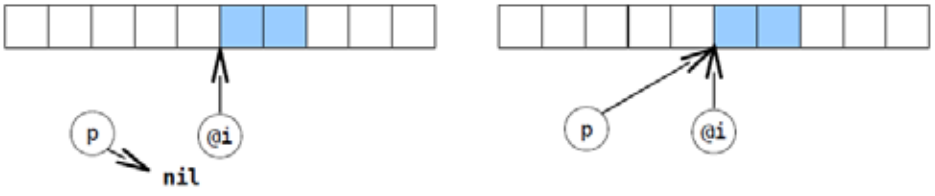


Figura 8

Listagem 28

```
01  program PonteiroAplicado;
02  var
03     i: integer;
04     p: ^integer;
05  begin
06     i := 127;
07     p := @i;
08        writeln(p^);
09        p^ := 231;
10        writeln(' ', i);
11  end.
```

Para ler ou modificar o conteúdo de uma variável apontada por um ponteiro, utiliza-se o *operador de desreferenciamento*, cujo símbolo é ^. Este novo operador, apesar da mesma simbologia do operador de declaração, é pós-fixado na variável ponteiro para indicar que se trata da variável apontada e não do próprio ponteiro (a ideia de uso do circunflexo remete a uma seta vertical que aponta para cima lembrando que se está apontado para outra parte da memória). A Listagem 28 Ilustra o desreferenciamento de ponteiros. Na Linha-08 o valor da variável apontada por p, ou seja, i, é apenas lido da memória e impresso pelo writeln. Na Linha-09 a variável apontada, ainda i, é modificada pela atribuição. Assim a saída deste programa deve ser,

Para acessar uma variável via ponteiro recorre-se inicialmente a célula de memória onde está o endereço, valor o qual conduzirá a outra célula. O fato de se não ir diretamente ao local final caracteriza uma *indireção*. Vários níveis de indireção surgem quando se operam com ponteiros de ponteiros.

```
127 231
```

A **indireção** dos ponteiros se mantem em cópias dos ponteiros. Assim, por exemplo, se no exemplo da Listagem-4.1 houvesse outro ponteiro de inteiro q ainda seria legal escrever,

Um modelador é um recurso da Linguagem Pascal que obriga um valor de um tipo a se tornar de outro tipo. É a chamada *coerção de dados*. A sintaxe de uma coerção é *tipo*(*valor*) onde *tipo* denota o tipo de dados para o qual *valor* deve ser coagido.

```
i := 127;

p := @i;

q := p;         // Cópia de ponteiro

writeln(q^);           // A cópia mantém a indireção
```

A atribuição q:=p copia o endereço armazenado em p para o ponteiro q. Assim q^ funciona como p^, ou seja, duas referências à variável i. Se o tipo base do ponteiro q não for o mesmo do ponteiro p a atribuição falhará (ocorre um erro de compilação). Entretanto, utilizando a **coerção de modeladores** é possível driblar essa restrição.

Listagem 29

```
01  program PonteiroAplicado2;
02  type
03     PtInt = ^integer;
04     PtReal = ^real;
05  var
06     i: integer;
07     p, q: PtInt;
08     r: PtReal;
09  begin
10     i := 127;
11     p := @i;
12     r := PtReal(p);  // Coerção de ponteiro de inteiro para real
13     q := PtInt(r);  // Coerção de ponteiro de real para inteiro
14     writeln(q^);
15  end.
```

A coerção de ponteiros é ilustrado na Listagem 29. Foram definidos os tipos ponteiro de inteiro (PtInt) e ponteiro de real (PtReal), nas respectivas Linhas 03 e 04, para serem utilizados como modeladores. Na Linha-12 o modelador coage o ponteiro de real r a receber o endereço do ponteiro de inteiro p. O processo inverso ocorre na Linha-13 onde desta vez o ponteiro de inteiro q é coagido a receber o conteúdo do ponteiro de real r. Apesar das coações q é definido como referência da variável i e a Linha-14 deve imprimir 127.

No caso de ponteiros de ponteiros é fundamental lembrar que para declará-los é necessário criar tipos auxiliares. Já o desreferenciamento é feito com ^^. A inicializado requer compatibilidade de tipos. A Listagem 30 ilustra o uso de ponteiros de ponteiros. Neste programa p recebe o endereço de i e em seguida q recebe o endereço de p (q é do tipo ponteiro de ponteiro). Na

Linha-12 o desreferenciamento duplo q^^ é necessário porque dois níveis de indireção estão presentes. A saída deste programa será,

```
127 127 127
```

Listagem 30

```
01  program PonteiroAplicado3;
02  type
03     PtInt = ^integer;
04  var
05     i: integer;
06     p: PtInt;
07    q: ^PtInt;
08     begin
09       i := 127;
10       p := @i;
11       q := @p;
12       writeln(i, ' ', p^, ' ', q^^);
13  end.
```

3. Ponteiros e Arrays

Se um ponteiro aponta para uma array é possível utilizar este ponteiro como se fosse a própria array. O operador de indexação, [], é igualmente pós-fixo no ponteiro e o valor de índice é um número entre 0 e *len*-1, onde len é o comprimento do array. Não há conservação dos limites de índices adotados no array original.

Listagem 31

```
01  program AplicacaoPonteiro4;
02  var
03     arr: array[1..5] of integer = (7,4,11,8,6);
04     pt: ^integer;
05     i: integer;
06  begin
07     pt := @arr[1];
08     for i := 0 to 4 do
09        write(pt[i], ' ');
10  end.
```

Na Listagem 31 pt recebe o endereço da primeira célula do array arr e dessa forma pode ter acesso ao array completo. Apesar da indexação no intervalo 1..5, o laço da Linha-08 varre a estrutura com índices no intervalo 0..4 haja vista que o acesso é realizado via ponteiro (Linha-09).

Intuitivamente percebe-se que este mecanismo de acesso de arrays via ponteiros não implica necessariamente no apontamento para a primeira célula. Poder-se-ia ter escrito,

```
pt := @arr[2];
```

Nesse caso, se o limite nulo inferior for mantido, o vetor manipulável será menor (em uma unidade). O laço,

```
for i := 0 to 3 do write(pt[i], ' ');
```

imprimirá os quatro elementos finais de arr. Já o laço,

```
for i := -1 to 3 do write(pt[i], ' ');
```

imprimirá todo conteúdo de arr.

Uma string em Pascal é um caso especial de array formada de caracteres. Quando um ponteiro de caractere aponta para o primeiro caractere de uma string, este ponteiro se comporta como a própria string podendo inclusive ser repassado diretamente aos comandos de saída write/writeln (observe que o mesmo não acontecem para outros tipos de arrays, apenas de caracteres).

Listagem 32

```
01  program ponteiros_e_strings;
02  var
03     s: string[20] = 'ola mundo!';
04     p: ^char;
05  begin
06     p := @s[1];
07     write(p);
08  end.
```

Contador de referência é uma metodologia utilizada para otimizar o gerenciamento de memória dinâmica associado aos ponteiros. Baseia-se na definição implícita de um contador para cada ponteiro que armazena o número de variáveis que se referem a ele. Quando este contador zera a memória apontada é desalocada. Dessa forma o programador não precisa se preocupar em desalocar memória dinâmica que requisitou pois isso ocorre automaticamente.

Na Listagem 32 o ponteiro p recebe o endereço da cadeia s (Linha-06). A impressão (Linha-07) é feita diretamente via ponteiro.

O tipo PChar em Pascal denota ponteiro de caractere (^char) com a conveniência de permitir atribuições diretamente. Internamente uma variável Pchar aloca e desaloca memória convenientemente utilizando **contadores de referência** de forma a manter strings de tamanho variante.

A Listagem 33 ilustra o uso de Pchar. Nas Linhas 06 e 08 são respectivamente inicializadas, com caracteres de constantes strings, as variáveis p de tipo PChar e s de tipo string. Quando p recebe o endereço da cadeia apontada por s (Linha-09), o contador de referência de 'teste de uso de PChar' é zerado e esta string é desalocada. A string 'uma string' possui agora contador de referência igual a dois e pode ser tanto acessada por s quanto por p (Linha-10). A saída deste programa é,

```
teste de uso do PChar

uma string
```

Listagem 33

```
01  program uso_de_pchar;
02  var
03     p: Pchar;
04     s: string;
05  begin
06     p := 'teste de uso do PChar';
07     writeln(p);
08     s := 'uma string';
09     p := @s[1];
10     writeln(p);
11  end.
```

4. Ponteiros em Módulos

Argumentos ponteiros em módulos, sejam procedimentos ou funções, são comumente utilizados para se referir a variáveis *fora do escopo* do módulo (ou seja, variáveis que não são locais ao módulo). Esse mecanismo permite que provisoriamente seja possível ler ou alterar variáveis *alheias*, sejam elas escalares ou vetoriais.

Argumentos ponteiros funcionam como quaisquer ponteiros permitindo aritmética e desreferenciamento à célula de memória endereçada. Há o inconveniente entretanto de, para se declarar um argumento ponteiro em um módulo Pascal, é necessário definir previamente um tipo ponteiro como em,

```
type PtInt = ^integer;

procedure proc(q: PtInt);              // O argumento de proc é um ponteiro
```

Uma aplicação usual de argumentos ponteiros são os chamados *argumentos saída*. Ao invés do ponteiro trazer o endereço de uma variável que se deseja *ler* o valor, ele traz o endereço de uma variável para onde se deseja *escrever* um valor. Esta estratégia quebra as restrições tanto da saída única das funções (pode-se usar vários ponteiros cada qual associado a uma saída diferente) quanto de ausência de saída formal nos procedimentos (argumentos ponteiros agem como saída nos procedimentos).

Listagem 34

```
01 program equacao_do_segundo_grau;
02 type
03    ptreal = ^real;
04
05 function eq2g(a,b,c: real; p, q: ptreal): boolean;
06 const TOL = 1.0e-8;
07 var delta: real;
08 begin
09    delta := b*b - 4*a*c;
10    if (abs(a) < TOL) or (delta<0) then begin
11       eq2g := false;
12       exit;
13    end;
14    p^ := (-b + sqrt(delta))/(2*a);
15    q^ := (-b - sqrt(delta))/(2*a);
16    eq2g := true;
17 end;
18
19 var
20    a,b,c,x1,x2: real;
21 begin
22    writeln('Coeficientes: ');
23    readln(a,b,c);
24    if eq2g(a, b, c, @x1, @x2) then
25       writeln('Raizes: ', x1:0:2, ' e ', x2:0:2)
26    else
27       writeln('Raízes não podem ser calculadas!');
28 end.
```

Seja o problema da solução do polinômio de segundo grau,

$$a\,x^2 + b\,x + c = 0$$

onde devem ser fornecidos os coeficientes *a*, *b* e *c* e calculadas as raízes, se existirem. Pelo *método de Báskara* as duas raízes reais são dadas por,

$$x = \frac{-b \pm \sqrt{\Delta}}{2a}, \text{ onde } \Delta = b^2 - 4ac$$

desde que *a* seja não nulo e seja não negativo (se *a* é não nulo e é negativo, as raízes são complexas).

O programa da Listagem 34 resolve o problema da equação de segundo grau. A função eq2g (das Linhas 05 a 17), que possui cinco argumentos, recebe os três coeficientes do polinômio de segundo grau via argumentos a, b e c. A função retorna três saídas: as duas primeiras através dos argumentos ponteiros p e q e a terceira via saída natural da função (boolean). A saída booleana retrata o sucesso da operação.

Quando true retorna significa que existem raízes reais e elas estão nas células de memória apontadas por p e q. Quando false retorna, não há raízes (seja por porque *a = 0* ou $\Delta<0$) e os valores apontados por p e q devem ser ignorados. O teste de nulidade de Δ (no programa representado pela variável delta) consiste da expressão abs(delta)<TOL que verifica se o módulo do valor em delta está suficientemente próximo de zero para ser considerado como tal (este é o teste mais consistente para verificar nulidade de valores de ponto flutuante).

Quanto menor o valor da constante TOL (tolerância) mais rigoroso será o teste de nulidade (na prática valores de tolerância demasiado pequenos são problemáticos. Por exemplo, para uma tolerância na ordem de 10^{-15}, 0.0000000001 não passaria num teste de nulidade o que, em muitos casos, é incoerente). Se um dos testes da Linha-10 falhar o status da variável interna eq2g é mudado para false e a função é manualmente encerrada (exit, Linha-12).

Do contrário as raízes são calculadas e, por desreferenciamento, registradas nas células apontadas por p e q (Linhas 14 e 15). Antes do encerramento da função, eq2g (variável interna) é mudada para true (Linha-16) indicando que raízes foram calculadas com êxito.

A chamada de eq2g no programa principal acontece dentro de uma estrutura de decisão (Linha-24) ilustrando como é possível tomar diretamente uma decisão a partir da saída de uma função. Quando qualquer um dos blocos da decisão bidirecional for executado (Linhas 25 ou 27), eq2g já terá sido executada. A chamada a eq2g repassa como dois últimos parâmetros os endereços das variáveis globais x1 e x2 que ao final portarão as raízes, se existirem.

Uma importante habilidade que ponteiros permitem a módulos é o acesso a arrays fora de escopo (não locais). Para estruturar isso, repassa-se ao módulo o endereço da primeira célula do vetor (que pode ser ou não tipado) e um inteiro que represente o comprimento da array ou a memória em bytes que ela ocupa. Estando isso disponível nos parâmetros do módulo, o acesso às

células da array externa se fará com a pós-fixação do operador de indexação no ponteiro. Como visto na Sessão - 3 , o ponteiro se portará como se fosse a array original (lembre-se que neste caso o índice inferior sempre é zero). No protótipo,

```
type PtReal = ^real;
function Max(vec: PtReal; len: longint): real;
```

a função Max recebe, através do ponteiro vec, o endereço da primeira célula de uma array cujo comprimento é repassado por len. Esta função deve retornar o maior valor contido na array endereçada por vec. O programa da Listagem 35 contextualiza a função Max. Nesta função a primeira hipótese de retorno é o primeiro elemento da array (Linha-09). O laço da Linha-10, associado ao teste aninhado na Linha-11, faz a variável local Max ser atualizada constantemente para o maior valor até a iteração corrente. Com o encerramento do laço Max conterá o valor procurado. No programa principal, a array global x é carregada com valores aleatórios ao mesmo tempo em que é impresso na tela (Linhas 20 a 23). A chamada a Max é efetuada de dentro do writeln (Linha-25) sendo os argumentos o endereço da primeira célula de x (@x[1]) e seu comprimento (length(x)). Um exemplo de saída do programa é,

```
23 82 50 49 81 90 80 73 17 32 79 45 86 56 39 89 57 49 1 26
Valor máximo = 90
```

Listagem 35

```
01  program extremos;
02
03  type
04      PtReal = ^real;
05
06  function Max(vec: PtReal; len: integer): real;
07  var i: integer;
08  begin
09        Max := vec[0];
10        for i:= 1 to len-1 do
11            if vec[i]>Max then
12            Max := vec[i];
13  end;
14
15  var
16       x: array[1..20] of real;
17       i: integer;
18  begin
19      randomize;
20      for i := 1 to length(x) do begin
21          x[i] := random(100);
22          write(x[i]:0:0, ' ');
23      end;
24      writeln;
25      writeln('Valor máximo = ', Max( @x[1], length(x) ):0:0);
26  end.
```

5. Alocação Dinâmica

Novas variáveis podem ser criadas em tempo de execução através do mecanismo de *alocação dinâmica*. Estas variáveis não possuem nomes e dependem de ponteiros para serem manipuladas. Ponteiros tipados possuem em Pascal duas abordagens de alocação, a *escalar* e a *vetorial,* que são estudadas nas Sessões 5.1 e 5.2 . A alocação com ponteiros não tipados é estudada Sessão 5.3 .

5.1. Alocação Dinâmica Escalar

Na abordagem escalar uma única célula de memória, de tamanho igual ao do tipo base, é alocada. Como não há operadores em Pascal que proporcionem alocação dinâmica, a unidade system disponibiliza os comandos new e dispose para respectivas alocação e desalocação escalar.

Listagem 36

```
01  program Alocacao_Dinamica_1;
02  var
03      i: integer;
04      p: ^integer;
05  begin
06       i := 120;
07       new(p);
08       p^:= i;
09       i := i div 2;
10       writeln(i, ' ', p^);
11       dispose(p);
12  end.
```

O programa da Listagem 36 ilustra o uso da alocação dinâmica escalar. Uma nova variável é alocada na Linha-07. O comando new recebe como argumento o ponteiro que armazenará o endereço da variável dinâmica criada em heap. Qualquer desreferenciamento feito com p, depois da chamada de new, afetará a nova variável sem nome criada dinamicamente. Na Linha-08 ocorre uma cópia do conteúdo em i para a variável apontada por p. Não há relação, neste exemplo, entre i e p. Ao fazer i receber metade do próprio valor (Linha-09) e em seguida imprimir-se i e p^ obtém-se,

```
60 120
```

Na Linha-13 dispose, que também recebe um ponteiro como argumento, libera a memória alocada por new.

Listagem - 37

```
01  program Alocacao_dinamica_2;
02  type
03      ponto = record
04          x,y,z: real;
05      end;
06  var
07      q: ^ponto;
08  begin
09      new(q);
10      q^.x := 3.0;
11      q^.y := 1.0;
12      q^.z := 0.5;
13      dispose(q);
14  end.
```

As mesmas regras se mantêm para variáveis de tipo estrutura heterogênea. O operador de resolução de campo (.) mantém a mesma sintaxe. A Listagem 37 ilustra a alocação escalar com tipo estrutura heterogênea. Neste programa o tipo record precisa ser definido (Linha-03) para que possa atuar como tipo base na declaração do ponteiro q (Linha-07). Cada campo do record precisa de seu próprio desreferenciamento (Linhas 10, 11 e 12).

5.2. Alocação Dinâmica Vetorial

Na abordagem vetorial com ponteiros tipados a memória alocada é representada por uma ou mais células contíguas de tamanho igual ao do tipo base (ou seja, uma array). A unidade system disponibiliza, para tal abordagem, os comandos getmem para a alocação e freemem para a desalocação. O comando `getmem` pode ser usado com a sintaxe,

```
getmem(<ponteiro>, <memória_em_bytes>);
```

ou ainda,

```
<ponteiro> := getmem(<memória_em_bytes>);
```

onde, em ambas, o ponteiro recebe o endereço de um bloco de memória cujo tamanho em bytes é passado como argumento. É comum utilizar este comando associado com sizeof para facilitar os cálculos. Por exemplo em,

```
getmem(pt, sizeof(integer)*10); // se integer tem 4-bytes
                                // então 40-bytes são alocados
```

aloca um bloco de memória que compota dez inteiros e atribui para pt o endereço.

O comando freemem é similar a dispose contendo como argumento único o ponteiro para a área de memória que deve ser liberada. No exemplo,

```
freemem(pt);
```

a memória apontada por pt é liberada.

Listagem 38

```
01  program Alocacao_Dinamica_3;
02  var
03     i, j, n: integer;
04     p, q: ^integer;
05  begin
06     write('Total: ');
07     read(n);
08     i := 0;
09     j := 0;
10     getmem(p, sizeof(integer)*n);
11     q := p;
12     while j<n do begin
13      if (i mod 3 = 0) and (i mod 7 <> 0) then begin
14           q^ := i;
15         q := q + sizeof(integer);
16         inc(j);
17       end;
18       inc(i);
19    end;
20    for i := n-1 downto 0 do begin
21       q := p + i * sizeof(integer);
22       write(q^, ' ');
23    end;
24    freemem(p);
25 end.
```

O programa da Listagem 38 ilustra o uso de getmem e freemem. O programa imprime em ordem decrescente os primeiros n inteiros (n fornecido pelo usuário) que são divisíveis por 3 mas não são divisíveis por 7. Para armazenar estes valores é alocado, na Linha-10, o espaço necessário em *heap*.

Para acessar individualmente cada uma das n células que compõe o bloco, utiliza-se um ponteiro auxiliar q. Há dois contextos de uso de q no programa: o da Linha-15, engajado na inicialização da array e o da Linha-21, utilizado na impressão dos dados.

Na Linha-15 o valor do endereço em q é incrementado de sizeof(integer) em cada iteração do laço da Linha-12 cujo teste da Linha-13 obtém sucesso. Haja vista que o valor inicial de q é o mesmo de p (Linha-11), então q passa por cada célula da array dinâmica (ver Figura - 9). Assim as atribuições da Linha-14 armazenam os valores encontrados em suas posições apropriadas na array. Já na Linha-21 as posições apropriadas são calculadas diretamente por uma expressão completa causando o mesmo efeito de acesso.

Os comandos inc e dec, unidade system, fazem respectivamente a incrementação e decrementação numérica de variáveis ordinais (inteiros, caracteres e enumerações). Estes comandos possuem versões de um e dois argumentos. Na versão de um argumento o total incrementado (ou decrementado) é um. Na versão de dois argumentos este valor corresponde ao segundo argumento. Assim, se x contém 3, as chamadas inc(x) e dec(x) fazem x mudar para 4 e 2 e as chamadas inc(x,3) e dec(x,2) para 6 e 1 respectivamente.

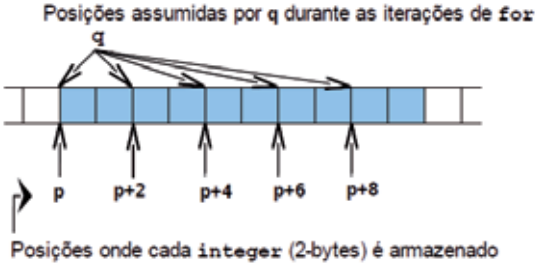


Figura 9

O operador de indexação, [], pode ser usado com ponteiros tipados que apontam para vetores dinâmicos. O resultado é exatamente a mesma sintaxe de arrays estáticas. Por exemplo,

```
q := p + sizeof(integer)*i;
q^ := 21;
```

é equivalente a,

```
p[i] := 21;
```

Essa sintaxe dispensa o uso na variável auxiliar.

Listagem 39

```
01  program Alocacao_Dinamica_4;
02  var
03     i, j, n: integer;
04     p: ^integer;
05  begin
06     write('Total: ');
07     read(n);
08     i := 0;
09     j := 0;
10     getmem(p, sizeof(integer)*n);
11     while j<n do begin
12        if (i mod 3 = 0) and (i mod 7 <> 0) then begin
13           p[j] := i;
14           inc(j);
15        end;
16        inc(i);
17     end;
18     for i := n-1 downto 0 do write(p[i], ' ');
19     freemem(p);
20  end.
```

Reescrevendo-se a Listagem - 38 utilizando o operador de indexação, é obtido o programa da Listagem 39.

Em alguns compiladores Pascal o uso do operador de indexação pós-fixado a ponteiros não é diretamente suportado. Se este for o caso a saída é alocar memória para um ponteiro de um array de uma única célula. A Listagem 40 ilustra o uso desta técnica. O programa imprime em ordem inversa os primeiros dez termos da série de Fibonacci. O tipo array na Linha-08 permite a criação de arrays de apenas uma célula (indexação de zero a zero) e vec (Linha-05) é um ponteiro deste tipo. A memória alocada para vec (Linha-10) deve ser a do vetor completo (neste caso um total de max células). Como o tipo IntVec restringe os limites entre zero e zero, o compilador emitirá advertências sobre o uso de limites fora desta faixa (isso não compromete o funcionamento programa). Nas Linhas 11, 12, 14 e 16 é notório que o uso do vetor dinâmico exige o circunflexo de acordo com a sintaxe,

```
<ponteiro>^[<indice>]
```

que só é necessária no contexto apresentado dos ponteiros de arrays.

Listagem 40

```
01 program Alocacao_Dinamica_5;
02 type
03    IntVec = array[0..0] of integer;
04 var
05    vec: ^IntVec;
06    i: integer;
07 const
08    max = 10;
09 begin
10    vec := getmem( sizeof(integer)*max );
11    vec^[0] := 1;
12    vec^[1] := 1;
13    for i := 2 to max-1 do
14       vec^[i] := vec^[i-1] + vec^[i-2];
15    for i := max-1 downto 0 do
16       write(vec^[i], ' ');
17    freemem(vec);
18 end.
```

5.3. Usando Ponteiros Não Tipados

Ponteiros não tipados podem conter endereços para quaisquer variáveis. Isso significa que, se o ponteiro for a única variável disponível num dado contexto, então não será possível determinar o tipo da informação apontada. Na maioria dos casos é necessário fornecer alguma informação adicional como tamanho da célula de memória apontada.

Para ilustrar isso considere o problema geral de *zerar* todos os bytes de uma estrutura qualquer, independente de ser um escalar (como inteiro ou real), uma estrutura heterogênea ou uma array. O procedimento zerar da Listagem 41 resolve a questão. O argumento p recebe o endereço de uma estrutura qualquer enquanto size recebe seu tamanho em bytes. Utilizando o laço da Linha-05 e a expressão p+i na Linha-06, o ponteiro p passa por cada um dos bytes que forma a estrutura (como uma array de bytes) e por desreferenciamento zera-se o conteúdo ali (Linha-07). O número de passos do laço é o total de bytes da estrutura.

Listagem 41

```
01  procedure zerar(p: pointer; size: longint);
02  var i: longint;
03      b: ^byte;
04  begin
05     for i := 0 to size-1 do begin
06        b := p + i;
07        b^ := 0;
08     end;
09  end;
```

Se x é uma variável real e precisa ter os bytes zerados pelo procedimento zerar, deve-se escrever,

```
zerar(@x, sizeof(real));
```

Se y for uma variável de uma estrutura heterogênea chamada Reg, seus bytes podem ser zerados com,

```
zerar(@y, sizeof(Reg));
```

Se vec for uma array de inteiros que precisa ter os bytes zerados então deve-se escrever,

```
zerar( @vec[1], sizeof(integer)*length(vec) );
```

Em caso de alocação dinâmica, os comandos getmem e freemem, da unidade system, permitem também a alocação de memória utilizando pomeiros não tipados. A memória alocada com ponteiros não tipados pode ser manipulada utilizando **aritmética de ponteiros**. Por exemplo, para criar um bloco de memória que armazene 25 valores reais de 4-bytes utiliza-se,

Aritmética de Ponteiros: termo utilizado para se referir às operações de soma e subtração de ponteiros com deslocamentos de memória (medidos em bytes). Esta aritmética não prevê operações *entre* ponteiros por ser incoerente. Seu grande objetivo é referenciar corretamente células de memória vizinhas a uma célula dada.

```
getmem(p, 100); // 25x4 = 100-bytes
```

onde p é não tipado. Para aplicar a aritmética deve-se utilizar um ponteiro auxiliar de real conforme expressão,

```
q := p + i*4;
```

onde q é de tipo ^real e i inteiro. Variando i entre 0 a 24 faz-se q referir-se a cada uma das 25 posições de células dentro do bloco alocado. Para acessar cada célula deve-se desreferenciar q logo após receber o resultado na expressão. No exemplo,

```
for i := 0 to 24 do begin
    q := p + i*4;
    q^ := 1000/(i+1);
end;
```

o bloco alocado via p recebe os valores 1000.0, 500.0, 333.4, 250.0 … 40.0.

Uma alternativa às expressões oriundas da aritmética de ponteiros é a coerção do ponteiro não tipado em tipado. Neste caso retoma-se o operador de indexação []. Reescrevendo-se o exemplo anterior tem-se,

```
q := p;          // Coerção

for i := 0 to 24 do

    q[i] := 1000/(i+1);
```

onde p é não tipado e q é de tipo ^real. A coerção faz q funcionar como uma array de 25 valores real.

6. Problemas com Ponteiros

Os principais problemas causados por ponteiros são,

- Ponteiros Pendurados
- Vazamento de Memória

Um ponteiro pendurado é aquele cujo endereço armazenado deixou de ser seguro. Após uma chamada a dispose ou freemem a área de memória apontada é liberada, mas o valor de endereço armazenado pelo ponteiro é mantido. O desreferenciamento acidental deste ponteiro a partir deste ponto (sem reapontamento ou realocação) caracterizará acesso ilegal de memória. Se o programador não puder redefinir o valor do ponteiro logo após a liberação de memória então aconselha-se fazê-lo apontar para *lugar nenhum,* ou seja,

```
...

dispose(p);

p := nil;                // ponteiro novamente seguro
```

Vazamento de memória (*memory leak*) é a perda de acesso a um bloco de memória alocado dinamicamente. Isso acontece quando o ponteiro que aponta para a área dinâmica alocada passa acidentalmente a apontar para outra parte da memória sem desalocação prévia. O bloco de memória sem referência é chamado *memória perdida em heap* e não pode ser reutilizado até que o programa seja encerrado.

Listagem 42

```
01  program Vazamento_de_Memoria;
02  var
03     p: ^integer;
04     i: integer;
05  begin
06     new(p);
07     p := @i;
08     dispose(p);
09  end.
```

O programa da Listagem 42 Ilustra o vazamento de memória. A memória alocada via p (Linha-06) é perdida em *heap* quando p é redefinido para apontar para i (Linha-07). A chamada a dispose (Linha-08) não surte efeito nem sobre a área alocada dinamicamente (ela está *perdida* em *heap*) nem sobre a de i que p agora aponta (por ter sido alocada estaticamente). A Figura - 10 esquematiza o vazamento de memória implementado na Listagem 42.

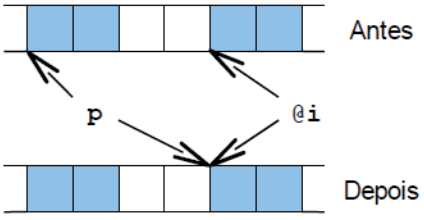


Figura 10

Um exemplo crítico de vazamento de memória, conhecido como *ladrão de memória*, usa um laço infinito para gerar incontáveis blocos perdidos em *heap*. Em Pascal implementa-se,

```
while true do new(p);
```

Não é recomendada a execução do ladrão de memória.

7. Estudo de Caso: Lista Escadeada Elementar

O primeiro passo para implementar uma lista encadeada elementar em Pascal é a definição de uma estrutura heterogênea para os nodos da lista. Considerando uma lista de valores inteiros então o tipo nodo proposto é,

Lista Encadeada Elementar: é uma lista de objetos, denominados nodos, cujo tipo corresponde a um tipo estruturado heterogêneo contendo um campo da chave e outro de ponteiro do próprio tipo estrutura heterogênea em questão. Isso permite que o primeiro nodo da lista contenha uma chave e o endereço para o segundo nodo que contém outra chave e o endereço para o terceiro nodo e assim por diante. O último nodo da lista é aquele que contém uma chave e um endereço para *parte alguma*.

```
pnodo = ^nodo;

nodo = record

    chave: integer;

    prox: pnodo;

end;
```

Observe a necessidade de definição de um tipo ponteiro de nodo antes da definição da própria estrutura. Isso levanta duas questões. A primeira é *para que?* e a segunda é *como é possível?*. A resposta da primeira é pela necessidade de definição de prox que conterá uma referência ao próximo nodo da lista e dinamicamente isso deve ser feito via um endereço de nodo. A segunda é na realidade uma nova faceta do Pascal. Apesar de ser apenas definido na linha seguinte, o tipo nodo pode *sim* ser utilizado para definir pnodo. Esse fenômeno é conhecido como *declaração prematura* e é exclusiva dos tipos ponteiros.

Para definir a lista propriamente dita propomos um outro tipo estrutura heterogênea denominado lista como segue,

```
lista = record

    raiz: pnodo;

    total: integer;

end;

plista = ^lista;
```

Este novo tipo mantém um ponteiro raiz para o primeiro nodo da lista e o total de nodos listados. Apesar de não serem visíveis diretamente nessa estrutura, os demais nodos estarão presentes pois o primeiro aponta para o segundo, que aponta para o terceiro e assim sucessivamente. O tipo plista é necessário para possibilitar a módulos afetarem uma lista que recebam como argumento. Por exemplo para inicializar uma lista deve-se utilizar o procedimento,

```
procedure Init(Q: plista);

begin

   Q^.raiz := nil;

   Q^.total := 0;

end;
```

init zera uma lista tanto em nodos quanto em contagem. Pode ser chamado assim,

```
var
    L: lista;
begin
    init(@L);
    ...
```

Para inserir uma nova chave numa lista propõe-se o seguinte algoritmo: (i) constrói-se dinamicamente um novo nodo contendo a chave de inserção; (ii) Faz-se o prox desse novo nodo apontar para o primeiro nodo da lista, ou seja, raiz, (iii) faz-se raiz da lista apontar para este nodo e (iv) incrementar o campo total da lista em uma unidade para indicar que um novo nodo foi inserido.

Assim a última chave inserida é sempre a primeira da lista e a proporção que novas chaves são inseridas as mais antigas vão ficando mais distantes de raiz. O procedimento da Listagem 43 implementa a inserção. A lista é repassada via ponteiro Q e a chave de inserção via inteiro x. As etapas são marcadas como comentários.

Listagem 43

```
01  procedure inserir(Q: plista; x: integer);
02  var p: pnodo;
03  begin
04      new(p);
05      p^.chave := x;                          // (i)
06      p^.prox := Q^.raiz;          // (ii)
07      Q^.raiz := p;                // (iii)
08      inc(Q^.total);         // (iv)
09  end;
```

Para imprimir em saída padrão o conteúdo de uma lista deve-se varrê-la desde o nodo raiz até o último nodo (aquele cujo prox vale nil). Para tanto utiliza-se um ponteiro auxiliar de pnodo e um laço pré-teste. O procedimento imprimir na Listagem 44 Implementa o processo. O ponteiro auxiliar p é inicializado com o valor de raiz da lista apontada por Q (Linha-04). O laço na Linha-05 verifica se o final da lista já foi alcançado ao passo que a atribuição na Linha-07 provoca um salto de um nodo para outro. Cada nodo visitado por p tem a chave impressa na Linha-06.

Para eliminar toda uma lista encadeada elementar da memória propomos o seguinte algoritmo: (i) Se o campo raiz é diferente de nil então siga para o passo ii, do contrário termine; (ii) Fazer uma cópia do endereço em raiz num ponteiro auxiliar de pnodo; (iii) Fazer prox de raiz se tornar o novo raiz; (iv) aplicar dispose no ponteiro auxiliar e voltar para i. A Listagem 45 ilustra o

processo e os passos são marcados como comentários. A Linha-09 adicionalmente zera o campo de contagem de nodos trazendo a lista ao estado inicial de vazia.

Listagem 44

```
01  procedure imprimir(Q: plista);
02  var p: pnodo;
03  begin
04      p := Q^.raiz;
05      while p <> nil do begin
06          write(p^.chave, ' ');
07          p := p^.prox;
08      end;
09  end;
```

Listagem 45

```
01  procedure limpar(Q: plista);
02  var p: pnodo;
03  begin
04      while Q^.raiz <> nil do begin       // i
05          p := Q^.raiz;                   // ii
06          Q^.raiz := p^.prox;             // iii
07          dispose(p);                     // iv
08      end;
09      Q^.total := 0;
10  end;
```

Síntese da Parte 2

No Capítulo - 3 foi tanto apresentada a sintaxe básica de definição de tipos personalizados quanto as estruturas nativas do Pascal que permitem a modelagem de objetos enumerados (enumerações), de objetos segmentados por campos (estruturas heterogêneas) e objetos agrupáveis (conjuntos). No Capítulo - 4 foram introduzidos e contextualizados os ponteiros. Foram apresentadas que relações eles mantêm com outras construções da linguagem como arrays e módulos assim como a importante participação que exercem no âmbito da alocação dinâmica de memória. Problemas com ponteiros são também abordados. O capítulo encerra com o estudo de caso da lista encadeada elementar.

Atividades de avaliação

1. Reescreva o programa da Listagem-4.7 de forma que seja possível, com as informações de saída, saber o tipo de problema que inviabilizou o cálculo das raízes, quando esse for o caso.
2. Construir módulo que inverta a sequência dos caracteres de uma string (use Pchar).
3. Construa módulo que receba um vetor e retorne o maior e menor valor existente.
4. Construa módulo que receba um vetor e retorne os dois maiores valores existentes.
5. Escrever duas versões de programa, uma utilizando arrays dinâmicas e outra utilizando alocação dinâmica, para resolver o problema de receber uma quantidade arbitrária de notas de alunos (esta quantidade de notas também é entrada) e determine o valor do desvio padrão da amostra. O desvio padrão de uma amostra $x_1, x_2, x_3, \ldots, x_n$ é dada por, $\sqrt{\sum_{i=1}^{n} (x_i - \bar{x})^2 / n}$, onde x_i se refere a i-ésima nota e $\bar{x}$ à média de todas as notas.
6. Seja P um ponteiro que aponta para um bloco de memória alocado dinamicamente. Se P precisar apontar para um bloco de memória maior, sem perder as informações para as quais já aponta, deve-se: (I) alocar o novo espaço, maior que o anterior; (II) copiar todo conteúdo do bloco apontado por P para a nova memória alocada; (III) desalocar o espaço apontado por P; (IV) apontar P para a nova memória alocada. O processo de cópia deve ser de tal forma que independa do tipo de informação apontada. Construir módulo com este fim e contextualizá-lo num programa.
7. Crie uma estrutura vector (usando estruturas heterogêneas) que mantenha dois campos: um ponteiro de real data e um inteiro n. O campo data refere-se a um vetor de real cujo comprimento é o valor do campo n. Construa também módulos que permitam as seguintes operações com variáveis de tipo vector: (a) construtor (recebe um comprimento qualquer e retorna um vector novo com tal comprimento e com todos os elementos de data iguais a zero); (b) valor modular (retorna raiz quadrada da somatória dos quadrados dos elementos de data); (c) normalizador (modifica um vector de forma que todos os elementos em data sejam divididos pelo valor modular); (d) destrutor (recebe um vector, desaloca a memória apontada por data e zera seu comprimento).

8. Implementar para a lista encadeada elementar as operações seguintes: busca de uma chave, remoção de uma chave e reversão da lista (dispô-la no sentido contrário).

Referências

CANNEYT, Michaël Van, **Reference guide for Free Pascal**, Document version 2.4 , March 2010

http://www.freepascal.org/docs-html/

http://en.wikipedia.org/wiki/Tower_of_Hanoi

Parte 3

Modularização

Capítulo 5

Modularização

Objetivos:

- A modularização, estudada no Capítulo-1 desta unidade, é uma técnica que permite a segmentação de um programa em unidades menores mais fáceis de gerenciar. São as funções e os procedimentos. Para estudá-la com mais detalhes o capítulo aborda noções sobre controle e processo bem como um estudo mais aprofundado sobre o mecanismo de passagem de argumentos introduzido no Capítulo-1. O capítulo ainda cobre outros importantes conceitos relacionados como a recursividade e o aninhamento modular. O Capítulo-2 finaliza a unidade apresentado os recursos que o Pascal disponibiliza para a manipulação de arquivos. Isso inclui desde fundamentos sobre o funcionamento da memória secundária até recursos e comandos que permitirão a implementação de programas que manipulem arquivos.

1. Noções de Controle e Processo

Sistemas operacionais modernos permitem a execução simultânea de vários programas. Esta simultaneidade é na verdade aparente porque existem poucos processadores (ou mesmo somente um) que devem resolver todas as solicitações dos programas em execução. O sistema operacional cria uma fila de solicitações dos programas (chamados comumente de *processos*) para que sejam resolvidas uma a uma pela primeira CPU livre.

As solicitações entram na fila conforme o momento que são invocadas de forma que pendências de diversos programas podem estar presentes. A alta velocidade de processamento faz os usuários não *sentirem* a permutação de atendimento de solicitações entre programas. O multiprocessamento se resume então no *compartilhamento do tempo* de trabalho das CPUs entre fragmentos de tarefas nos quais os processos são quebrados.

O Compartilhamento de Tempo ou *Time Sharing* é um recurso das CPUs modernas que surgiu pela necessidade de execução concorrente de várias aplicações. Sistemas que tiram proveiro deste recurso são denominados *Sistemas Multitarefas*

O compartilhamento de tempo *entre* processos ocorre de forma similar *dentro* de cada processo onde tarefas internas competem pelo tempo de CPU(s) da aplicação. Estes *subprocessos* competindo entre si são denominados *threads*. Se mais de um *thread* está envolvido diz-se tratar-se uma aplicação *multithreading*.

Cada *thread* (caso mais de um esteja envolvido) segue um *fluxo de execução* que começa num programa principal e pode ser *desviado* para um módulo (função ou procedimento). Dessa forma apenas um dado módulo (ou o programa principal), num *thread*, executa por vez. Diz-se neste contexto que aquele módulo (ou o programa principal) possui o *controle* do fluxo. Quando ocorre desvio de controle, ou seja, um módulo é chamado, este passa a possuir o controle e o chamador deve aguardar sua finalização para resgatá-lo de volta. Um programa termina sua execução quando o controle, que está com o programa principal, é retornado ao sistema operacional.

Os programas em Pascal estudados neste livro executam sobre um único *thread* de forma que o fluxo de controle está exatamente com foco num único módulo (ou no programa principal) num dado instante da execução.

O controle de uma função ou procedimento pode forçadamente ser devolvido ao seu chamador, ou seja, antes do término da execução completa de seu código. O comando exit, unidade system, efetua esta tarefa. Por exemplo, no procedimento,

Os comandos round, trunc, frac e int, unidade system, são comandos utilizados para operar números reais. Todos possuem apenas um argumento de entrada. Round e trunc retornam respectivamente o arrendondamento e o piso do valor de entrada, ambos inteiros. Frac retorna a parte fracionária da entrada, também real e Int retorna uma versão real da parte inteira da entrada. Os exemplos round(4.8), trunc(4.8), Frac(4.8) e Int(4.8) devolvem respectivamente 5, 4, 0.8 e 4.0.

```
procedure print(n: integer);
begin
    if n<0 then exit;
    while n>0 do begin
        write(n);
        n = n div 3;
    end;
end;
```

quando valores negativos são repassados o procedimento chama exit devolvendo imediatamente o controle ao chamador. Do contrário uma sequencia de valores é impressa por ação da execução do laço while.

Como visto anteriormente, funções possuem uma variável interna com mesmo nome da função e cujo valor antes da devolução de controle ao chamador é justamente seu retorno. Assim a presença de um exit numa função é estratégica quando o valor de retorno já foi determinado, mas o final da função ainda não foi alcançado (linha contendo o último end; da função) . No exemplo,

```
procedure print(n: integer);
begin
    if n<0 then exit;
    while n>0 do begin
        write(n);
        n = n div 3;
    end;
end;
```

caso a entrada seja um valor negativo, então a função deve retornar imediatamente ao chamador com retorno zero. Para implementar esse comportamento associou-se uma estrutura de decisão (que verifica a possível negatividade de n), uma atribuição direta do valor 0 a fnc (variável interna) e uma chamada a exit.

Um programa também pode devolver diretamente o controle ao sistema operacional independendo do módulo que esteja executando. Isso representa *matar* o programa e em Pascal é conseguido com uma chamada ao comando halt, unidade `system`.

O *piso* de um valor real x é o maior inteiro menor ou igual a x. Representa-se por.O *teto* de um valor real x é o menor inteiro maior ou igual a x. Representa-se por.

2. Parâmetros Formais e Reais

No Capítulo-1 as funções e os procedimentos foram introduzidos como módulos construtivos de um programa. De fato estes devem trabalhar como subprogramas cujas entradas são os parâmetros e a saída é o retorno (para o caso de funções). Se um dado módulo possui argumentos, eles devem ser fornecidos no momento da chamada seguindo a ordem de declaração e utilizando tipos apropriados. As variáveis utilizadas na definição dos argumentos de um módulo são chamadas de *parâmetros formais* do módulo. As constantes e/ou variáveis repassadas numa chamada de módulo, e que alimentarem os parâmetros formais, são chamados *parâmetros reais*.

Seja, por exemplo, o protótipo (cabeçalho) de função,

```
function teste(x: integer; y: real): boolean;
```

então a chamada seguinte é legal,

```
b := teste(5, 6.0);
```

onde b é booleano. Aqui x e y são parâmetros formais e 5 e 6.0 são parâmetros reais. A presença da parte decimal no segundo argumento ajuda o compilador a reconhecer o valor como real. Se os valores repassados como parâmetros reais possuem tipos diferentes dos parâmetros formais, o compilador tenta fazer a conversão, como no exemplo,

```
teste(5, 6);
```

Neste caso o 6 inteiro é convertido para um valor real. Se for porém a situação seguinte,

```
teste(5.0, 6.0);
```

causará um erro de compilação porque 5.0 (que é real) não pode ser transformado em inteiro diretamente. Estas situações caracterizam a *coerção* em Pascal. Coagir um valor significa transformá-lo noutro tipo. A coerção pode ser de *estreitamento* ou de *alargamento*. No estreitamento o tipo final do valor é menos expressivo (possui menor memória ou representa faixas mais restritas de valores) e por isso pode ocorrer perda de dados. Exemplos são, de real para integer, de longint para char, extended para real e etc. No alargamento ocorre exatamente o inverso e logo não há perda de dados.

Em Pascal a coerção de alargamento entre parâmetros reais e formais é normalmente aceita pelo compilador desde que haja compatibilidade. Entretanto a coerção de estreitamento frequentemente reporta um erro. A rigor as restrições de coerção também ocorrem em atribuições explícitas como,

```
x := y;
```

se x e y forem de tipos distintos ou for necessária uma coerção de estreitamento, o compilador reportará um erro de conversão.

A passagem de valores entre parâmetros formais e reais é conhecida como *atribuição implícita*.

3. Modos de Passagem de Argumentos

Há em Pascal três modos distintos de passagem de argumentos em módulos (sejam procedimentos ou funções). São,

- Por Valor (ou cópia)
- Por Endereço
- Por Referência

Na passagem por valor os parâmetros reais têm seus valores copiados para os parâmetros formais (como a maioria dos exemplos vistos até aqui). Este modelo isola ao máximo o contexto de quem chama do contexto de quem é chamado.

Na chamada por endereço o parâmetro formal é de tipo ponteiro do tipo do parâmetro real. Este modelo permite que um dado módulo tenha acesso a informações que estão em outras partes do programa. Na prática o chamador concederá ao argumento ponteiro do módulo chamado acesso a uma de suas variáveis locais, a uma variável global ou ainda a um ponteiro que detém. Por exemplo em,

```
type

    ptint = ^integer;

procedure inc(q: ptint);

begin

    q^:= q^ + 1;

end;
```

o argumento formal q recebe o endereço de um inteiro que está fora do procedimento inc e, por desreferenciamento, permite mudar a célula de memória apontada (neste exemplo aumenta de uma unidade o inteiro com endereço em q). Em Pascal é necessário antes predefinir o tipo ponteiro (uso de type) antes de utilizá-lo como tipo de um argumento formal (neste exemplo ptint é definido previamente). Como o argumento real precisa ser um endereço, então uma chamada válida a inc é,

```
x := 56;                // x é inteiro

inc(@x);

write(x);               // 57
```

Como já visto no Capítulo - 4, a prefixação de @ numa variável representa semanticamente o endereço desta variável. Repassar apenas o x significa naturalmente uma tentativa inválida de coerção (inteiro para endereço de inteiro) reportada com erro pelo compilador. Como o procedimento inc possui permissão de mudar o conteúdo da célula de memória com endereço em q, então o valor em *x* é de fato incrementado da unidade.

Ponteiros, em argumentos formais, são comumente utilizados para se referir a arrays (em aplicações que lidam com arrays muito grandes seria problemática a passagem de argumentos por valor pois demandaria muito tempo em cópia de dados além do desperdício de memória). Uma função, por exemplo, para procurar o valor máximo em uma array de inteiros pode ter protótipo,

```
type ptint = ^integer;

function max(arr: ptint; int len): integer;
```

Neste caso como deve ser repassado uma array de inteiros, então o primeiro argumento é um ponteiro de inteiro (irá receber o endereço para o primeiro inteiro da array). O comprimento da array deve ser repassada como segundo argumento (por valor) e as células individuais poderão ser acessadas pelo operador pós-fixo [] (os valores de índice estão entre 0 e len-1). Para usar a função max deve-se repassar naturalmente o endereço da primeira célula de uma array de inteiros existente como,

```
var
    data: array[1..10] of integer;
    x: integer;
begin
    {...}
    x := max(@data[1], 10);
    {...}
```

Esta sintaxe é entretanto problemática porque depende do comprimento da array e do uso explícito de ponteiros. Uma alternativa é definir o parâmetro formal com a mesma sintaxe de uma array dinâmica. Exemplo,

```
procedure teste(x: array of integer);
{...}
```

O efeito é o uso da passagem por endereço, mas sem necessidade de fornecer o comprimento ou usar operadores de ponteiros. A Listagem 46 Ilustra o uso deste recurso. Na Linha-03, data não é uma array dinâmica mas sim um ponteiro para uma array. Seu uso é similar entretanto a uma array dinâmica incluindo índice limite inferior igual a zero (Linha-06). A array definida na Linha-12 é repassada a max (Linha-14) sem o uso dos operadores @ e [].

Listagem 46

```
01  program maximo;
02
03  function max(data: array of integer): integer;
04  var k: integer;
05  begin
06      max := data[0];
07      for k := low(data) to high(data) do
08          if data[k] > max then max := data[k];
09  end;
10
11  const
12      arr: array[1..10] of integer = (23,41,64,11,9,2,29,88,76,52);
13  begin
14      writeln( max(arr) );        // 88
15  end.
```

A terceira e última forma de passar um argumento para um módulo Pascal é utilizando *referências*. Uma referência é nada mais que um ponteiro camuflado. Elas permitem que um argumento formal se torne um *apelido*

do parâmetro real sem uso explícito de ponteiros ou seu operadores. Assim qualquer modificação realizada sobre o argumento formal estará de fato alterando o argumento real. Esta estrada de duas vias obriga então que argumentos reais, cujos argumentos formais são referências, sejam necessariamente variáveis. Para definir um argumento de módulo como uma referência basta torná-lo membro de uma sessão interna var. Nos exemplos de protótipos,

```
procedure teste_1 (var x: integer);

function teste_2 (var a, b: real): boolean;

procedure teste_3 (y: integer; var ch: char);
```

os argumentos formais x, a, b e ch são exemplos de referências. Observe em teste_3 que y não é referência pois não está numa sessão interna var. Para demonstrar o uso de referências considere o procedimento,

```
procedure dobrar(var x: integer);

begin

    x := x*2;

end;;
```

O procedimento dobrar duplica o valor que está armazenado no parâmetro formal x. Como x é uma referência, a operação terá efeito sobre o parâmetro real que for repassado numa chamada a dobrar. Exemplo,

```
var

    q: integer;

begin

    q := 26;

    dobrar(q);

    writeln(q);                 // 52

end.
```

O uso de referências permite que procedimentos possam funcionar como funções, ou seja, possuam uma saída tangível. Por exemplo, no protótipo,

```
procedure get(a, b, c: integer; var x: real);
```

pode-se abstrair que a, b e c são argumentos de entrada e que x é um argumento de saída. Em geral o valor que x contém na entrada do módulo não é importante, mas seu estado final, antes do retorno do controle, representa o valor de saída. No programa da Listagem 47 a função eq2g recebe três argumentos por valor (coeficientes de um polinômio de segundo grau) e gera três saídas: duas através dos argumentos (referências) x1 e x2 (raízes do polinômio) e uma terceira saída natural da função (booleano). Semanticamente a função tenta calcular as raízes e colocá-las em x1 e x2.

Quando o primeiro coeficiente é nulo ou o delta é negativo (teste da Linha-07) a função retorna com false (associação das Linhas 08 e 09). As raízes são calculadas nas Linhas 11 e 12 e a validade delas é validada pelo retorno true adiante (Linha-13). No programa principal p e q funcionam como argumentos reais de eq2g que é chamada dentro de uma estrutura de decisão (Linha-22). Um retorno true de eq2g indica que as raízes de fato existem, foram calculadas e armazenadas nas variáveis p e q. A decisão então desvia para a impressão dos valores de raízes (Linha-23). Do contrário nenhuma raiz é calculada e sequer o conteúdo de p e q são modificados. O desvio para a impressão da mensagem na Linha-25 indica isso.

Listagem 47

```
01 program equacao_do_segundo_grau;
02
03 function eq2g(a, b, c: real; var x1, x2: real): boolean;
04 var delta: real;
05 begin
06     delta := b*b - 4*a*c;
07     if (abs(a)<0.001) or (delta<0) then begin
08         eq2g := false;
09         exit;
10     end;
11     x1 := ( -b + sqrt(delta) )/(2*a);
12     x2 := ( -b - sqrt(delta) )/(2*a);
13     eq2g := true;
14 end;
15
16 var
17    c: array[1..3] of real;
18    p, q: real;
19 begin
20    write('Coeficientes: ');
21    readln(c[1], c[2], c[3]);
22    if eq2g(c[1], c[2], c[3], p, q) then
23        writeln( p:0:3, ' ', q:0:3 )
24    else
25        writeln('Polinomio invalido ou raizes complexas!');
26 end.
```

4. Recursividade

A recursividade é um recurso de linguagem que permite um módulo, seja ele função ou procedimento, fazer uma chamada de si mesmo, ou seja, dentro do corpo de código do módulo é possível fazer uma ou mais chamadas daquele módulo. A recursividade permite implementações mais claras de diversos algoritmos e por essa razão é um recurso nativo do Pascal. Para calcular, por exemplo, o fatorial de um inteiro pode-se usar a seguinte função,

Além da palavra reservada var, as sessões de argumentos formais em módulos contam com a palavra reservada out para definir referências. Out ajuda, pelo menos em inglês, a lembrar que o argumento deve funcionar como saída.

```
function fat(n: integer): integer;

begin

    if n<2 then

        fat := 1

    else

        fat := n * fat(n-1);

end;
```

Nesta função o identificador fat, sem qualquer código pós-fixado, representa a variável interna cujo valor antes da finalização da função é seu valor de retorno. O mesmo identificador com os parênteses pós-fixados, fat(n-1), possui semântica diferente e representa uma chamada recursiva ao módulo fat. Observe que o argumento desta chamada é o valor de entrada n decrescido de uma unidade e logo o retorno da função deve ser n multiplicado pelo fatorial de n-1.

Este retorno entretanto não pode devolver imediatamente o controle ao chamador porque depende do retorno da chamada a fat(n-1). Por sua vez esta chamada causará outra chamada, desta vez a fat(n-2), e esta a fat(n-3) e assim por diante. Cada vez que uma nova chamada é realizada a fat, uma nova versão da função é empilhada em memória sobre aquela que a chamou gerando uma cadeia de dependências de retorno.

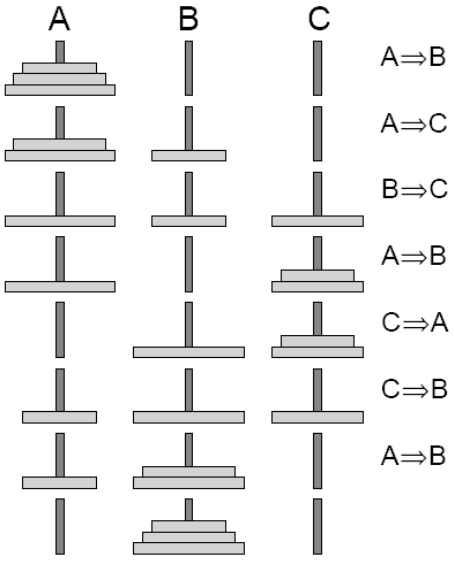
O processo de empilhamento de funções seria infinito não fosse a *condição de parada* implementada nas duas primeiras linhas de código do corpo de fat. O if obriga que entradas menores que 2 retornem imediatamente o valor 1 (o fatorial de 0 e 1 é de fato 1, mas o fatorial de valores negativos não existe, por isso cautela no uso desta função).

Assim quando o empilhamento alcançar valores menores que 2, nenhuma nova chamada recursiva será realizada e começará um processo, de trás para a frente (em relação a ordem das chamadas recursivas) de resoluções de retornos até que a chamada original seja alcançada. A sequência de operações a seguir descreve o cálculo do fatorial de 5 pela função fat,

```
Fat(5)                          // 1 chamada pendente
5 * fat(4)                      // 2 chamadas pendentes
5 * 4 * fat(4)          // 3 chamadas pendentes
5 * 4 * 3 * fat(2)              // 4 chamadas pendentes
5 * 4 * 3 * 2 * fat(1)  // 5 chamadas pendentes
5 * 4 * 3 * 2 * 1               // 4 chamadas pendentes
5 * 4 * 3 * 2                   // 3 chamadas pendentes
5 * 4 * 6                       // 2 chamadas pendentes
5 * 24                          // 1 chamada pendente
120                             // resolvido
```

Para ilustrar um uso mais sofisticado da recursividade, consideremos o problema da *Torre da Hanói*. Sejam três pinos A, B e C onde se podem empilhar discos perfurados de diâmetros distintos. O objetivo do problema é mover todos os discos que estão no pino A (origem) para o pino B (destino) utilizando o pino C como auxiliar. Há duas restrições associadas, (i) nunca um disco poderá ficar sobre outro de diâmetro menor e (ii) só poderá ser movido um disco de cada vez. Inicialmente A contém n discos empilhados em ordem decrescente de diâmetro de baixo para cima e os outros dois estão vazios.

O jogo *Torre de Hanói* foi inventado pelo matemático Francês Édouard Lucas em 1883. Édouard teve inspiração de uma lenda Hindu para construir o jogo. Já seu nome foi inspirado na torre símbolo da cidade de Hanói, capital e segunda maior cidade do Vietnã.



Para movimentar n discos do pino A para o pino B, usando C como auxiliar, é necessário primeiramente mover os n-1 discos do topo de A para o pino C deixando caminho livre para que o disco na base de A (o enésimo disco) possa ser movido para sua posição final em B. Da mesma forma, para proporcionar o movimento dos n-1 discos, agora em C, para seu destino final

em B, é necessário movimentar todos os n-2 no topo de C para A usando B como auxiliar. O raciocínio prossegue recursivamente gerando pendências de movimentos e tendo como condição de parada a tentativa de movimentação de zero discos.

O programa da Listagem 48 ilustra como implementar uma solução para o problema da Torre de Hanói em Pascal. O procedimento hanoi tem por objetivo listar a sequencia de movimentações que devem ser feitas para a resolução do problema. Seu primeiro argumento corresponde a quantidade de discos a serem movidos.

Os três argumentos finais referem-se respectivamente aos rótulos dos pinos origem, destino e auxiliar. Na Linha-05 está a condição de parada quando não há mais discos a movimentar. Na Linha-06 ocorre a primeira chamada recursiva que movimenta os n-1 discos do topo de A para C usando B como auxiliar (observe a sequencia dos pinos de entrada).

Quando estes são resolvidos (na verdade são criados apenas pendências), é feita a movimentação do disco na base do pino A para o pino B na Linha-07 (neste caso a movimentação mecânica é substituída por uma mensagem de texto indicando os rótulos dos pinos de partida e chegada). Na Linha-08 ocorre a segunda chamada recursiva que movimenta os n-1 discos em C para B utilizando A como auxiliar (novamente os parâmetros mudam). No programa principal, Linha-12, o procedimento hanoi é chamado para o caso de três discos rotulados por 'A', 'B' e 'C'. A saída do programa é (veja também Figura-5.1),

```
A > B | A > C | B > C | A > B | C > A | C > B | A > B |
```

Listagem 48

```
01  program torre_de_hanoi;
02
03  procedure hanoi(n: integer; A, B, C: char);
04  begin
05      if n = 0 then exit;
06      hanoi(n-1, A, C, B);
07      write(A, ' > ', B, ' | ');
08      hanoi(n-1, C, B, A);
09  end;
10
11  begin
12      hanoi(3, 'A', 'B', 'C');
13  end.
```

5. Aninhamento Modular

Em Pascal é possível construir um módulo (função ou procedimento) dentro do corpo de outro módulo. Este recurso é conhecido como *aninhamento modular* e os módulos aninhados de *módulos internos*. No exemplo,

```
procedure teste;

    function soma(a,b: integer): integer;

    begin

        return a+b;

    end;

begin

    write( soma(3, 8) );

end;
```

a função soma é interna ao procedimento teste.

Um módulo interno é de propriedade exclusiva do módulo que o contém e dessa forma não pode ser chamado fora dele. Assim soma, no exemplo anterior, só pode ser chamado dentro do corpo da função teste (como ilustrado) ou de dentro de outras funções internas a teste (neste caso não há nenhuma).

Para ilustrar o aninhamento modular, consideremos o problema de implementação da *busca binária*. Este algoritmo consiste na busca por uma chave em uma array cujos valores estão ordenados. A seguir descrevemos o algoritmo da busca binária. Seja L uma array ordenada onde se deseja determinar a posição de uma chave x (caso esteja contida) e Φ uma subarray de L.

Seja M(Φ) a função que determina o valor da célula mediana de Φ, Φ^{**} a subarray de Φ formada pelas células anteriores a M(Φ) e Φ^{**} a subarray de Φ formada pelas células de Φ posteriores a M(Φ). Cada etapa de uma busca binária consiste em comparar M(Φ) com x. Caso os valores sejam iguais, significa que obteve sucesso e a posição procurada é o índice em L onde está M(Φ). Do contrário, se M(Φ) for diferente de x, então Φ passa a ser ou Φ^{*}, caso M(Φ) seja maior que x, ou Φ^{**}, caso M(Φ) seja menor que x, e o processo é repetido até que x seja encontrado ou Φ tenha comprimento zero (busca sem sucesso). Na primeira etapa da busca binária, Φ é tomada como sendo a array L integral.

Listagem 49

```
01  program busca_binaria;
02
03  function busca_bin(L: array of integer; x: integer): integer;
04      function rbusca(min, max: integer): integer;
05      var med: integer;
06      begin
07          rbusca := -1;
08          if max<min then exit;
09          med := (min + max) div 2;
10          if L[med] = x then
11              rbusca := med
12          else if L[med]>x then
13              rbusca := rbusca(min, med-1)
14          else
15              rbusca := rbusca(med+1, max);
16      end;
17  begin
18      busca_bin := rbusca( 0, length(L)-1 );
19  end;
20
21  const
22     M: array[0..11] of integer = (2,3,6,11,21,24,38,39,44,50,70,78);
23  var
24     x: integer;
25  begin
26     x := busca_bin(M, 11);
27     if x > 0 then
28         writeln(x)
29     else
30         writeln('nao encontrado');
31  end.
```

Na Listagem 49 a função busca_bin implementa a busca binária em Pascal. A array de busca e a chave procurada são recebidos respectivamente através dos argumentos L e x (Linha-03). Internamente busca_bin aninha a função recursiva rbusca que efetua a busca propriamente dita. Os argumentos de rbusca (min e max na Linha-04) representam respectivamente os índices em L (módulos internos tem acesso a todos os recursos locais do

módulo que os aninha) e que limitam a subarray investigada (Φ). Na Linha-07 supõe-se que x não está em L (primeira hipótese) e por isso a variável interna rbusca recebe -1 (definimos que uma busca sem sucesso deve retornar um valor de índice negativo).

A Linha-08 contém a condição de parada da recursão causando retorno imediato quando o índice inferior (min) extrapola o superior (max). Na Linha-09, med é utilizada para calcular a posição mediana da subarray investigada (Φ). O processo de decisão entre as Linhas 10 e 15 verifica se a chave foi encontrada (Linha-11), se a pesquisa continua em Φ^* (chamada recursiva da Linha-13) ou em Φ^{**} (chamada recursiva da Linha-15).

Nenhuma das duas chamadas recursivas mencionadas inclui o valor em med e por essa razão utiliza-se med-1 na Linha-13 e med+1 na Linha-15. Na Linha-18 busca_bin faz uso de rbusca passando os índices apropriados de L (array completa) como argumentos. O programa principal entre as Linhas 21 e 31 ilustra o uso de busca_bin utilizando como entrada a array constante M. Esta array é declarada com índice inferior zero afim de o retorno de busca_bin não necessitar de correção (lembre-se que o índice inferior de L é zero).

6. Unidades

A saída gerada pelo compilador quando recebe como entrada um arquivo de código de um programa é um arquivo binário executável. Ele representa o código de máquina que pode ser executado diretamente em hardware e ao mesmo tempo está atrelado a serviços do sistema operacional sobre o qual se procedeu a compilação. Se sessões de cláusula uses estão presentes, então o compilador acessará e anexará recursos de outros arquivos que são binários mas que não são executáveis. Em Pascal estes arquivos são denominados *unidades* e são também provenientes de outros processos de compilação mas a estrutura básica de implementação é diferente.

A estrutura básica para implementação de uma unidade Pascal é a seguinte,

```
unit <nome_da_unidade>;

    <recursos>

interface

    <interfaces>

implementation

    <implementações>

end.
```

As palavras reservadas unit, interface e implementation formam o corpo estrutural de quaisquer unidades em Pascal. Unit deve ser seguido por um identificador que dá nome a unidade e ao arquivo desta unidade (que ainda possui uma extensão .pas). Se o nome do arquivo de unidade for diferente do nome de unidade o compilador emitirá um um erro de compilação.

Como acontece com programas, a identificação da unidade encerra em ponto-e-vírgula. Interface e implementation abrem respectivamente as sessões de interface e implementação da unidade. A sessão de implementação é essencialmente um contêiner de módulos (que podem ser funções ou procedimentos) que desempenham tarefas relacionadas a uma determinada área de aplicação (como redes, computação gráfica, aritmética, matrizes, recursos de sistema, manipulação de arquivos, programação web e etc).

Entretanto nem tudo que fica nessa sessão deve ser acessado por usuários da unidade (imagine, por exemplo, uma função que deva ser utilizada por outras funções da unidade mas nunca pelo usuário da unidade). Somente recursos que são *atalhados* na sessão de interface são de fato acessíveis por usuários da unidade. Para atalhar, por exemplo, um módulo basta copiar seu protótipo para a sessão de interface. No exemplo,

```
unit teste;


interface


procedure ola;


implementation


procedure ola;
begin
    write('OLA');
end;


procedure tchau;
begin
    write('TCHAU');
end;


end.
```

existem dois procedimentos implementados, mas apenas um deles está atalhado na sessão de interface. Dessa forma a compilação do programa,

```
program novo;

uses teste;

begin

    ola;

    tchau;

end;
```

reconhece o módulo ola, mas emite erro de compilação ao encontrar tchau alegando não existir o procedimento.

De forma geral, se uma constante, variável, tipo ou módulo é definido ou prototipado na sessão de interface, ele será acessível ao usuário, mas se esta definição ocorrer na sessão de implementação, será de uso exclusivo da própria unidade.

As unidades que acompanham o Free Pascal ficam num mesmo diretório (por exemplo, no Linux utilizado para compilar os exemplos deste livro elas estão em /usr/lib/fpc/2.4.0/units/). O arquivo fpc.cfg (a localização dependerá do sistema) contém, entre outras informações relevantes ao compilador, os endereços para as unidades instaladas e por essa razão a simples referência com uses é suficiente para carregar apropriadamente unidades declaradas. Entretanto, se as unidades estiverem no mesmo diretório do arquivo fonte que as solicita (com uses), elas são carregadas normalmente.

Cada nova unidade implementada em um projeto deve ser compilada independentemente. Para compilar um arquivo de código fonte de unidade, em linha de comando, utiliza-se a mesma sintaxe de compilação de programas. Exemplo,

```
$ fpc teste.pas
```

A compilação de um arquivo fonte de unidade tem como saída dois arquivos com mesmo nome da entrada mas com extensões diferentes, em geral .o e .ppu. O arquivo de extensão .o é o arquivo objeto (que surge também na compilação de programas) e representa uma ponte entre o arquivo de código fonte e o arquivo binário final de extensão .ppu. Mesmo havendo no mesmo diretório do projeto o arquivo de código de unidade .pas, é o arquivo binário .ppu que o compilador requer quando uma cláusula uses precisa ser resolvida em outra parte do projeto. Assim se o .pas de uma unidade é modificado, ele precisa ser recompilado ou do contrário o .ppu antigo será utilizado. Em projetos assistidos por uma IDE, normalmente uma compilação resolve os problemas de recompilação de unidades que tiveram seus códigos modificados.

A Listagem 50 mostra a implementação da unidade ncomplex que permite representar e operar números complexos (soma, subtração, multiplicação e divisão). Na interface ficam a definição do tipo de dados complex, que representa um número complexo (re representa a parte real e im a imaginária) além dos protótipos das operações básicas. Apenas a função conj, Linha-39, não possui interface.

Listagem 50

```
01 unit ncomplex;
02
03 interface
04
05 type complex = record
06     re, im: real;
07 end;
08
09 procedure init(var c: complex; re, im: real);
10 function soma(a,b: complex): complex;
11 function subt(a,b: complex): complex;
12 function mult(a,b: complex): complex;
13 function quoc(a,b: complex): complex;
14 procedure print(c: complex);
15
16 implementation
17
18 procedure init(var c: complex; re, im: real);
19 begin
20     c.re := re;
21     c.im := im;
22 end;
23
24 function soma(a,b: complex): complex;
25 begin
26     init(soma, a.re + b.re, a.im + b.im);
27 end;
28
29 function subt(a,b: complex): complex;
30 begin
31     init(subt, a.re - b.re, a.im - b.im);
32 end;
33
34 function mult(a,b: complex): complex;
35 begin
36     init(mult, a.re*b.re - a.im*b.im, a.re*b.im + a.im*b.re);
37 end;
38
39 function conj(x: complex): complex;
40 begin
```

Quando uma raiz de número negativo surge na resolução de um problema baseado em números reais, diz-se que o problema não possui solução real. Como alternativa a matemática definiu a constante $i = \sqrt{-1}$ e chamou todos os números escritos com formato *a+bi*, sendo *a* e *b* reais, de números complexos. O valor de *a* é chamado de parte real e *b* de parte imaginária. O número *a-bi* é dito conjugado de *a+bi*. As operações básicas de soma, subtração, multiplicação e divisão são obtidas assim,

$$(a+bi)\pm(c+di)=(a\pm c)+(b\pm d)i$$
$$(a+bi)(c+di)=(ac-bd)+(ad+bc)i$$
$$\frac{a+bi}{c+di}=\frac{a+bi}{c+di}\frac{(c-di)}{(c-di)}=\frac{ac+bd}{c^2+d^2}+\frac{bc-ad}{c^2+d^2}i$$

Ainda na Listagem 50, o procedimento init inicializa um número complexo, que recebe por referência no primeiro argumento, com os valores de seus dois últimos argumentos. As funções soma, subt, mult e quoc retornam um novo complexo respectivamente igual a soma, subtração, mul-

tiplicação e divisão dos dois complexos que recebem como argumentos. A função conj, sem interface, retorna o conjugado do complexo que recebe (o conjugado do complexoé). Ela é utilizada em quoc para facilitar a implementação da divisão. Print imprime o complexo de entrada em notação de parênteses.

Listagem 51

```
01  program teste_complexos;  07      print( soma(a,b) );
02  uses ncomplex;            08      print( subt(a,b) );
03  var a, b: complex;        09      print( mult(a,b) );
04  begin                     10      print( quoc(a,b) );
05      init(a, 3, 2);        11  end.
06      init(b, -2, 6);
```

O programa da Listagem 51 ilustra o uso da unidade ncomplex imprimindo o resultado das quatro operações básicas implementadas para dois complexos dados. A saída obtida é,

```
(1.0,8.0)
(5.0,-4.0)
(-18.0,14.0)
(0.2,-0.6)
```

Atividades de avaliação

1. Se uma array for passada por referência a um módulo, ele poderá modificá-lo. Utilizando este recurso implemente uma função capaz receber e ordenar uma array de comprimento qualquer.
2. Implemente uma versão de resolução do problema da Torre de Hanói utilizando quatro pinos (um origem, um destino e dois auxiliares) de forma a diminuir, para uma mesma quantidade de discos dada, o número de movimentações.
3. Implementar uma versão não recursiva (iterativa) para a busca binária.
4. A sequência de Fibonacci é uma série infinita de números naturais cujos dois primeiros termos são 1 e os demais são calculados pela soma dos dois termos anteriores. Os seis primeiros termos da sequência são então 1, 1, 2, 3, 5 e 8. Criar duas versões de procedimentos, uma iterativa e outra recursiva, que imprimam os *n* primeiros termos da sequência de Fibonacci, onde *n* é um argumento de entrada.

```
41      init(conj, x.re, -x.im);
42  end;
43
44  function quoc(a,b: complex): complex;
45  var x: complex; d: real;
46  begin
47      x := mult( a, conj(b) );
48      d := b.re * b.re + b.im * b.im;
49      init(quoc, x.re/d, x.im/d);
50  end;
51
52  procedure print(c: complex);
53  begin
54      writeln('(', c.re:0:1, ',', c.im:0:1, ') ');
55  end;
56
57  end.
```

5. O algoritmo de Euclides para cálculo do máximo divisor comum (*mdc*) de dois números naturais *a* e *b* possui os seguintes passos: (i) se *b* for zero, então pare pois o *mdc(a, b)* é o próprio *a*, do contrário, prossiga; (ii) *c* recebe o resto da divisão de *a* por *b*; (iii) *a* recebe o valor de *b* e *b* recebe o valor de *c*; (iv) volte ao passo *i*. Utilizando este algoritmo implemente uma função para o cálculo do *mdc* de dois naturais recebidos como argumentos.

6. Construa uma versão recursiva para o algoritmo de Euclides descrito na questão anterior.

7. O mínimo múltiplo comum *(mmc)* de dois inteiros, *a* e *b*, pode ser calculado por $mmc(a,b) = \frac{ab}{mdc(a,b)}$. No caso de uma lista de inteiros, o *mmc* é recursivamente definido por *F(L) = mmc(x, F(L*))*, onde *F* é o *mmc* da lista de inteiros *L*, *x* é o primeiro valor de *L* e *L** é a sub-lista de *L* obtida pela extração de *x*. Escreva função que receba uma lista de inteiros e retorne o *mmc* entre seus valores.

8. Inverter uma array significa dispor seus elementos em ordem inversa. Construa uma função recursiva para essa tarefa.

9. Construa uma unidade que permita manipular frações reduzidas. Uma fração reduzida é aquela cujos numerador e denominador são inteiros primos entre si (o *mdc* entre eles é um). Devem estar presentes módulos para inicializar, somar, subtrair, multiplicar, dividir e imprimir frações reduzidas.

10. Use a unidade construída na questão anterior para obter o numerador de $\sum_{k=0}^{200} \frac{k}{2k+1}$.

Capítulo 6

Arquivos

1. Memória Secundária

Executar um programa significa processar sequencialmente suas instruções de máquina. Este processo requer não apenas uma CPU de processamento eficiente, mas também de memória onde possam ser armazenadas e buscadas rapidamente as instruções. É a chamada *memória principal*. A performance desta memória é justificada pela associação de duas características: o *acesso aleatório* e a *volatilidade*. O acesso aleatório é a capacidade que o hardware possui de permitir que qualquer célula de memória seja acessada num tempo constante e muito pequeno. A volatilidade refere-se a condição de presença de corrente elétrica no dispositivo para que alguma informação esteja presente.

Memória Secundária é um termo utilizado para se referir a qualquer dispositivo que almeje o armazenamento não volátil de informações (estarão registradas mesmo após o desligamento do dispositivo). De forma geral, devido a diferença funcional, existe maior quantidade de memória secundária em relação a primária (que é também mais cara). A memória secundária é representada por dispositivos como discos rígidos, *pendrivers* e cartões de memória (de câmeras digitais, celulares e etc).

O gerenciamento de memória principal e secundária pelo sistema operacional é feita de forma distinta. A disposição dos programas em execução na memória principal não é diretamente visível ao usuário que apenas pode obter informações gerais como o total de memória utilizado. No caso da memória secundária toda informação é visualizável em forma de uma árvore de *diretórios* e *arquivos*.

Um arquivo é tecnicamente uma sequencia de bytes com nome próprio, tamanho definido e endereço exclusivo (veja Figura - 10). Um diretório ou pasta é um contêiner de arquivos e também de outras pastas (por isso a hierarquia de dados em memória secundária é análoga a uma árvore). A primeira

pasta, aquela que contém todas as demais, é chamada de *raiz* e o conjunto de todas as pastas e seus respectivos arquivos, de *sistema de arquivos*. Há regras para montar um sistema de arquivos e elas mudam conforme o sistema operacional e o dispositivo utilizados.

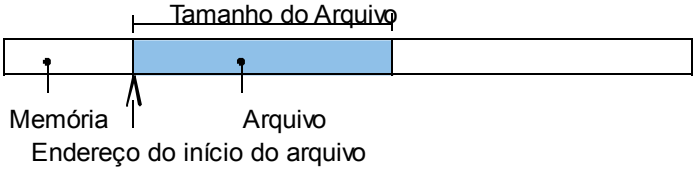
O mesmo dispositivo de memória secundária, como por exemplo discos rígidos, pode inclusive possuir mais de um sistema de arquivo com mesmas regras ou regras diferentes. Cada fatia lógica em memória secundária, que mantém um sistema de arquivos, é denominada de *partição*. Usualmente as regras de montagem de um sistema de arquivos são referenciadas simplesmente como sistema de arquivos. Assim são exemplos de sistemas de arquivos o FAT e o NTFS da Microsoft, o ext3, ext4 e reiserFS do Linux e o HPFS da IBM.

Partição Lógica
é possível aos sistemas operacionais atuais definir que parte, por exemplo, de um disco será utilizada. Na prática, uma vez estabelecido isso, todas as operações de leitura e gravação de arquivos ocorrerão nessa parte. Não existe uma barreira física de acesso ao restante do disco e por essa razão diz-se que o particionamento é lógico.

Discos rígidos representam a mais comum forma de memória secundária que permite o armazenamento maciço de informação. Internamente eles são formados por discos magnéticos rígidos (daí a origem do nome) fixos a um eixo central que gira em uma alta velocidade sobre um mancal de precisão. Um braço mecânico, com liberdade angular de movimentação, mantém diversas extensões com agulhas magnetizadas que interagem com a superfície de cada uma das faces dos discos rígidos efetuando operações de leitura e gravação.

Essa parafernália justifica a lentidão das operações em discos rígidos, pelo menos quando comparadas àquelas em memória principal (que devido a volatilidade, dispensa partes mecânicas). Apesar disso ainda representam a melhor opção para armazenamento maciço não volátil no que diz respeito a análise custo *versus* quantidade de memória *versus* velocidade de acesso.

Para acessar informações num dispositivo de armazenamento primário ou secundário é necessário inicialmente *deslocar-se* até o endereço onde fica o dado pretendido e só então efetuar a leitura e/ou gravação. Em alguns hardwares esse deslocamento é instantâneo, mas em outros toma um tempo considerável. Essa disparidade ocorre devido a presença de partes móveis no segundo caso (*inércia de dispositivo*). Assim os dispositivos do primeiro caso estão haptos ao *acesso aleatório*, ou seja, acesso a qualquer parte do disco em qualquer tempo enquanto os da segunda categoria são melhores aproveitados no *acesso sequencial*, ou seja, ao invés de saltar entre dados distantes é mais interessante efetuar acessos seguidos a células sequenciais minimizando o efeito do deslocamento.



Há dispositivos de memória secundária de acesso aleatório e de acesso sequencial. Nos dispositivos de acesso aleatório, como os discos rígidos, o tempo de acesso a blocos fundamentais de bytes ocorre num tempo aproximadamente constante definido pelo fabricante, mas que irá variar de modelo para modelo. Nos dispositivos de acesso sequencial, como as fitas magnéticas, o tempo de acesso a blocos fundamentais de bytes muda consideravelmente.

Nestes dispositivos uma agulha magnetizada de leitura fica fixa enquanto dois rolos, compartilhando uma mesma fita magnética (como num cassete) giram levando ou trazendo a fita conforme a posição que ela deva ficar sob a

agulha. A restrição de acesso nativa de dispositivos de acesso sequencial os tornam mais interessantes para operações de gravação sequencial de grandes quantidades de dados como em *backups* (cópias de segurança).

Um *sistema de manipulação de arquivos*, em uma linguagem de programação, é um conjunto de recursos que permite criar, modificar ou excluir arquivos de um sistema de arquivos. Tais sistemas devem manter uma camada abstrata que permita ao programador tratar arquivos em diferentes tipos de memória secundária com o mesmo conjunto de recursos de linguagem. Para que isso seja possível, um sistema de manipulação de arquivos não trabalha diretamente em memória secundária.

Ao invés disso ele cria e manipula arquivos em memória principal. Cada instância em memória principal de um arquivo, chamada de *fluxo* ou *stream*, é ligada logicamente a uma instância em memória secundária (arquivo propriamente dito) e ocasionalmente, ou forçadamente, o conteúdo é copiado de uma para outra. Esse processo é conhecido como *descarregamento* ou *flushing*. Um arquivo que requer muitas operações tem a opção de fazer poucos descarregamentos e assim tira proveito do que cada tipo de memória melhor oferece. Eventualmente, enquanto se manipula um arquivo, se houver queda de anergia por exemplo, o conteúdo em memória é perdido e aquele no arquivo corresponderá ao proveniente do último descarregamento

Pascal possui mais de um sistema de manipulação de arquivos. O mais comumente abordado é aquele mantido pela unidade system e que será estudado neste capítulo. A unidade sysutils possui um sistema de manipulação de arquivos alternativo semelhante ao da linguagem C.

Salvar um documento, uma imagem, uma planinha, um desenho e etc, utilizando uma aplicação especializada, constitui uma tarefa de descarregamento forçado da memória principal para a secundária.

2. Fluxos

Em Pascal os arquivos são classificados em *binários* e *de texto*. Um arquivo binário é aquele gerado pela concatenação de dados provenientes diretamente da memória principal e sem processamento adicional (isso significa que o arquivo contém uma *imagem* do que esteve em memória). Arquivos binários podem ser *tipados* e *não tipados*. Um arquivo binário tipado é aquele formado pela concatenação de dados binários de mesmo tipo e tamanho denominados de *registros de arquivo*.

Um arquivo de dados não tipado concatena dados binários não necessariamente de mesmo tipo ou tamanho. Um arquivo texto é aquele constituído exclusivamente por caracteres visíveis (da tabela ASCII) e comummente representam algum tipo de documentação. Eles requerem processamento adicional para serem gerados, ou seja, os dados em memória, que precisam ser gravados num arquivo texto, precisam ser primeiramente transformados do

formato original para strings antes de poderem ser registrados.

Para lidar com arquivos em Pascal é necessário definir primeiramente um fluxo (*stream*) para ele. Uma variável que define um fluxo é denominada *manipulador*. Para criar um manipulador de fluxo binário não tipado usa-se a sintaxe,

```
var
        <manipulador>: file;
```

onde a palavra reservada file significa *arquivo*. Exemplo,

```
var
        F: file;
```

onde F define um fluxo para um arquivo binário não tipado. Para definir um fluxo de arquivo binário tipado, usa-se a sintaxe,

```
var
        F: file of <tipo_base>;
```

onde file of são palavras reversavas que significam *arquivo de*. O *tipo base* define o tipo comum dos dados concatenados no arquivo. No exemplo,

```
var
        F: file of integer;
```

F define um fluxo para um arquivo tipado constituído pela concatenação de números inteiros.

Para definir um manipulador para um fluxo de texto usa-se a sintaxe,

```
var
        <manipulador>: text;
```

onde text é uma palavra reservada que significa *texto*. No exemplo,

```
var
        T: text;
```

T define um manipulador para um fluxo de arquivo texto.

Um fluxo, depois de definido, precisa ser vinculado a um arquivo (que necessariamente não precisa existir ainda) em um dispositivo real (como um disco). Esta vinculação é realizada pelo comando assign cuja sintaxe é,

```
assign(<fluxo>, <nome_do_arquivo_no_dispositivo_real>);
```

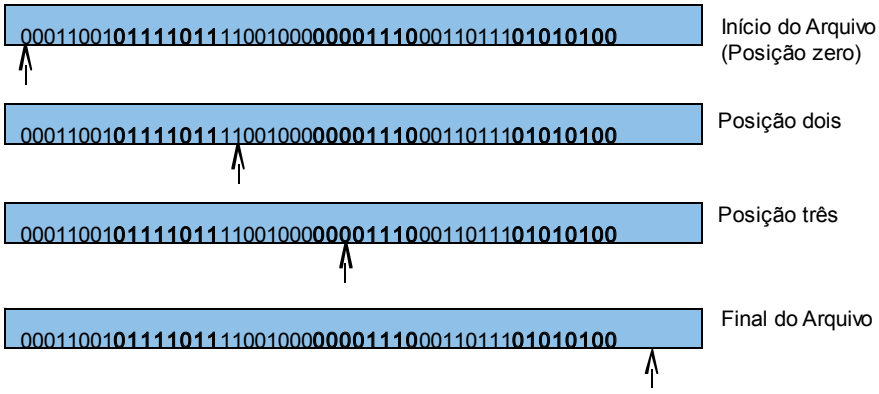
O primeiro argumento é uma referência a um fluxo que deve ser vinculado a um arquivo cujo nome é o segundo argumento. Um nome de arquivo em geral segue as regras de endereçamento de arquivos do sistema operacional onde o programa é compilado. Se o arquivo é aberto ou criado na mesma pasta do binário executável, não é necessário detalhar o endereço com hierarquia de pastas, como em,

Na maioria dos sistemas um nome de arquivo é construído pela sequência das pastas que se sucedem hierarquicamente desde a raiz até o nome propriamente dito do arquivo. No Linux, e demais sistemas baseados no Unix, estas pastas são separadas pelo caractere barra comum, '/', como no exemplo '/temp/data/teste.dat'. No Windows a raiz de cada partição recebe uma letra e utiliza-se barra invertida, '\' para separar as pastas. Um exemplo equivalente ao anterior é 'c:\temp\data\teste.dat'.

```
assign(F, 'teste.dat');
```

Neste exemplo o fluxo F se vincula a um arquivo de nome teste.dat na mesma pasta do binário executável. Se existir uma subpasta na pasta do binário executável, o programador poderá usar a *notação ponto* para acessar arquivos dentro dela. Exemplo,

```
assign(F, './data/teste.dat');          // Linux
```



no Linux ou,

```
assign('.\data\teste.dat');             //Windows
```

em Windows. Em ambos casos a subpasta data deve existir dentro da pasta que contém o binário executável. Do contrário haverá erro de acesso.

3. Modos de Acesso

Depois da vinculação, o passo seguinte é definir o *modo de acesso* do arquivo. O modo de um arquivo define basicamente o suporte às operações de *leitura* (verificação dos dados que ele contém) e *escrita* (adição de novos dados ou modificação de dados já existentes). O modo de um arquivo também define a sua chamada *posição corrente* que equivale a um ponto de entrada de uma nova operação de leitura ou escrita no arquivo.

A posição corrente é representada por uma medida em registros entre o início do arquivo e o ponto de operação. Assim, por exemplo, a posição zero indica que zero registros precedem a posição corrente (início do arquivo). Na Figura-6.2 uma seta ilustra a posição corrente em quatro pontos distintos de um mesmo arquivo cujos registros possuem um byte cada.

Há três modos de acesso de arquivos em Pascal,

- Modo de Escrita
- Modo de Leitura
- Modo de Anexação

No modo de escrita, para fluxos de texto, será permitida apenas a gravação de dados. Para fluxos binários será possível leitura e gravação. Neste modo, se o arquivo ainda não existir, ele é logicamente criado em memória secundária. Se ele já existir então seu conteúdo é sumariamente apagado antes das novas operações de escrita. Assim o processamento inicia sempre com um arquivo de tamanho zero (bytes). Para definir um arquivo em modo de escrita usa-se o comando rewrite que recebe como argumento único o fluxo que deve configurar. Exemplo,

```
assign(F, 'teste.dat');
rewrite(F);
...
```

onde F é um fluxo. Neste exemplo, se teste.dat não existe ainda, ele é criado, mas do contrário, tem o conteúdo apagado.

No modo de leitura, o arquivo precisa existir para que possa ser aberto. Neste modo, caso seja um arquivo de texto, os dados já existentes poderão ser lidos mas nunca modificados nem novos dados adicionados. Para arquivos binários as operações de leitura e gravação são permitidas. A posição corrente de um arquivo aberto em modo de leitura é zero (início do arquivo). Para abrir um arquivo em modo de leitura usa-se o comando reset cujo único argumento é o fluxo que deve ser vinculado ao arquivo. No exemplo,

```
assign(F, 'teste.dat');
reset(F);
...
```

o arquivo 'teste.dat', que precisa existir, é vinculado ao fluxo F e sua posição corrente marcada no início do arquivo.

No modo de anexação um arquivo já existente é aberto para receber novos dados. Se o arquivo ainda não existe um erro ocorrerá. Se ele já existe, seu conteúdo original é mantido. Um arquivo aberto no modo de anexação tem sua posição corrente apontada para o fim do arquivo afim de que as novas operações agreguem novos dados e não sobrescrevam os que já existem. Somente fluxos para arquivos texto podem ser abertos neste modo. Para abrir um arquivo texto em modo de anexação usa-se o comando append cujo único argumento é o fluxo que será vinculado ao arquivo. No exemplo,

```
assign(F, 'teste.txt');
append(F);
...
```

o fluxo de texto F é vinculado ao arquivo 'teste.txt'.

Nem sempre é possível abrir um arquivo. Quando este for o caso então um erro em tempo de execução ocorrerá e o programa será suspenso. Utilizando a diretiva {$I-}, antes do código de abertura do arquivo, é possível desligar a emissão explícita do erro e manter a execução. O custo desta desabilitação é a necessidade de checar manualmente se ocorreu silenciosamente um erro e impedir que ele cause outros erros mais adiante.

Essa checagem manual é feita com a função IOResult que retorna o status da última operação executada pelo sistema de manipulação de arquivos. Se IOResult retornar zero, nenhum erro terá ocorrido. Do contrário o valor de retorno, que é um inteiro, corresponderá ao código de um erro. A Tabela-6.1 lista os possíveis códigos de erros de saída de IOResult e seus respectivos significados. Para religar a emissão de erros usa-se a diretiva {$I+}.

Códigos de IOResult

2	Arquivo não encontrado	106	Número inválido
3	Pasta não encontrada	150	Disco protegido contra escrita
4	Muitos arquivos abertos	151	Dispositivo não conhecido
5	Acesso negado	152	Dispositivo não está pronto
6	Manipulador de arquivo inválido	153	Comando não conhecido
12	Modo de acesso inválido	154	Checagem CRC falhou
15	Número de disco inválido	155	Especificação de dispositivo inválida
16	Não pode remover diretório corrente	156	Encontrado erro em disco
17	Não pode renomear volume	157	Tipo de mídia inválida
100	Erro ao ler em disco	158	Setor não encontrado
101	Erro ao escrever em disco	159	Impressora sem papel
102	Arquivo não vinculado	160	Erro ao escrever para dispositivo
103	Arquivo não aberto	161	Erro ao ler de dispositivo
104	Arquivo não aberto para escrita	162	Falha de Hardware
105	Arquivo não aberto para leitura		

Tabela 1

A função existe na Listagem 52 testa a existência de um arquivo cujo nome é repassado como argumento (nome). A diretiva de desligamento na Linha-04 impede que, no caso de não existência do arquivo, um erro causado por reset (Linha-06) suspenda o programa. Se nome for uma string não vazia e IOResult retornar zero então o arquivo existe. Estas condições são implementadas na expressão booleana da Linha-08 cujo valor é atribuído a variável interna existe (retorno da função).

Listagem 52

```
01  function existe(nome: string): boolean;
02  var F : file;
03  begin
04    {$I-}
05    assign (F, nome);
06    reset (F);
07    {$I+}
08    existe := (IoResult = 0) and (nome <>
09  '');
10    close(F);
    end;
```

Para fechar um fluxo aberto (e descarregar o último conteúdo em memória principal) usa-se o comando close com sintaxe,

```
close(<fluxo>);
```

Para forçar o descarregamento da memória principal para a secundária antes mesmo do fechamento do arquivo, use o comando flush cuja sintaxe é,

```
flush(<fluxo>);
```

Para renomear um arquivo utiliza-se o comando rename. Ele requer dois argumentos sendo o primeiro o nome do fluxo do arquivo que se deseja renomear e o segundo uma string com o novo nome (as regras de nomes são regidas pelo sistema operacional que o programador utiliza). Este comando requer a vinculação de assign mas deve ser chamado sem que o arquivo seja aberto (por reset, rewrite ou append). No exemplo,

```
var
    F: file;
begin
    assign(F, 'teste.dat');
    rename(F, 'novo.dat');
end.
```

o arquivo teste.dat é renomeado para novo.dat.

Para excluir um arquivo definitivamente utiliza-se o comando erase. Ele possui como argumento único o fluxo que se vincula ao arquivo que deve ser eliminado da memória secundária. De forma análoga a rename, erase requer a vinculação de assign mas deve ser chamado sem que o arquivo esteja aberto. No exemplo,

```
var
    F: file;
begin
    assign(F, 'teste.dat');
    remove(F);
end.
```

o arquivo teste.dat é definitivamente excluído.

4. Utilizando Arquivos Binários

Os comandos read e write, utilizados até agora para leitura via entrada padrão e escrita para a saída padrão, possuem versões para leitura e escrita em arquivos. Adicionalmente eles recebem como primeiro argumento o fluxo de onde serão lidos ou para onde serão escritos dados. Os demais argumentos correspondem a uma lista de variáveis ou constantes cujos conteúdos deverão ser copiados, sem processamento adicional, para o arquivo vinculado ao fluxo no primeiro argumento. Detalhes de formatação, como espaços, tabulações, novas linhas ou controle de casas decimais não possuem funcionalidade no âmbito dos arquivos binários.

No exemplo,

```
write(F, x, y);
```

se F for um fluxo binário então o conteúdo das variáveis x e y serão escritos sequencialmente no arquivo vinculado a F. Similarmente em,

```
read(F, x, y);
```

dois dados consecutivos do arquivo vinculado ao fluxo F são lidos através das variáveis x e y.

As operações de leitura de read e escrita de write ocorrem na posição corrente do arquivo. Depois que a operação se encerra a posição corrente tem sido deslocada do total de bytes processados. Assim, por exemplo, escrever continuamente dados para o final do arquivo mantém a posição corrente sempre no final. No programa da Listagem 53 100 valores inteiros são gerados aleatoriamente e gravados num arquivo de fluxo F. As 100 chamadas a write na Linha-10 copiam cada valor inteiro para o arquivo e ao mesmo tempo deslocam a posição corrente para o novo fim de arquivo. Daí na próxima iteração não haverá sobreposição, mas sim concatenação do novo inteiro.

Listagem 53

```
01 program arquivo_de_inteiros;
02 var
03     f: file of integer;
04     i: integer;
05 begin
06     assign(f, 'teste.dat');
07     rewrite(f);
08     randomize;
09     for i := 1 to 100 do
10         write(f, random(100));
11     close(f);
12 end.
```

Os comandos ramdomize e random, unidade system, são utilizados para gerar números pseudo aleatórios em Pascal. ramdomize é um procedimento sem argumentos que deve ser chamado uma única vez antes de chamadas futuras a random para possibilitar o máximo de aleatoriedade possível. As chamadas a random requerem um ou nenhum argumento e retornam números pseudo aleatórios. Quando não há argumento o valor pseudo aleatório gerado é real e está entre 0.0 e 1.0. Se o argumento existe, ele deve ser um inteiro n e a saída gerada um inteiro entre 0 e n-1.

Para verificar ou modificar manualmente a posição corrente de um arquivo, a unidade system dispõe os comandos a seguir,

eof	Abreviação de *end of file* (fim de arquivo). Recebe um fluxo como argumento e retorna verdadeiro se a posição corrente atingiu o final do arquivo vinculado. Do contrário retorna falso.
filepos	Recebe um fluxo e retorna o valor da posição corrente (int64) do arquivo vinculado.
seek	Modifica a posição corrente do arquivo cujo fluxo é passado como primeiro argumento. A nova posição (int64) é dada como segundo argumento.

Para ler o conteúdo do arquivo gerado pelo programa da Listagem-6.2, construiu-se o programa da Listagem 54. Neste novo programa reset, na Linha-07, acessa o arquivo em disco e define a posição corrente para o início do arquivo. O laço while da Linha-08 utiliza eof para examinar sucessivamente se a posição corrente de f já atingiu o fim do arquivo. As chamadas a read, Linha-09, fazem tanto a variável i assumir em cada iteração do laço cada um dos inteiros armazenados no arquivo quanto avançar automaticamente a posição corrente. Cada valor de inteiro lido através de i é impresso pelo laço em saída padrão (Linha-10).

Listagem 54

```
01  program lendo_arquivo_de_inteiros;
02  var
03      f: file of integer;
04      i: integer;
05  begin
06      assign(f, 'teste.dat');
07      reset(f);
08    while not eof(f) do begin
09          read(f, i);
10          write(i, ' ');
11      end;
12      close(f);
13  end.
```

Se duas variáveis inteiras, a e b, tomarem o lugar de i no programa da Listagem-6.3, o laço principal poderá ser equivalentemente escrito assim,

```
while not eof(f) do begin
    read(f, a, b);
    write(a, ' ', b, ' ');
end;
```

o que demonstra a flexibilidade, no contexto dos arquivos, do comando read.

Para mudar manualmente a posição corrente em um arquivo binário utiliza-se o comando seek. O primeiro argumento é o fluxo vinculado ao arquivo e o segundo argumento é o valor da nova posição corrente. Em Pascal, ao invés de bytes, a posição corrente é medida em número de registros. Um registro é um bloco cuja memória é igual a do tipo base em arquivos tipados e 128-bytes em arquivos não tipados. Seek não funciona com arquivos de texto. Zero é o valor da posição inicial de um arquivo ao passo que o n-ésimo registro possui posição n-1.

O total de registros de um arquivo é retornado pelo comando filesize cujo único argumento é o fluxo vinculado ao arquivo que se deseja saber o número de registros. Por exemplo, para apontar para o octogésimo inteiro do arquivo gerado pelo programa da Listagem-6.2, usa-se,

```
seek(f, 79);
```

Quando a posição corrente de um arquivo binário não está no fim do arquivo então uma operação de escrita causa uma *sobreposição* como ilustrado na Figura 11. Por exemplo, para substituir o quadragésimo inteiro do arquivo gerado pelo programa da Listagem 53 usa-se,

```
seek(F, 39);
write(F, 123);
```

Esta operação irá gravar o valor 123 sobre aquele que está na quadragésima posição do arquivo com fluxo F. O tamanho do arquivo não é modificado e a posição corrente final é a quadragésima primeira.

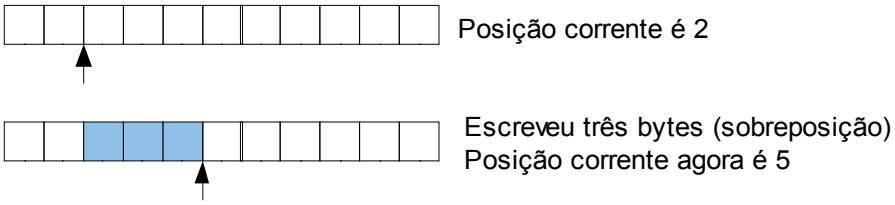


Figura 11

Listagem 55

```
01  procedure inserir(fnome: string; pos, x: integer);
02  var F: file of integer;
03      tmp, a: integer;
04  begin
05      assign(F, fnome);
06      reset(F);
07      seek(F, pos);
08      read(F, tmp);
09      seek(F, pos);
10      write(F, x);
11      while not eof(F) do begin
12          read(F, a);
13          seek(F, filepos(F)-1);
14          write(F, tmp);
15          tmp := a;
16      end;
17      write(F, tmp);
18      close(F);
19  end;
```

Caso seja necessário inserir dados *no meio* do arquivo então o comportamento padrão de sobreposição não será desejável. Uma forma de resolver o problema é proposta pela função na Listagem 55. Os argumentos de entrada são o nome do arquivo, lido pela string fnome, o valor da posição onde deve haver a inserção, lido pelo inteiro pos e o item de inserção lido pelo inteiro x (supõe-se um arquivo binário de inteiros). O primeiro passo é abrir o arquivo em modo de leitura (Linhas 05 e 06) e Ir para a posição onde deve ocorrer a inserção do novo registro (Linha-07).

Em seguida faz-se uma cópia do registro nesta posição em tmp (Linha-08) e retorna-se a posição de inserção (a leitura causou avanço da

posição corrente e por essa razão é necessário regredir manualmente com seek, conforme Linha-09). A inserção acontece em seguida por sobreposição (Linha-10). O item sobreposto não foi perdido porque possui cópia em tmp.

No laço entre as Linhas 11 e 16 a variável a recebe uma copia do registro na posição corrente (Linha-12) e em seguida regride a essa posição (Linha-13) para gravar o último valor em tmp (Linha-14). Com o valor de a atribuído a tmp (Linha-15) fecha-se um ciclo de permutações de dados que provocam o deslocamento em uma posição para a frente de todos os registros que sucedem a posição de inserção.

Quando o laço se encerra, o valor em tmp é o do antigo último registro do arquivo. Sua concatenação ao arquivo (Linha-17) conclui a transposição de dados e promove o aumento real do arquivo. Veja Figura 12.

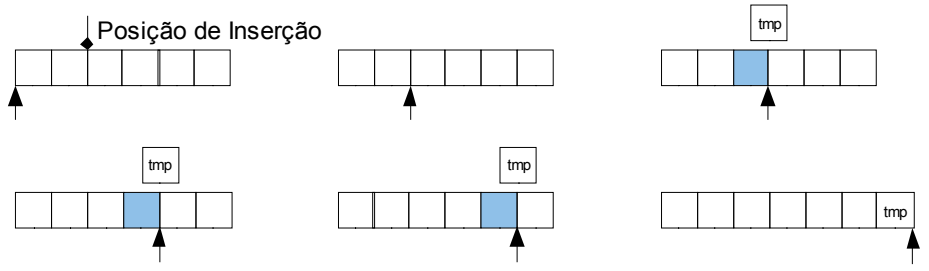


Figura 12

A remoção de dados em arquivos binários é realizada por operações de *truncagem*. Uma truncagem corresponde a eliminação de todo conteúdo do arquivo a partir da posição corrente. Para efetuar uma truncagem em Pascal usa-se o comando truncate repassando como único argumento o fluxo vinculado ao arquivo que se deseja truncar. Por exemplo para eliminar os três últimos inteiros de um arquivo binário de inteiros chamado teste.dat, implementa-se,

```
assign(F, 'teste.dat');

reset(F);

seek(F, filesize(F) - 3);

truncate(F);

close(F);
```

Observe que a expressão filesize(F)-3 calcula a posição no arquivo vinculado a F do terceiro inteiro contado de trás para frente. A chamada seguinte à truncate elimina os três inteiros reduzindo o arquivo.

A eliminação de registros internos a um arquivo binário requer que eles primeiramente sejam *movidos* para o final do arquivo antes da truncagem (veja esquema na Figura 13). Para movimentar um registro indesejado para o

final do arquivo deve-se efetuar sucessivas trocas entre ele e o registro logo à esquerda. Isso deslocará tal registro em direção ao final do arquivo ao mesmo tempo que trará em uma posição para trás todos os demais registros que inicialmente se encontravam a sua frente. Na Figura 14 o registro indesejado possui valor 9. Após seis trocas ele atinge a última posição, onde é eliminado.

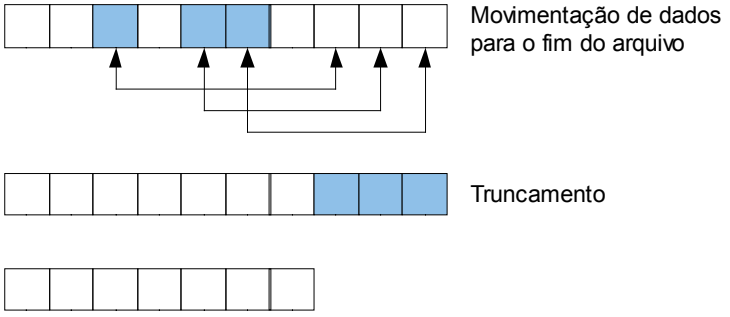


Figura 13

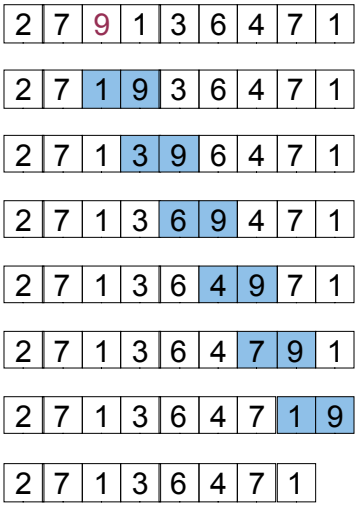


Figura 14

Listagem 56

```
01 procedure remover(fnome: string; pos: integer);
02 var F: file of integer;
03     a, b: integer;
04 begin
05     assign(F, fnome);
06     reset(F);
07     seek(F, pos);
08     while filepos(F) < filesize(F)-1 do begin
09         read(F, a, b);
10         seek(F, filepos(F)-2);
11         write(F, b, a);
12         seek(F, filepos(F)-1);
13     end;
14     truncate(F);
15     close(F);
16 end;
```

O procedimento da Listagem 56 implementa a movimentação com remoção de um registro interno a um arquivo binário de inteiros. Os argumentos de entrada são o nome do arquivo (fnome) e a posição onde se localiza o registro indesejado (pos). Primeiramente o arquivo é vinculado ao fluxo F e a posição corrente mudada para a do registro indesejado (Linhas 05, 06 e 07). O laço entre as Linhas 08 e 13 efetua as trocas dos registros vizinhos.

Para tanto são lidos de cada vez dois registros usando as variáveis a e b (Linha-09). Em seguida regride-se duas posições para trás (Linha-10) e, por sobreposição, grava-se b e a nesta ordem (Linha-11) (troca propriamente dita). Para que na iteração seguinte a nova troca possa ocorrer corretamente, deve-se regredir uma posição para trás (Linha-12).

O teste de continuação do laço na Linha-08 verifica se a posição corrente do arquivo atingiu a *penúltima* posição haja vista que são lidos dois registros de cada vez. O uso de eof neste caso causaria uma tentativa de leitura além do fim do arquivo. Quando o laço se encerra a posição corrente terá atingido o último registro que por sua vez possuirá o valor indesejado. O truncamento na Linha-14 elimina o último registro do arquivo.

Todas as operações descritas nessa sessão se estendem a quaisquer tipos base que definam um fluxo. No seguinte exemplo,

```
type
    aluno = record
        nome: string[25];
        idade: byte;
        media_final: real;
    end;
var
    G: file of aluno;
```

F define um fluxo para um arquivo binário cujos registros são de um tipo estrutura heterogênea. Cada registro utiliza 30-bytes (25 + 1 + 4) de forma que o avanço, ou recuo, da posição corrente ocorre em múltiplos desse valor, ou seja, a posição 0 está a zero bytes do início do arquivo, a posição um a 30-bytes, a posição dois a 60-bytes e assim por diante. No exemplo,

```
x.nome := 'Raul';
x.idade := 30;
x.media_final := 9.5;
write(G, x);
```

a variável x, de tipo aluno, é inicializada e seu novo conteúdo gravado no arquivo binário de fluxo G. Se o mesmo arquivo for aberto para leitura, estas informações poderão ser resgatadas usando-se,

```
read(F, x);

writeln('Nome: ', x.nome,

        ', idade: ', x.idade,

        ', Media final: ', x.media_final:0:1);
```

5. Utilizando Arquivos de Texto

Assim como acontece com a saída padrão, a escrita de dados em um arquivo texto efetua em bastidores uma conversão de dados para o formato texto (string) antes da impressão propriamente dita. A gravação é feita pelos comandos write/writeln cujo primeiro argumento deve ser um fluxo de arquivo de texto. Os demais argumentos, independente de seus formatos, são um a um convertidos em strings formatadas para por fim serem gravadas no arquivo. A notação com dois-pontos (:), vista rapidamente no Capítulo-1, é estendida aqui para permitir essa formatação.

Para formatar um valor inteiro em write/writeln usa-se a notação i:n onde i define o inteiro e n o número de caracteres que a versão em texto do número i deve ter. Se n for maior que o número de caracteres mínimo para representar i então ocorrerá preenchimento com caracteres em branco à esquerda. Na chamada,

```
x := 5;

writeln('|', x:4, '|');
```

onde x é inteiro, imprime em saída padrão,

```
|  5|
```

Para formatar um valor real em write/writeln usa-se a notação r:m:d onde r define um valor real, m o total de caracteres que a string final deve ter e d o número de casas decimais que devem ser consideradas. Se m for maior que o mínimo de caracteres para representar o real então ocorrerá preenchimento à esquerda com caracteres em branco. No exemplo,

```
x := 3.14159;

writeln('|', x:5:2, '|');
```

onde x é real, imprime,

```
|  3.14|
```

Observe que o ponto (.) também entra na contagem do total de caracteres da string e por essa razão entre | e 3 existe apenas um caractere em branco.

Para formatar uma string em write/writeln usa-se a notação s:n onde s define uma string e n o total de caracteres que a string final deve possuir. Se o valor em n for maior que o comprimento de s, haverá preenchimento com caracteres em branco à esquerda. No exemplo,

```
x := 'carro';

writeln('|', x:10, '|');
```

onde x é uma string, imprime,

```
|     carro|
```

Para ilustrar o uso da notação com dois-pontos em arquivos de texto, consideremos o exemplo seguinte,

```
program alo_mundo;
var
    F: text;
    x: integer;
    q: real;
begin
    assign(F, 'saida.txt');
    rewrite(F);
    x := 123;
    q := 34.0/125;
    writeln(F, x:5, q:12:5, 'Ola mundo!!':20);
    close(F);
end.
```

Este programa gera um arquivo de texto chamado saida.txt cujo conteúdo é,

```
123     0.27200         Ola mundo!!
```

writeln insere adicionalmente um caractere de nova linha. Assim uma anexação futura (uso de append) ocorrerá na linha seguinte do arquivo. Por exemplo a execução de,

```
assign(F, 'saida.txt');

append(F);

writeln(F, x:5, q:12:5, 'Ola mundo!!':20);

close(F);
```

Adiciona texto à saida.txt cujo conteúdo final fica,

```
123     0.27200         Ola mundo!!
123     0.27200         Ola mundo!!
```

Ao contrário de um arquivo binário, saida.txt pode ser visualizado em um editor de texto (como o *notepad*, *word* no Windows ou *pico*, *gedit*, *kedit* em Linux).

Listagem 57

```
01  program Alunos;
02
03  procedure inserir(fnome, nome: string; idade: byte; media: real);
04
05  var F: text;
06  begin
07      {$I-}
08      assign(F, fnome);
09      reset(F);
10      {$I+}
11      if ioResult=0 then
12          append(F)
13      else
14          rewrite(F);
15      writeln(F, nome:25, idade:3, media:5:1);
16    end;
17
18  var s: string;
19      i: byte;
20      m: real;
21      ch: char;
22  begin
23      repeat
24          write('Nome> ');  readln(s);
25          write('Idade> '); readln(i);
26          write('Media> '); readln(m);
27          inserir('aluno.txt', s, i, m);
28          writeln('continuar S/N?');
29          readln(ch);
30      until (ch='n') or (ch='N');
31  end.
```

O programa da Listagem 57 ilustra a inserção de dados num arquivo texto chamado aluno.txt. Cada linha do arquivo deve conter sequencialmente o nome de aluno, sua idade e sua média final. A inserção dos dados de um novo aluno é feita pelo procedimento inserir que recebe como argumentos o nome do arquivo base de inserção (argumento fnome), o nome do aluno (argumento nome), a idade (argumento idade) e a média final (argumento media).

Na primeira parte do procedimento é testada a existência do arquivo base (Linhas 06 a 09). Caso ele exista, é aberto em modo de anexação (Linha-11) senão ele é criado (Linha-13). A inserção de dados propriamente dita ocorre na Linha-14 e o fechamento do arquivo na Linha-15.

O programa principal mantém um laço entre as Linhas 23 e 30 que requisita constantemente por dados de novos alunos e os insere em aluno.txt (Linha-27). Cada chamada repassa o nome deste arquivo e os dados do novo aluno, abre o arquivo, grava e por fim o fecha.

No final de cada inserção o usuário tem a opção de parar ou continuar dependendo do valor que fornecer a ch (Linhas 29 e 30). Um aspecto de aluno.txt após algumas inserções é listado a seguir,

```
          joao da silva 25  9.5
    maria joana da silva 23  8.0
    antonio manuel alves 31  9.0
     isabel moura brasil 23  7.0
     emanuel lucas sousa 30  8.7
     arthur brandao lima 22  5.6
         alfedro pacheco 29  7.8
```

A leitura de um arquivo de texto em Pascal é realizada linha a linha usando-se o comando readln com dois argumentos sendo o primeiro o fluxo vinculado ao arquivo que se deseja ler e o segundo uma string (com tamanho suficiente para caber a linha de maior comprimento). O final de um arquivo texto deve ser testado pelo comando seekeof similar a eof em arquivos binários (a sintaxe é equivalente). No exemplo,

```
var F: text;
    linha: string;
begin
    {$I-}
    assign(F, 'aluno.txt');
    reset(F);
    {$I+}
    if ioResult=0 then
        while not seekeof(F) do begin
            readln(F, linha);
            writeln(linha);
        end;
    close(F);
end;
```

o arquivo aluno.txt é lido linha a linha através da variável linha e impresso na saída padrão.

Arquivos de texto gerados pela transformação de dados numéricos em strings não podem ser *diretamente* lidos como contêineres de números. Para trazer dados de volta a formatos como real ou integer o programador precisará usar o comando val (veja Capítulo-2) em cada string numérica gravada.

Um arquivo de texto pode ser lido como uma sequência de caracteres. Operacionalmente é como lidar com um arquivo binário cujos registros são caracteres. O reconhecimento de final de linha nesta nova abordagem é feita pelo comando seekeofln que identifica o caractere especial de nova linha. O laço,

```
while not seekeoln(F) do begin
    read(F, ch);
    write(ch);
end;
```

onde ch é uma variável de tipo char, imprime em saída padrão o conteúdo da primeira linha de um arquivo de texto ao qual o fluxo F está vinculado.

Síntese da Parte 3

No Capítulo 1 foi estudada a modularização que incluiu noções sobre controle de execução em um processo monothread, aspectos sobre a passagem de argumentos em procedimentos e funções, noções sobre uso e aplicações da recursividade, a importância do aninhamento modular e por fim a compilação de módulos afins numa unidade Pascal. No Capítulo 2 foi estudado o sistema padrão de manipulação de arquivos da Linguagem Pascal incluindo a noção de fluxo de informações entre memória principal e secundária. Os principais comandos desde sistema foram apresentados convenientemente em suas respectivas categorias (dos arquivos binário e de texto) e adicionalmente exemplificados para melhor fixar suas funcionalidades.

Atividades de avaliação

1. Construir programa que gere um arquivo binário contendo 1000 inteiros aleatórios entre 200 e 5000.
2. Construir programa que crie um arquivo contendo apenas inteiros pares oriundos de um arquivo gerado pelo programa da questão anterior.
3. Construir programa que crie uma cópia de um arquivo dado, mas com nome distinto.
4. Construir programa que leia um arquivo gerado pelo programa do problema-1 e inverta sua sequencia original de inteiros.
5. Uma estrutura heterogênea chamado ponto mantém dois campos, x e y, que representam uma coordenada no plano. Utilizando este tipo de dados, construir programa que crie um arquivo binário contendo 200 pontos da função $f(x) = 5x^3 - 7x$ com x no intervalo [5, 50]. Escolha valores de x equidistantes.
6. Construir programa que leia o arquivo gerado na questão anterior e crie uma versão texto dele.
7. O procedimento da Listagem-6.5 ilustra como movimentar um registro para o fim do arquivo e depois eliminá-lo por truncamento. Reconstruir o procedimento de forma que seja possível movimentar

mais de um registro para o final do arquivo para eliminá-los com uma única truncagem.

8. Utilizando o programa da Listagem 57 como base, construir um programa que forneça ao usuário as possibilidades de cadastro de novo aluno e de visualização da listagem, em saída padrão, de alunos cadastrados.

9. Inserir no programa do problema 8 a capacidade de busca por nome, ou seja, o usuário fornece um fragmento de nome de aluno e o programa lista as linhas que contêm aquele fragmento.

10. Inserir no programa do problema 8 a possibilidade de remover um registro (remover um aluno).

Referências

CANNEYT , Michaël Van, **Reference guide for Free Pascal**, Document version 2.4 , March 2010

http://www.freepascal.org/docs-html/

Sobre o autor

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

Computação

Fiel a sua missão de interiorizar o ensino superior no estado Ceará, a UECE, como uma instituição que participa do Sistema Universidade Aberta do Brasil, vem ampliando a oferta de cursos de graduação e pós-graduação na modalidade de educação a distância, e gerando experiências e possibilidades inovadoras com uso das novas plataformas tecnológicas decorrentes da popularização da internet, funcionamento do cinturão digital e massificação dos computadores pessoais.

Comprometida com a formação de professores em todos os níveis e a qualificação dos servidores públicos para bem servir ao Estado, os cursos da UAB/UECE atendem aos padrões de qualidade estabelecidos pelos normativos legais do Governo Federal e se articulam com as demandas de desenvolvimento das regiões do Ceará.

ISBN 978-85-78264-49-9

Universidade Estadual do Ceará

UAB UNIVERSIDADE ABERTA DO BRASIL

CAPES