BORLAND • OSBORNE/McGRAW-HILL

TURBO PASCAL

AVANÇADO

GUIA DO USUÁRIO

Inclui Versões 4.0 e 5.0

BEST SELLER

McGraw-Hill

HERBERT SCHILDT

TURBO PASCAL AVANÇADO

HERBERT SCHILDT

Tradução

Marcos Piani
Paulo Cândido de Oliveira Filho

Revisão Técnica

Luciano Gama de Souza Ramalho
Consultor em Informática

McGraw-Hill
São Paulo
Rua Tabapuã, 1.105, Itaim-Bibi
CEP 04533
(011) 881-8604 e (011) 881-8528

Rio de Janeiro • *Lisboa* • *Porto* • *Bogotá* • *Buenos Aires* • *Guatemala* • *Madrid* • *México* • *New York* • *Panamá* • *San Juan* • *Santiago*

Auckland • Hamburg • Kuala Lumpur • London • Milan • Montreal • New Delhi • Paris • Singapore • Sydney • Tokyo • Toronto

Do original

Advanced Turbo Pascal®

Editor: MILTON MIRA DE ASSUMPÇÃO FILHO

Coordenadora de Revisão: Daisy Pereira Daniel
Supervisor de Produção: José Rodrigues
Composição e Arte-final: JAG Composição Editorial e Artes Gráficas Ltda.

Dados de Catalogação na Publicação (CIP) Internacional
(Câmara Brasileira do Livro, SP, Brasil)

Schildt, Herbert.
S36t Turbo Pascal avançado / Herbert Schildt ; tradução Marcos Piani, Paulo Cândido de Oliveira Filho ; revisão técnica Luciano Gama de Sousa Ramalho. — São Paulo : McGraw-Hill, 1988.

1. PASCAL (Linguagem de programação para computadores) 2. Turbo Pascal (Programa de computador) I. Título.

CDD-001.6425
88-0759 -001.6424

Índices para catálogo sistemático:

1. PASCAL : Linguagem de programação : Computadores . Processamento de dados 001.6424
2. Turbo Pascal : Computadores : Programas : Processamento de dados 001.6425

Para os meus pais

SUMÁRIO

INTRODUÇÃO

Este livro é exatamente o tipo de livro de programação que eu gostaria de ler. Há vários anos, quando comecei a programar, procurei um livro que trouxesse algoritmos para problemas tais como ordenações, listas encadeadas, simulações e análise de expressões, de uma maneira clara e direta. Eu também queria informações sobre os problemas mais comuns encontrados durante a programação. Nunca encontrei o livro que procurava. Agora, já um programador experiente, eu escrevi tal livro.

Este livro trata de vários assuntos, e traz muitos algoritmos, procedimentos, funções e rotinas escritos na linguagem Turbo Pascal. O Turbo Pascal é o padrão em Pascal para microcomputadores. Existem compiladores Turbo Pascal disponíveis para praticamente qualquer microcomputador. Eu usei o compilador Turbo Pascal do IBM PC nos exemplos deste livro. Entretanto, a maior parte dos exemplos pode ser compilada e executada em qualquer compilador Pascal.

O Capítulo 1 é uma breve revisão da linguagem. O Capítulo 2 discute ordenação, incluindo matrizes e arquivos de disco. O Capítulo 3 trata de pilhas, filas, listas encadeadas e árvores binárias. Talvez isto pareça demais para ser discutido em um só capítulo, mas estes conceitos juntos formam uma unidade bastante sólida.

O Capítulo 4 fala dos métodos de alocação dinâmica, e o Capítulo 5 apresenta um resumo do interfaceamento com o sistema operacional e rotinas em linguagem de máquina. O tópico do Capítulo 6 é estatística, incluindo um programa completo. O Capítulo 7 é sobre códigos, cifras, e inclui um pequeno resumo sobre a história da criptografia. O Capítulo 8 descreve vários geradores de números aleatórios e discute seu

uso em duas simulações: as filas de caixa de um supermercado e o gerenciamento de uma carteira de ações pelo método *random-walk*.

O Capítulo 9, o meu favorito, contém um analisador recursivo descendente completo. (Há alguns anos eu teria dado qualquer coisa por um programa como este.) Este capítulo é para quem precisa analisar expressões. O Turbo Database Toolbox e o Graphix Toolbox, duas extensões muito úteis do Turbo Pascal, são comentados nos Capítulos 10 e 11, respectivamente. O assunto do Capítulo 12 é eficiência, portabilidade e depuração de programas. O livro termina com um apêndice sobre a conversão de programas de outras linguagens para Turbo Pascal e outro sobre as diferenças entre as versões 3.0 e 4.0.

H. S.

CAPÍTULO 1

UMA REVISÃO DE PASCAL

Este livro utiliza a apresentação e resolução de problemas para ilustrar conceitos avançados da linguagem de programação Pascal, dando especial ênfase ao Turbo Pascal da Borland. Cada capítulo trata de uma área diferente, apresentando soluções para tarefas de programação, e analisando o estilo e a estrutura de cada solução apresentada. Assim, tópicos avançados de Pascal são explorados, bem como a teoria de programação por trás de cada algoritmo. Para tirar total proveito deste livro você deve ter uma certa experiência com programação em Pascal, saber operar o sistema Turbo Pascal e ser capaz de digitar e compilar programas simples.

Para melhor identificação, todos os nomes de variável, palavras reservadas do Pascal, nomes de funções e de procedimentos estarão em negrito.

AS ORIGENS DO PASCAL

O Pascal, cujo nome é uma homenagem ao filósofo Blaise Pascal, foi inventado por Niklaus Wirth no início da década de 70. Originalmente, a linguagem de programação Pascal foi criada para ser uma linguagem educacional, introdutória, para ajudar programadores iniciantes a desenvolver bons hábitos, permitindo a elaboração de programas claros, concisos e estruturados. Antes do Pascal, a introdução à programação se fazia, em geral, através do Fortran, uma linguagem desestruturada e bem mais antiga. Wirth acreditava que muitos dos erros mais comuns de programação poderiam ser evitados com

o uso de uma linguagem estruturada por blocos, e que trouxesse, embutido, um severo controle de tipos. O Pascal é assim. Isto significa que variáveis e constantes de um tipo não podem ser livremente misturadas com variáveis e constantes de outro tipo. Por exemplo, ambos os termos de uma atribuição devem ser do mesmo tipo (com uma exceção: a variável do lado esquerdo pode ser do tipo **real** quando a expressão do lado direito for **inteira**). O controle exaustivo de tipos ajuda a prevenir erros, pois força o programador a elaborar expressões sempre compatíveis.

No Turbo Pascal, nomes de variáveis consistem em uma letra ou em um sublinhado (_), seguidos de letras, dígitos ou sublinhados (o comprimento máximo permitido é de 127 caracteres).

Todas as seqüências de caracteres (*strings*) são colocadas entre apóstrofos (diferente de outras linguagens, tal como o Basic, onde aspas são usadas).

O Pascal permite que se declarem dados escalares de cinco tipos:

Tipo	**Significado**
char	Um caractere de 1 byte de tamanho.
byte	Um inteiro de 1 byte de tamanho, com valor entre 0 e 255.
integer	Um inteiro com valor entre –32768 e 32767.
real	Um número decimal, com até 11 dígitos de precisão.
Boolean	TRUE (verdadeiro) ou FALSE (falso).

Apesar da semelhança, o controle de tipos do Pascal não permitirá que **byte** e **char** sejam intercambiados. Variáveis do tipo **byte** são usadas no lugar de variáveis do tipo **integer**, para acelerar a execução, pois em algumas máquinas operações com *bytes* são mais rápidas que operações com inteiros. Além dos escalares, o Pascal admite matrizes (*arrays*) de cada um dos cinco tipos.

Apesar de seu propósito educacional original, a facilidade de uso e o enfoque altamente estrutural converteram ao Pascal muitos programadores experientes de outras linguagens. Quando começou a ser usado mais freqüentemente na confecção de softwares comerciais, o Pascal necessitava de muitas extensões. Antes do Turbo Pascal, cada compilador trazia um grupo diferente de extensões, acarretando sérias dificuldades à transposição de programas. Entretanto, como o Turbo Pascal se tornou um padrão, muitas de suas extensões podem, agora, ser usadas livremente.

O PASCAL ENQUANTO LINGUAGEM ESTRUTURADA

O Pascal é uma linguagem estruturada, semelhante em alguns aspectos ao Algol e ao C. O que caracteriza uma linguagem estruturada é a possibilidade de dividir a informação em compartimentos estanques, independentes um do outro. Ou seja, todas as informações e instruções necessárias à realização de uma tarefa qualquer podem ser seccionadas e escondidas do resto do programa. Isto, em geral, é feito através do uso de sub-rotinas, algumas vezes chamadas *subprogramas*, com *variáveis locais* temporárias. Deste modo, é possível evitar que outras partes do programa sejam afetadas por ocorrências internas a uma sub-rotina. O uso excessivo de *variáveis globais*, conhecidas por todo o programa, permite o aparecimento de efeitos colaterais imprevistos e indesejáveis. No Pascal, todos os subprogramas são funções discretas ou procedimentos.

As funções e os procedimentos são os blocos fundamentais do Pascal. Num programa qualquer, uma tarefa específica pode ser definida e escrita em separado, numa função ou num procedimento. Depois de corrigida, tal função ou procedimento funcionará de modo apropriado, sem causar efeitos colaterais em outras partes do programa. Todas as variáveis declaradas naquela função ou procedimento são conhecidas apenas ali.

Em Pascal, estruturas são criadas através de blocos de instruções. Um *bloco de instruções* é um grupo de comandos conectados logicamente, de modo a poderem ser tratados como uma unidade. Um bloco é criado quando se colocam linhas de instrução entre as palavras **begin** e **end**, como ocorre a seguir:

```
if x < 10  then
begin
        WriteLn ('numero muito baixo, tente de novo');
        ResetContaX (-1);
end;
```

Neste exemplo, os comandos após **if** e entre **begin** e **end** serão executados se **x** for menor que 10. O primeiro comando escreve na tela a linha **numero muito baixo, tente de novo**, e a segunda linha chama um procedimento. Estes dois comandos podem ser tratados como uma unidade, um bloco de instruções. Eles estão ligados, não havendo maneira de executar um deles sem que o outro também o seja. Em Pascal, cada comando pode ser um comando simples ou um bloco de comandos. O uso de blocos de comandos permite a criação de programas bastante legíveis e com uma lógica fácil de acompanhar. Os blocos também ajudam na confecção de programas melhores e com menos erros, visto que o sentido e a função de cada bloco são claros.

Com o crescimento da popularidade do Pascal, cada vez mais programadores passaram a fazer em Pascal todos os seus programas. A existência de compiladores Pascal para quase todos os computadores torna fácil e barato o processo de transposição de programas entre máquinas diferentes.

No Turbo Pascal, cada programador pode criar sua própria biblioteca de funções, adequada a seus usos e a seu estilo. E, como o Turbo Pascal permite que arquivos externos sejam incluídos no programa durante a compilação, grandes projetos podem ser mais facilmente administrados.

EXTENSÕES DO TURBO PASCAL

Nesta seção serão abordados os acréscimos mais importantes, realizados pela Borland ao Pascal padrão, quando da criação do Turbo Pascal. Portanto, se você já tiver alguma experiência com o Turbo Pascal, poderá passar diretamente ao Capítulo 2.

A decisão de expandir uma linguagem deve sempre estar baseada em razões sólidas. Cada acréscimo feito reduzirá a portabilidade dos programas escritos nesta linguagem, ou seja, acréscimos afastam a linguagem do padrão. Em geral, uma linguagem é expandida porque faltam a ela funções necessárias. Ao Pascal faltavam algumas características que um programador profissional precisa e espera encontrar numa linguagem, tais como manipulação de seqüências de caracteres, endereçamento absoluto e facilidades de sobreposição. O Pascal era uma linguagem educacional. A Borland fez dele uma linguagem de uso geral.

Os acréscimos implementados pela Borland se concentram em duas áreas, comandos e procedimentos predefinidos. Primeiro serão discutidos os comandos, depois os procedimentos, e finalmente algumas diferenças entre o Pascal padrão e o Turbo Pascal.

ABSOLUTE

Uma das mais importantes expansões do Turbo Pascal é o modificador **absolute.** No Pascal padrão era impossível obrigar uma variável a residir num endereço específico da memória. Isto não era necessário numa linguagem educacional. Entretanto, numa linguagem de desenvolvimento de *software* é essencial.

Imagine um programa que use uma área reservada da memória; a memória de vídeo por exemplo. Para que isto seja feito, deve ser possível especificar um determinado endereço para uma variável. No Turbo Pascal, basta adicionar o modificador **absolute** à declaração da variável, após declarar o tipo da mesma. No IBM PC e compatíveis o endereço é dado na forma *segmento:offset*. Em outras máquinas usa-se a forma padrão de endereçamento. Num IBM PC, uma variável real, residente no segmento 0, offset 9000, é declarada do seguinte modo:

```
contador: real absolute 0:9000;
```

Muito cuidado ao declarar variáveis usando o **absolute**; você pode arruinar seu programa, seu sistema operacional, ou ambos.

O **absolute** também pode ser usado para fazer com que duas variáveis dividam o mesmo espaço da memória, bastando para isto usar o nome de uma das variáveis como alvo do **absolute** da outra. No exemplo abaixo, **teste** e **contador** dividem o mesmo espaço:

```
teste: integer;
contador: integer absolute teste;
```

EXTERNAL

No Pascal padrão, um programa não pode usar funções ou procedimentos escritos em outra linguagem. Entretanto, existem muitas ocasiões em que isto é útil e necessário. Um sintetizador de voz, por exemplo, pode ser controlado por um módulo especial escrito em *assembly*. Para isto existe no Turbo Pascal o modificador **external**, cuja função é permitir que sub-rotinas externas sejam incluídas em um programa.

O **external** informa ao programa em Pascal que uma rotina escrita em *assembly*, residente num determinado arquivo, será usada. O subprograma externo declarado não deve estar escrito no programa em Pascal; você deve apenas especificar os parâmetros (se houver) e o nome do arquivo de disco onde tal subprograma pode ser encontrado. Para declarar, por exemplo, um procedimento chamado **Falar** com um parâmetro tipo seqüência de caracteres (*string*) como **external**, num arquivo **fala**, você escreveria assim:

```
procedure Falar (palavra:string); external 'fala';
```

Uma função é declarada de modo semelhante.

Para utilizar subprogramas externos você, além de saber programar em *assembly*, deve conhecer a arquitetura de seu computador. Posteriormente, examinaremos com mais detalhes o uso de rotinas e comandos em *assembly* dentro de programas escritos em Turbo Pascal.

INLINE

O comando **inline** permite a inclusão de instruções em linguagem de máquina num programa, característica que pode ser útil tanto para controlar dispositivos especiais de *hardware* quanto para escrever rotinas curtas em linguagem de máquina, usando o Turbo Pascal como apoio. O **inline** será discutido com mais detalhes no Capítulo 5, mas o método básico de uso é o que segue.

O código de máquina a ser inserido no programa é escrito entre parênteses, após a palavra **inline**. Cada byte ou palavra são separados por uma barra (/). Os sinais "+" e "–" podem ser usados para somar ou subtrair. Um asterisco indica referência de *counter location*. Todo o código é escrito em números, isto é, mnemônicos não podem ser usados. Como **inline** é um comando, a linha termina com um ponto-e-vírgula. Por exemplo, a linha

```
inline ($C9 \$E900);
```

insere três bytes no programa: $C9, $E9 e 00.

OVERLAY

Uma das extensões mais importantes do Turbo Pascal é o comando **overlay**. Este permite que sejam escritos, compilados e executados programas cujo tamanho excederia à capacidade de memória do computador. Isto é feito armazenando, em disco, subprogramas, chamados sobreposições (*overlays*), que são lidos e usados quando necessário.

Para criar uma sobreposição, o comando **overlay** é colocado na frente da função ou procedimento a ser usado. Quando o Turbo Pascal encontra um comando **overlay**, ele imediatamente compila a rotina que se segue num arquivo separado. O *Manual de Turbo Pascal* traz uma explicação detalhada sobre o uso deste comando.

shr E shl

shr significa *shift right* (deslocamento à direita) e **shl** significa *shift left* (deslocamento à esquerda). Estes operadores são usados só com variáveis do tipo **integer**. Eles causam um número especificado de deslocamentos, à direita ou à esquerda, nos bits da variável sobre a qual operam. Este tipo de operação é usado para escrever programas a nível de sistema e controladores de dispositivos. A forma geral destas operações é:

expressão inteira **shr** *número*
ou
expressão inteira **shl** *número*

onde *número* é um inteiro entre 1 e 15.

Vejamos um exemplo do funcionamento destes operadores. A seguir temos a representação binária de uma variável **integer** chamada **contador**, cujo valor atual é 8:

0000 1000

Depois da execução da linha de comando

```
contador : = contador shr 1;
```

contador terá valor 4, representado assim:

0000 0100

Todos os bits foram deslocados uma posição para a direita. Se o Turbo Pascal agora executar o comando

```
contador : = contador shr 3;
```

contador passará a valer 32, sendo escrito

0010 0000

Como você deve ter notado, um deslocamento à direita equivale a uma divisão por dois, enquanto um deslocamento à esquerda equivale a uma multiplicação por dois. Muitas vezes, deslocamentos são usados para dobrar ou reduzir um número à metade.

SEQÜÊNCIAS DE CARACTERES (STRINGS)

No Pascal padrão não há nenhum método trivial de manipulação de seqüências de caracteres. Entretanto, o Turbo Pascal possui um tipo especial de matriz de caracteres chamado **string**, que facilita muito o uso de seqüências de caracteres. Tais seqüências podem ter entre 0 e 255 caracteres de comprimento. Para declarar uma seqüência de caracteres de tamanho máximo 20, escreve-se simplesmente:

```
var
  sequencia_teste:string[20];
```

Uma variável do tipo **string** pode ter qualquer comprimento maior que zero e menor ou igual ao tamanho máximo especificado.

XOR

O operador **XOR**, ou "ou exclusivo", pode ser aplicado a operandos dos tipos **integer** e **Boolean.**

O uso de **XOR** com operandos inteiros causa uma operação lógica "ou exclusivo" entre cada bit dos dois números. Com operandos booleanos, é obtida uma resposta **TRUE/FALSE.**

Na operação **XOR** entre inteiros, cada bit do resultado é determinado pela seguinte tabela-verdade:

XOR	0	1
0	0	1
1	1	0

Cada bit do resultado será 1 *se, e somente se,* um bit for 1 e o outro 0 nos operandos, deste modo:

```
      1011 0010
XOR   0110 1001
      ---------
      1101 1011
```

Com operandos **Boolean**, o resultado é obtido através da tabela-verdade abaixo:

XOR	F	T
F	F	T
T	T	F

O resultado da operação **XOR** será **TRUE** se apenas um dos operandos for **TRUE**. Por exemplo, se **X** e **Y** forem variáveis do tipo **Boolean** e **X** for **TRUE** e **FALSE**, o resultado de **X XOR Y** será **TRUE**.

PROCEDIMENTOS PREDEFINIDOS

O Turbo Pascal traz um rico sortimento de procedimentos predefinidos, criados para simplificar a tarefa da programação. A maioria destes procedimentos se concentra em três áreas distintas: controle de tela, manipulação de seqüências de caracteres e acesso ao sistema operacional. Estes novos procedimentos são tratados em outra parte deste livro, mas aqui é dada uma rápida descrição de alguns deles.

MANIPULAÇÃO DE GRÁFICOS E TELA

Os procedimentos básicos de manipulação de tela, bem como seus efeitos, estão listados a seguir:

Procedimento	**Efeito**
CrtEol	Limpa a linha da posição do cursor até o final.
CrtExit	Libera o terminal através do envio de uma seqüência de caracteres.
CrtInit	Inicializa o terminal através do envio de uma seqüência de caracteres.
ClrScr	Limpa a tela.
DelLine	Apaga a linha da posição do cursor até o final.
GotoXY	Posiciona o cursor na posição da tela de coordenadas X, Y.
InsLine	Insere uma linha na posição do cursor.
LowVideo	Reduz a luminosidade dos caracteres enviados para a tela.
NormVideo	Retorna à luminosidade normal.

Os procedimentos anteriores são usados em vários programas deste livro.

Além destes procedimentos, o Turbo Pascal do IBM PC inclui rotinas de gráficos em cores, gráficos de tartaruga e rotinas de criação de janelas que podem ser usadas para dar um toque profissional aos seus programas.

MANIPULAÇÃO DE SEQÜÊNCIAS DE CARACTERES

Como já foi dito, uma das extensões mais importantes do Turbo Pascal é aquela que permite a criação de variáveis do tipo **string**. Os procedimentos abaixo listados foram adicionados ao Turbo Pascal como suporte a este novo tipo de dado:

Procedimento	Efeito
Delete	Apaga uma subseqüência, dada a posição inicial e o comprimento.
Insert	Insere uma seqüência em outra, dada a posição inicial da segunda.
Str	Converte uma variável **real** ou **integer** em uma seqüência de caracteres.
Val	Converte uma seqüência de caracteres em um valor **real** ou **integer**.

As seguintes funções também ajudam na manipulação de seqüências de caracteres:

Função	Efeito
Copy	Copia uma seqüência de caracteres inteira ou parcialmente.
Concat	Concatena duas seqüências.
Length	Dá o comprimento de uma seqüência.
Pos	Dá a posição inicial de uma subseqüência dentro de outra seqüência.

ACESSO AO SISTEMA OPERACIONAL

O Turbo Pascal possui um procedimento predefinido que permite aos programas acesso direto a rotinas do sistema operacional. Para os sistemas CP/M-80 e CP/M-86, os procedimentos **Bdos** e **Bios** chamam o sistema operacional. No IBM PC e em outras máquinas que utilizam o MS-DOS, o procedimento usado é o **MsDOS**. (*Nota do Revisor Técnico:* Há, também, o procedimento **intr**.)

Os três procedimentos funcionam do mesmo modo. Através deles, o número de uma função e os parâmetros necessários são passados ao sistema operacional. O uso das funções do sistema operacional é o assunto do Capítulo 5.

DIFERENÇAS ENTRE O TURBO PASCAL E O PASCAL PADRÃO

Além dos acréscimos e extensões já discutidos, o Turbo Pascal e o Pascal padrão diferem em alguns outros aspectos. Se você não tem muita experiência com o Turbo Pascal, é importante que esteja atento às seguintes diferenças entre este último e o Pascal padrão:

- O procedimento **new**, que aloca uma variável dinâmica para ser usada com ponteiros, não aceita especificação de registros variantes. Se necessário, isto pode ser contornado com o uso do procedimento **getmem**.

- Um **goto** não pode remeter para fora do bloco onde ele está. Isto quer dizer que tanto o **goto** quanto o rótulo indicador de destino devem pertencer ao mesmo bloco.

- Para usuários do CP/M-80: subprogramas recursivos não podem usar parâmetros do tipo **var**.

- O **get** e o **put** não foram incluídos no Turbo Pascal. Em vez disto, os comandos **read** e **write** foram expandidos para incluir entrada e saída de disco.

- O comando **packed** pode ser usado, mas não terá efeito algum. Além disto, os procedimentos **pack** e **unpack** também não foram incluídos.

- O procedimento **page** não foi incluído.

CAPÍTULO **2**

ORDENAÇÃO E BUSCA

É bem possível que não existam, em toda a ciência da computação, duas áreas tão importantes e tão extensamente estudadas quanto as que tratam da ordenação e da busca de informação. Rotinas de ordenação e busca são utilizadas por todos os programas de bancos de dados, bem como por compiladores, interpretadores e sistemas operacionais. Este capítulo trata dos processos básicos de ordenação e busca de informações. Em primeiro lugar, serão discutidos os métodos de ordenação, pois, em geral, a ordenação da informação torna mais fácil uma posterior busca da mesma.

ORDENAÇÃO

Ordenação (sorting) é o processo pelo qual um conjunto de dados similares é colocado numa ordem crescente ou decrescente, ou seja, uma lista ordenada crescente i composta por n elementos será da forma:

$$i_1 \leq i_2 \leq \ldots \leq i_n$$

Os algoritmos de ordenação se dividem em dois grandes grupos: a ordenação por matrizes, tanto na memória da máquina quanto em arquivos de disco, e a ordenação seqüencial, em arquivos de disco ou fita. Este capítulo se concentra na ordenação

matricial, por ser ela de maior importância para os usuários de microcomputadores. Entretanto, a ordenação seqüencial é também brevemente abordada.

A principal diferença entre os dois métodos de ordenação está na disponibilidade dos elementos. Na ordenação matricial, todos os elementos da matriz estão disponíveis ao mesmo tempo. Isto significa que qualquer elemento pode ser comparado ou permutado com qualquer outro a qualquer momento. Já num arquivo seqüencial, apenas um elemento estará disponível a cada momento. Por este motivo, as técnicas de ordenação usadas em cada um dos casos são muito diferentes.

Normalmente, quando se ordena uma informação, apenas uma parte desta informação é usada como *chave de ordenação*, com base na qual as comparações são feitas. Porém, se uma permutação é feita, toda a estrutura do dado permutado é transferida. Numa mala direta, por exemplo, o CEP pode ser usado como chave. Mas, se uma permutação for feita, o nome e o endereço acompanharão o CEP. Por motivos didáticos, todos os exemplos apresentados neste capítulo se referem a matrizes de caracteres; tais métodos, entretanto, poderão ser aplicados a quaisquer tipos de estrutura de dados.

CLASSES DE ALGORITMOS DE ORDENAÇÃO

Existem três processos distintos de ordenação de matrizes:

- por permutação;
- por seleção;
- por inserção.

Imagine, por exemplo, um baralho. Para ordená-lo por *permutação*, você espalharia as cartas sobre uma mesa, viradas para cima. Você, então, começaria a permutar as que estivessem fora de ordem, até que o baralho estivesse ordenado.

Para ordenar o baralho por *seleção*, você selecionaria, das cartas sobre a mesa, a de menor valor, separando-a das demais. Das cartas restantes, você novamente separaria a de menor valor, colocando-a sob a primeira. Quando não restasse nenhuma carta sobre a mesa, o baralho estaria ordenado (pois você selecionou sempre a carta de menor valor das que estavam na mesa).

Numa ordenação por *inserção*, você seguraria todo o baralho na mão, retirando uma carta por vez. Cada carta retirada você colocaria numa nova pilha, sobre a mesa, inserindo sempre cada carta em sua posição correta nesta pilha. O baralho estaria ordenado quando não restasse nenhuma carta em sua mão.

AVALIANDO OS ALGORITMOS DE ORDENAÇÃO

Para cada método de ordenação existem vários algoritmos, cada um com seus méritos próprios; a avaliação de um algoritmo de ordenação, entretanto, é feita com base nas respostas às seguintes perguntas:

- Quanto tempo o algoritmo gasta para ordenar informação num caso médio?
- Quanto tempo ele gasta em seu pior e em seu melhor caso?
- O algoritmo apresenta ou não um comportamento *natural*?
- Como ele se comporta frente a elementos com chaves de ordenação iguais?

O tempo gasto pelo algoritmo é de grande importância. A velocidade de ordenação de uma matriz é diretamente proporcional ao número de permutações e comparações realizadas, sendo as permutações mais lentas. Em alguns algoritmos o tempo de execução se relaciona de modo exponencial com o tamanho da lista, enquanto em outros esta relação é logarítmica.

Os tempos do melhor e do pior caso são importantes se tais situações são esperadas com freqüência. Um algoritmo de ordenação muitas vezes tem um bom caso médio e um péssimo pior caso, ou vice-versa.

Um algoritmo apresentará um *comportamento natural* se a quantidade de trabalho por ele realizada for diretamente proporcional à ordenação prévia da lista, isto é, se ele trabalhar menos numa lista já ordenada, aumentar a quantidade de trabalho na medida em que cresce o estado de desordem da lista, e realizar a maior quantidade de trabalho numa lista em ordem inversa (a quantidade de trabalho realizada por um algoritmo se mede pelo número de comparações e permutações por ele executadas).

Para entender a importância do tratamento dado pelo algoritmo a elementos de chaves iguais, imagine um banco de dados organizado por uma chave principal e por uma subchave, secundária. Por exemplo, uma mala direta em que o CEP seja a chave principal e o sobrenome a chave secundária. Quando um novo endereço é acrescentado e a lista reordenada, apenas as chaves secundárias referentes à chave principal daquele endereço devem ser rearranjadas.

Nas seções seguintes, algoritmos representativos de cada classe serão analisados e sua eficiência discutida. Todos os algoritmos usarão as declarações do tipo abaixo:

```
type
    Dado = char;
    Matriz_Dados = array[1...80] of Dado;
```

Assim, para mudar o tipo de dado basta mudar estas declarações. O tamanho da matriz é também arbitrário, podendo ser modificado sempre que necessário.

O BUBBLE SORT Um dos mais conhecidos algoritmos de ordenação é o *Bubble Sort*. Sua popularidade deriva de seu nome atraente e de sua simplicidade. Este, no entanto, é um dos piores algoritmos de ordenação jamais inventados.

O *Bubble Sort* utiliza o método de classificação por permutação. Ele realiza repetidas comparações e, quando necessário, permutações de elementos adjacentes. Seu nome vem da similaridade entre este método e o comportamento de bolhas em um tanque de água, onde cada bolha busca seu próprio nível. A seguir está listado o *Bubble Sort* em sua forma mais simples:

```
procedure Bubble(var item: Matriz_Dados; conta:integer);
var
   i,j: integer;
   x: Dado;
begin
   for i:=2 to conta do
   begin for j:=conta downto i do
      if item[j-1] > item[j] then
      begin
         x := item[j-1];
         item[j-1] := item[j];
         item[j] := x;
      end;
   end;
end;
```

Neste exemplo, **item** é a matriz de **Dado** a ser ordenada e **conta** é o número de elementos da matriz.

Um *Bubble Sort* é controlado por dois *loops*. Como a matriz possui um número **conta** de elementos, o *loop* externo faz com que ela seja varrida **conta**–1 vezes. Isto garante que, na pior das hipóteses, cada elemento esteja em sua posição correta ao fim do procedimento. O *loop* interno, por sua vez, realiza as comparações e permutações.

Esta versão do *Bubble Sort* coloca uma matriz de caracteres em ordem ascendente. O próximo programa lê uma seqüência de caracteres de um arquivo de disco chamado **TESTE.DAT** e a ordena. O mesmo programa pode ser usado com todas as outras rotinas deste capítulo, bastando para isto mudar o procedimento de ordenação.

```
program Ordenador;

{este programa lera, de um arquivo de disco chamado 'teste.dat', ate 80
caracteres, ordenando-os e mostrando o resultado na tela.}

type
   Dado = char;
   Matriz_Dados = array [1..80] of Dado;
var
   teste: Matriz_Dados;
   t,t2: integer;
   arq_teste: file of char;

procedure Bubble(var item: Matriz_Dados; conta:integer);
var
   i,j: integer;
   x: Dado;
begin
   for i:=2 to conta do
   begin for j:=conta downto i do
      if item[j-1] > item[j] then
      begin
         x := item[j-1];
         item[j-1] := item[j];
         item[j] := x;
      end;
   end;
end;

begin
    Assign(arq_teste, 'teste.dat');
    Reset(arq_teste);
    t:=1;

    while not Eof(arq_teste) do begin
       read(arq_teste, teste[t]);
       t:=t+1;
    end;

    t:=t-2; {retira o control-z no final do arquivo}
    Bubble(teste,t);  {ordena a matriz}

    for t2:=1 to t do write(teste[t2]);
    WriteLn;
end.
```

Para ilustrar melhor o modo de funcionamento do *Bubble Sort*, aqui estão as passagens necessárias para ordenar a seqüência **dcab:**

início	dcab
1)	adcb
2)	abdc
3)	abcd

Para se analisar uma rotina de ordenação é necessário determinar quantas

comparações e permutações serão feitas por tal rotina em seu melhor caso, em seu pior caso e em seu caso médio. No *Bubble Sort* o número de comparações será ~~sempre o~~ número especificado em **conta**, esteja a lista inicialmente ordenada ou não. Isto é, o *Bubble Sort* realizará sempre $(n^2-n)/2$ comparações, onde n é o número de elementos a serem ordenados. (Esta fórmula é o produto da multiplicação do número de execuções do *loop* externo ($n-1$) pelo número de execuções do *loop* interno ($n/2$).)

O número de permutações será 0 no melhor caso (uma lista já ordenada), $3/4(n^2-n)$ no caso médio e $3/2(n^2-n)$ no pior caso. Está além do objetivo deste livro demonstrar tais fórmulas, mas através delas você pode notar que, quanto mais desordenada estiver a lista, mais o número de elementos fora de ordem se aproxima do número de comparações a serem feitas (além disto, o *Bubble Sort* fará três permutações para cada elemento fora de ordem). O *Bubble Sort* se enquadra no grupo dos chamados *algoritmos exponenciais*, pois seu tempo de execução é proporcional ao quadrado do número de elementos da lista. Como o tempo de execução se relaciona diretamente ao número de comparações e permutações a serem realizadas, um *Bubble Sort* é péssimo para lidar com grandes listas.

Mesmo ignorando o tempo necessário para permutar elementos fora de ordem, e se cada comparação durar, por exemplo, 0,001 segundo, a ordenação de 10 elementos requererá aproximadamente 0,05 segundo, a ordenação de 100 elementos levará 5 segundos e a ordenação de 1000 elementos será feita em 500 segundos. O tempo necessário para ordenar 100.000 elementos (o tamanho de uma pequena lista telefônica) será de aproximadamente 5.000.000 de segundos, ou 1.400 horas (mais de 2 meses de ordenação contínua!). O gráfico da Figura 2-1 mostra a relação entre o número de elementos da matriz e o tempo de execução do *Bubble Sort*.

Algumas mudanças podem ser feitas no *Bubble Sort* para acelerar seu tempo de execução (e melhorar um pouco seu conceito entre os programadores). Por exemplo, o *Bubble Sort*, listado anteriormente, possui uma peculiaridade: um elemento cuja posição correta for no início da matriz (como o **a** do exemplo **dcba**) irá para sua posição em um movimento, enquanto um elemento cuja posição correta for no final da matriz (como o **d** no mesmo exemplo) se deslocará lentamente até sua posição. Se, em vez de ler a matriz sempre no mesmo sentido, o programa inverter o sentido a cada passagem, elementos cuja posição inicial for muito distante da posição correta irão mais rapidamente para seu lugar.

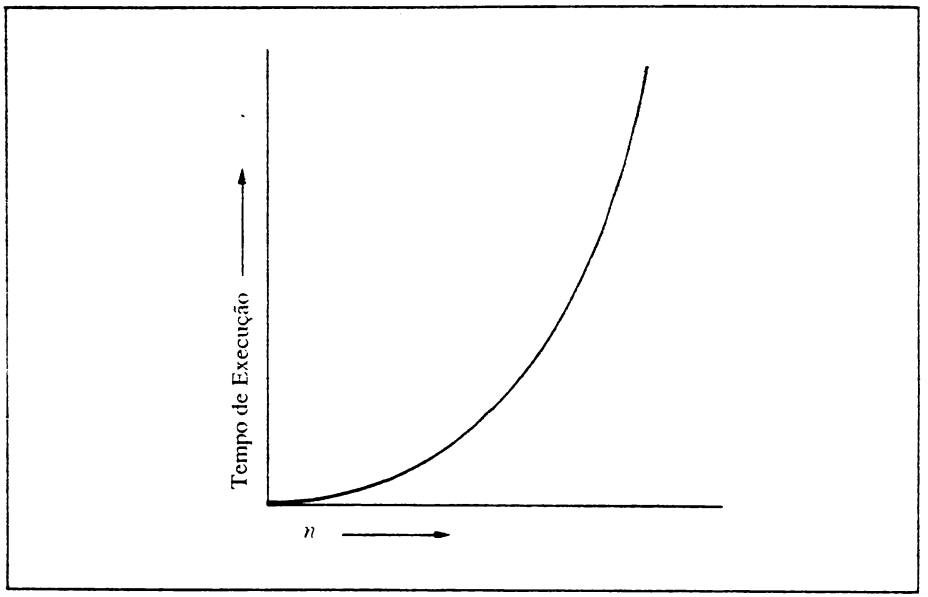


Figura 2-1. Tempo de execução *versus* número de registros numa ordenação de tempo quadrático (n^2).

Esta versão do *Bubble Sort* é chamada *Shaker Sort* por causa de seu movimento vibratório (de *to shake* = vibrar) ao longo da matriz:

```
procedure Shaker(var item: Matriz_Dados; conta:integer);

{esta e' uma versao melhorada do BubbleSort}

var
   j,k,l,r: integer;
   x: Dado;
begin
   l:=2; r:=conta; k:=conta;
   repeat
      for j:= r downto l do
         if item[j-1] > item[j] then
         begin
            x:=item[j-1];
            item[j-1]:=item[j];
            item[j]:=x;
            k:=j;
        end;

        l:=k+1;

        for j:=l to r do
           if item[j-1] > item[j] then
           begin
```

```
                x:=item[j-1];
                item[j-1]:=item[j];
                item[j]:=x;
                k:=j;
            end;

        r:=k-1;

   until l>r
end; {ShakerSort}
```

Apesar de representar um avanço em relação ao *Bubble Sort* original, o *Shaker Sort* também não é recomendável, pois seu tempo de execução ainda é exponencial. O número de comparações não mudou e o número de permutações foi apenas levemente reduzido.

ORDENAÇÃO POR SELEÇÃO Numa *ordenação por seleção*, o elemento de valor mais baixo da matriz é escolhido e permutado com o primeiro elemento. Dos elementos restantes, o de valor mais baixo é escolhido e permutado com o segundo elemento da matriz, e assim por diante, até que a matriz esteja ordenada. Se, por exemplo, a matriz **bdac** for ordenada por seleção as passagens serão:

início	badc
1)	~~adbc~~ abdc
2)	abdc
3)	abcd

Uma forma simples de ordenação por seleção é mostrada a seguir:

```
procedure Selecao (var item: Matriz_Dados; conta:integer);
var
   i,j,k: integer;
   x: Dado;

begin
   for i:=1 to conta-1 do
   begin
      k:=i;
      x:=item[i];
      for j:=i+1 to conta do {acha o menor elemento}
         if item[j] < x then
         begin
            k:=j;
            x:=item[j];
         end;
         item[k]:=item[i];
         item[i]:=x;
      end;
end; {ordenacao por selecao}
```

Como no *Bubble Sort*, o *loop* externo é executado $n-1$ vezes e o *loop* interno é executado $n/2$ vezes, o que significa um total de $(n^2-n)/2$ comparações, tornando também este método impróprio para listas grandes. O número de permutações no melhor caso é $3(n-1)$ e no pior $n^2/4+3(n-1)$ (aproximando-se do número de comparações).

O número de permutações no caso médio é igual a $n(\ln n+y)$, onde y é a constante de Euler (aproximadamente 0,577216). Isto quer dizer que o número de permutações num caso médio de ordenação por seleção é bem menor que no *Bubble Sort* (apesar do número de comparações ser idêntico).

ORDENAÇÃO POR INSERÇÃO Este é o último dos algoritmos simples de ordenação que veremos. O algoritmo de ordenação por inserção começa ordenando os dois primeiros elementos da matriz. O terceiro elemento é, então, colocado em sua posição correta em relação aos dois primeiros. O quarto elemento é inserido na lista de três elementos, e o processo continua até que todos os elementos estejam em sua posição correta. Por exemplo, na matriz **dcab** a ordenação por inserção funcionaria assim:

início	dcab
1)	cdab
2)	acdb
3)	abcd

O programa seguinte é uma versão da ordenação por inserção:

```
procedure Insercao(var item: Matriz_Dados; conta: integer);
var
   i,j: integer;
   x: Dado;
begin
   for i:=2 to conta do
   begin
      x:= item[i];
      j:= i-1;
      while (x < item[j]) and (j > 0) do
      begin
         item[j+1]:=item[j];
         j:=j-1;
      end;
      item[j+1]:=x;
   end;
end; {ordenacao por insercao}
```

Ao contrário do que ocorre no *Bubble Sort* e na ordenação por seleção, aqui, o número de comparações efetuadas depende do estado inicial da lista. Se a lista estiver na ordem correta, serão feitas $n-1$ comparações. Se a lista estiver na ordem inversa, o número

de comparações será de $(n^2 + n)/2-1$, sendo a média dos dois números igual a $(n^2+n-2)/4$.

Para as permutações, os números são os seguintes:

Melhor caso: $2(n-1)$

Caso médio: $(n^2+9n-10)/2$

Pior caso: $(n^2+3n-4)/2$

Ou seja, o número do pior caso é tão ruim quanto o do *Bubble Sort* ou o da ordenação por seleção, e o número do caso médio é pouca coisa melhor. Mas a ordenação por inserção apresenta duas vantagens. Primeiro, o algoritmo tem um comportamento natural, trabalhando menos em listas já ordenadas e mais em listas invertidas. Isto torna tal método útil para ser usado em listas que estejam quase em ordem. Além disto, a ordem de chaves iguais não muda, isto é, se uma lista com duas chaves for ordenada pelo método de inserção, ao final ela estará ordenada pelas duas chaves.

Apesar do número de comparações tornar este algoritmo útil para certos conjuntos de dados, o fato da matriz ser constantemente deslocada faz com que o número de movimentos seja significativo (ainda assim, é importante lembrar que a ordenação por inserção se comporta naturalmente).

ORDENAÇÕES APERFEIÇOADAS

Cada um dos algoritmos descritos incorre no erro fatal de ter um tempo de execução exponencial. Para grandes quantidades de dados, estes procedimentos seriam vagarosos – de um certo momento em diante vagarosos demais para serem usados. Qualquer programador conhece uma ou mais variações da história de terror chamada "a ordenação que durou três dias". Infelizmente, estas histórias são, muitas vezes, verdadeiras.

Quando uma ordenação demora muito tempo, o erro pode estar no algoritmo usado. A primeira resposta quase sempre será "vamos escrever isto em *assembly*". Mas, se o algoritmo for ruim, aumentar a velocidade não tornará a ordenação muito mais rápida (não importando qual linguagem seja usada). Lembre-se: se uma rotina tem um tempo de execução exponencial, o aumento da velocidade da linguagem ou do computador usados causará apenas uma pequena melhoria (o gráfico da Figura 2-1 será levemente deslocado para a direita, mas a curva ainda será a mesma). Tenha em mente que se um programa não

é rápido o bastante escrito em Turbo Pascal, ele não será rápido o bastante escrito em *assembly*. A solução é usar um programa melhor.

Nesta seção serão discutidos dois excelentes algoritmos de ordenação, o *Shell Sort* e o *QuickSort* (este último é geralmente considerado a melhor rotina de ordenação hoje disponível). Estes programas realizam uma ordenação, literalmente, num piscar de olhos.

O SHELL SORT O *Shell Sort* foi batizado com o nome de seu inventor, D.L. Shell. Entretanto, o nome pode também ter sido inspirado no próprio método de operação do algoritmo, que faz lembrar conchas (*shells*) apoiadas umas sobre as outras.

O método usado por este algoritmo é derivado da ordenação por inserção, e se baseia na redução de intervalos. A Figura 2-2 mostra o diagrama de um *Shell Sort* ordenando a matriz **fdacbe**. Primeiro são ordenados todos os elementos a três posições de distância. Em seguida, os elementos a duas posições de distância são ordenados. Finalmente, são ordenados os elementos adjacentes.

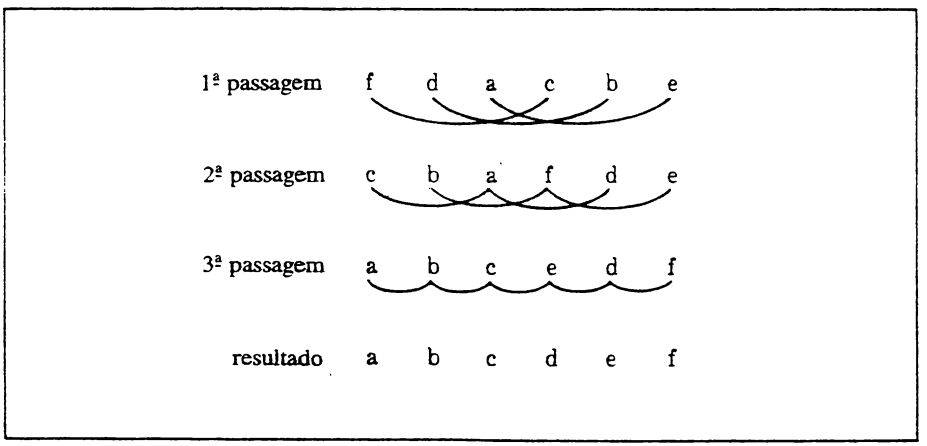


Figura 2-2 O Shell Sort.

```
procedure Shell(var item : Matriz_Dados; conta: integer);
const
   t = 5;
var
   i,j,k,s,w: integer;
   h: array[1..t] of integer;
   x: Dado;
begin
   h[1]:=9; h[2]:=5; h[3]:=3; h[4]:=3; h[5]:=1;
```

```
   for m:=1 to t do
   begin
      k:=h[m];
      s:=-k;
      for i:=k+1 to conta do
      begin
         x:=item[i];
         j:=i-k;
         if s = 0 then
         begin
            s:=-k;
            s:=s+1;
            item[s]:=x;
         end;
         while (x < item[j]) and (j > 0) and (j <= conta) do
         begin
            item[j+k]:=item[j];
            j:=j-k;
         end;
         item[j+k]:=x;
      end;
   end;
end; {ordenacao de Shell}
```

Apesar disto não ser óbvio, este método ordena a matriz, e o faz de maneira eficiente. O algoritmo é eficiente porque cada passagem da ordenação ou envolve poucos elementos ou envolve elementos que já estão numa ordem razoável. Assim, cada passagem aumenta a ordenação da lista.

A seqüência de intervalos usados pode ser mudada, desde que o último intervalo seja 1. Por exemplo, a seqüência 9, 5, 3, 1 funciona bem (é esta a seqüência usada no *Shell Sort* mostrado acima). Evite, porém, usar seqüências que envolvam múltiplos de 2, pois tais seqüências, devido a razões matemáticas complexas, reduzem a eficiência do algoritmo (mas a ordenação ocorrerá mesmo que você as use).

O *loop* interno **while** possui duas condições de teste. O **x<item[j]** é uma comparação necessária ao processo de ordenação. Já os testes **j>0** e **j<=conta** evitam que a ordenação ultrapasse as fronteiras da matriz. Estes testes de limites atrapalham consideravelmente a performance do *Shell Sort*. Por este motivo, versões um pouco diferentes do *Shell Sort* utilizam elementos especiais da matriz, chamados sentinelas. Os sentinelas não fazem parte da informação a ser ordenada, possuindo valores especiais que indicam o menor e o maior elementos possíveis da matriz. Deste modo, a verificação de limites se torna desnecessária. No entanto, o uso de sentinelas exige um conhecimento específico dos dados que estão sendo ordenados, limitando a generalidade do procedimento de ordenação.

O tempo de execução do *Shell Sort* é proporcional a $n^{1.2}$ para a ordenação de n elementos. Isto representa um avanço significativo em relação aos métodos vistos nas

seções anteriores, todos eles proporcionais a n^2. A Figura 2-3 mostra as curvas n^2 e $n^{1.2}$, lado a lado. Mas antes de decidir usar o *Shell Sort*, veja o *QuickSort*, que é ainda melhor.

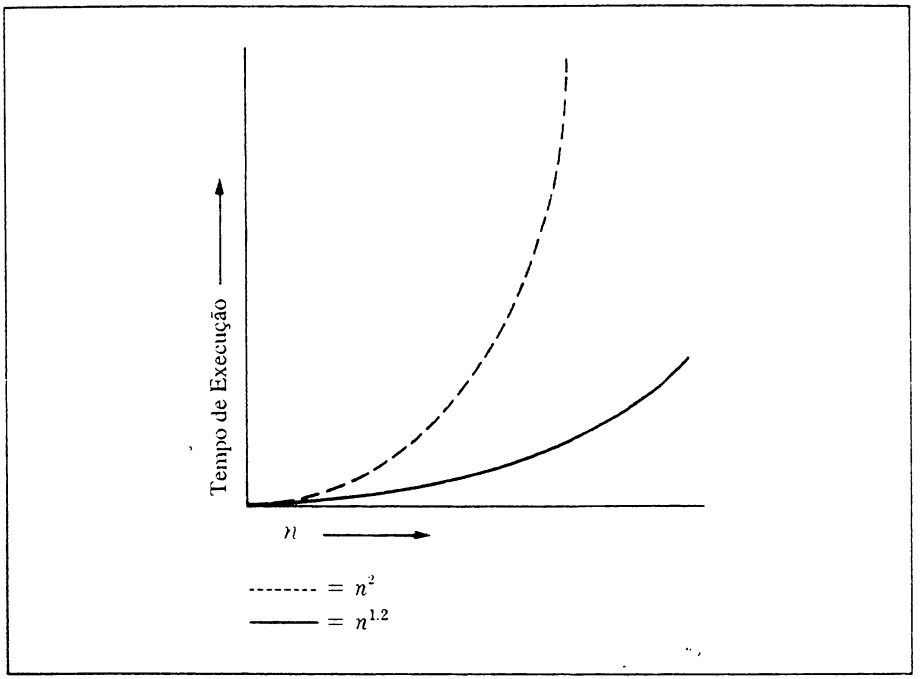


Figura 2-3 As curvas de n^2 e $n^{1.2}$.

O QUICKSORT O *QuickSort* (ordenador rápido), criado e batizado por C.A.R. Hoare, é tido como o melhor algoritmo de ordenação atualmente disponível. Ele se baseia no método de ordenação por permutação, o que é uma surpresa, considerando o péssimo desempenho da versão mais simples de ordenação por permutação, o *Bubble Sort*.

O *QuickSort* se utiliza de partições para ordenar a matriz. O algoritmo seleciona um valor, chamado *comparando*. Ele, então, divide a matriz em duas partes, com todos os elementos maiores ou iguais ao valor de comparação de um lado e todos os elementos menores que o valor de comparação de outro. Este processo se repete em cada uma das duas partes resultantes, até que a matriz esteja ordenada. Por exemplo, dada a matriz **fedacb**, e usando **d** como primeiro valor de comparação, a primeira passagem do *QuickSort* rearranjaria a matriz do seguinte modo:

início	fedacb
1)	bcadef
2)	abcdef

O processo se repete sobre cada uma das partes restantes, **bca** e **def.**

O método usado é recursivo por natureza. Na verdade, as implementações mais elegantes do *QuickSort* são algoritmos recursivos.

A seleção do elemento de comparação pode ser feita de dois modos. O elemento pode ser escolhido ao acaso ou pode-se selecionar o elemento médio de um pequeno subconjunto da matriz. Para otimizar a ordenação, o melhor seria selecionar o elemento de valor médio entre todos os elementos da matriz. Isto, entretanto, não é possível na maioria dos casos (mas mesmo no pior caso, aquele no qual o valor escolhido está numa das extremidades, o *QuickSort* trabalhará bem).

A versão do *QuickSort* mostrada aqui seleciona sempre o elemento central da matriz. Apesar desta não ser sempre uma boa escolha, ela é rápida e simples, e a ordenação funcionará corretamente.

```
procedure QuickSort (var item: Matriz_Dados; conta: integer);
   procedure qs (l,r: integer; var it: Matriz_Dados);
      var
      i,j: integer;
      x,y: Dado;
      begin
         i:=l; j:=r;
         x:=it[(l+r) div 2];
         repeat
            while it[i] < x do i:=i+1;
            while x < it[j] do j:=j-1;
            if i <= j then
            begin
               y:=it[i];
               it[i]:=it[j];
               it[j]:=y;
               i:=i+1; j:=j-1;
            end;
         until i > j;
         if l < j then qs(l,j,it);
         if l < r then qs(i,r,it);
      end;
begin
     qs(1,conta,item);
end; {QuickSort}
```

Aqui o procedimento **QuickSort** chama o procedimento de ordenação **qs**. Isto permite que a ligação entre **item** e **conta** seja mantida.

Pode-se assumir que o número de comparações realizadas pelo *QuickSort* é dado pela fórmula $n \log n$ e que o número de permutações seja de aproximadamente $n/6 \log n$.

Tais valores são significativamente melhores que os de qualquer dos outros métodos de ordenação vistos até aqui.

A equação $N = a^x$ pode ser escrita $x = \log_a N$

Assim, se existirem 100 elementos para serem ordenados, o *QuickSort* fará, em média, 200 comparações (pois log 100 é 2). Se comparado às 990 comparações requeridas em média pelo *Bubble Sort*, tal número é muito bom. Entretanto existe um detalhe que pode tornar o *QuickSort* um *slowsort* (ordenador lento). Se o elemento de comparação escolhido para cada partição for sempre o maior valor daquela parte da matriz, o tempo de execução do *QuickSort* será exponencial. Felizmente, este é um caso muito raro.

Você deve tomar cuidado ao escolher o método de seleção do elemento de comparação. Na maior parte das vezes, a própria natureza da informação a ser ordenada determinará o método a ser usado. Em grandes malas diretas, nas quais a ordenação em geral usa o CEP como chave, a seleção é simples. Como os CEPs têm uma distribuição ampla, um procedimento algébrico trivial pode produzir o elemento adequado. Em alguns bancos de dados; porém, as chaves de ordenação podem estar tão próximas que a seleção aleatória poderá ser o melhor método disponível. Um método muito comum e eficiente é tomar três elementos quaisquer de uma partição, escolhendo, entre estes, o elemento médio.

ORDENANDO OUTROS TIPOS DE INFORMAÇÃO

Até aqui só tratamos da ordenação de matrizes de caracteres. Como já foi mencionado, no início deste capítulo, matrizes contendo qualquer tipo de dado podem ser ordenadas, bastando para isto mudar a declaração de tipo de **Matriz_Dados.** Normalmente, entretanto, os dados a serem ordenados são complexos, como seqüências ou grupos de informação, registros (**records**), por exemplo. Ao adaptar os algoritmos de ordenação a outras estruturas de dados, podem ser necessárias mudanças na rotina de comparação, na de permutação, ou em ambas. O algoritmo básico, entretanto, permanecerá sempre o mesmo.

Como o *QuickSort* é a melhor entre as rotinas descritas neste capítulo, ele será usado nos exemplos subseqüentes. No entanto, as mesmas técnicas se aplicam a qualquer outro dos algoritmos vistos.

A ORDENAÇÃO DE SEQÜÊNCIAS DE CARACTERES O modo mais simples de ordenar seqüências de caracteres é criar uma matriz com elas, usando para isto o tipo

string do Turbo Pascal. Isto permite fácil indexação e permutação, além de manter inalterado o algoritmo básico *QuickSort*. A versão a seguir ordena seqüências de caracteres em ordem alfabética.

```
type
   Dado = string[80];
   Matriz_Dados = array[1..80] of Dado;

Procedure QsString(var item: Matriz_Dados; conta: integer);
   procedure QS(l,r: integer; var it: Matriz_Dados);
      var
      i,j: integer;
      x,y: Dado;
      begin
         i:=l;
         x:=it[(l+r) div 2];
         repeat
            while it[i] < x do i:=i+1;
            while x < it[j] do j:=j-1;
            if i <= j then
            begin
               y:=it[i];
               it[i]:=it[j];
               it[j]:=y;
               i:=i+1; j:=j-1;
            end;
         until i > j;
         if l < j then qs(l,j,it);
         if l < r then qs(i,r,it);
      end;
begin
     QS(1,conta,item);
end; {QsString}
```

Note que apenas a declaração de tipo de **Matriz_Dados** foi mudada para transformar a ordenação de caracteres em ordenação de seqüências de caracteres. Isto é possível devido à excelente integração dos tipos de **string** do Pascal, feita pela Borland. No Pascal padrão, ordenar seqüências exigiria uma rotina muito maior.

A comparação de seqüências toma mais tempo que a comparação de caracteres, pois mais elementos devem ser testados em cada caso.

A ORDENAÇÃO DE REGISTROS A maior parte dos programas de aplicação que requerem uma ordenação precisa ter um grupo de dados previamente ordenados. Uma mala direta é um exemplo excelente, pois nome, rua, cidade, Estado e CEP estão unidos. Quando este bloco unitário de informação é ordenado, qualquer que seja a chave usada, todo o bloco será movido se houver uma permutação. Para entender este processo, crie você um registro (**record**) para conter esta informação. Usando o endereço como exemplo, o registro adequado para manter a informação, no nosso caso, seria o seguinte:

```
type
   endereco = record
              nome: string[30];
              rua: string[40];
              cidade: string[20];
              estado: string[2];
              cep: string[5];
   end;
```

Após definir **endereco,** a declaração de tipo de Dado deve ser mudada:

```
Dado = endereco;
```

Além disto, a rotina de comparação do *QuickSort* deve ser mudada para adequá-la ao campo a ser ordenado. Na versão seguinte, o campo usado é o **nome.** Isto quer dizer que a mala direta será ordenada em ordem alfabética pelo nome:

```
procedure QsRegistro (var item: Matriz_Dados; conta: integer);
   procedure qs (l,r: integer; var it: Matriz_Dados);
      var
      i,j: integer;
      x,y: Dado;
      begin
         i:=l; j:=r;
         x:=it[(l+r) div 2];
         repeat
            while it[i].nome < x.nome do i:=i+1;
            while x.nome < it[j].nome do j:=j-1;
            if i <= j then
            begin
               y:=it[i];
               it[i]:=it[j];
               it[j]:=y;
               i:=i+1; j:=j-1;
            end;
         until i > j;
         if l < j then qs(l,j,it);
         if l < r then qs(i,r,it);
      end;
begin
     qs(1,conta,item);
end; {QsRegistro}
```

ORDENANDO ARQUIVOS DE DISCO

Existem dois tipos de arquivos de disco, os *seqüenciais* e os de *acesso aleatório*. Se o arquivo for pequeno o bastante para ser carregado na memória do computador, isto tornará a ordenação mais rápida. Entretanto, a maior parte dos arquivos é grande demais para ser ordenada diretamente na memória, requerendo técnicas especiais.

A ORDENAÇÃO DE ARQUIVOS DE ACESSO ALEATÓRIO Usados pela maioria dos programas de aplicação para microcomputadores, os arquivos de acesso aleatório apresentam duas grandes vantagens sobre os arquivos seqüenciais. Além da informação neles contida poder ser modificada sem que toda a lista seja novamente copiada, eles podem ser tratados como uma grande matriz gravada no disco, simplificando enormemente o processo de ordenação.

Com arquivos de acesso aleatório, o *QuickSort* básico, com apenas algumas modificações, pode ser usado para buscar registros diferentes no disco, como se você estivesse indexando uma matriz. Ao contrário do que ocorre em arquivos seqüenciais, aqui, o disco não precisa ter espaço para os arquivos já ordenados e os ainda não-ordenados, durante o processo de ordenação.

Cada situação de ordenação é particular, dependendo do tipo exato de estrutura a ser ordenado e da chave usada. Porém, o conceito geral de ordenação de arquivos de acesso aleatório pode ser compreendido através do programa a seguir, que ordena o registro de mala direta **endereco,** definido no exemplo anterior. (O programa assume que o número de elementos é 80. Numa aplicação real, o contador de registros deve ser mantido de modo dinâmico.)

```
program OrdMalaDireta;

type
    endereco = record
            nome : string[30];
            rua : string[40];
            cidade : string[20];
            estado : string[2];
            cep : string[5];

    end;
    str80 = string[80];
    Dado = endereco;
    Matriz_Dados = array[1..80] of Dado;
    arq_end = file of endereco;

var
   teste: Dado;
   t,t2: integer;
   arq_teste: file of endereco;

function Localiza (var fp: arq_end; i: integer): str80;
var
   t: endereco;
begin
   i:= i-1;
   Seek(fp,i);
   Read(fp,t);
   Localiza:= t.nome;
end; {Localiza}
```

```
procedure QsRand(var fp:arq_end; conta: integer);
    procedure QS(l,r: integer);
       var
       i,j,s: integer;
       x,y,z: Dado;
       begin
          i:=l; j:=r;
          s:=(l+r) div 2;
          Seek(fp,s-1);
          Read(fp,x);
          repeat
             while Localiza(fp,i) < x.nome do i:=i+1;
             while x.nome < Localiza(fp,j) do j:=j-1;
             if i <= j then
             begin
                Seek(fp,i-1); Read(fp,y);
                Seek(fp,j-1); Read(fp,z);
                Seek(fp,j-1); Write(fp,y);
                Seek(fp,i-1); Write(fp,z);
                i:= i+1; j:=j-1;
             end;
          until i > j;
          if l < j then QS(l,j);
          if l < r then QS(i,r);
       end;
begin
     QS(1,conta);
end; {QsRand}

begin
    Assign(arq_teste, 'teste.dat');
    Reset(arq_teste);
    t:=1;
    while not Eof(arq_teste) do begin
       Read(arq_teste,teste);
       t:=t+1;
    end;
    t:=t-1;
    QsRand(arq_teste,t);
end.
```

A função **Localiza** foi incluída para manter inalterada a parte essencial do *QuickSort*. **Localiza** traz, de um registro no disco, a seqüência **nome**. Como arquivos de disco são numerados a partir de 0, é necessário subtrair 1 constantemente dos argumentos de **Localiza** e **Seek**.

A ORDENAÇÃO DE ARQUIVOS SEQÜENCIAIS Ao contrário dos arquivos de acesso aleatório, arquivos seqüenciais não usam registro de tamanho fixo, sendo comum armazená-los de tal modo que o acesso aleatório seja difícil. Arquivos seqüenciais são utilizados por aplicações que necessitem de registros de tamanho variável ou métodos de

armazenamento seqüenciais por natureza. Por exemplo, a maioria dos arquivos de texto é seqüencial.

Não se pode tratar um arquivo seqüencial como uma matriz, pois não há como ter acesso a um elemento qualquer de tal arquivo. Não há como conseguir acesso rápido a um registro qualquer de um arquivo seqüencial gravado, por exemplo, em fita. Por esta razão, seria difícil aplicar a arquivos seqüenciais os algoritmos de ordenação de matrizes apresentados até aqui.

Existem dois meios de ordenar um arquivo seqüencial. O primeiro consiste em carregar o arquivo na memória do computador e então aplicar qualquer dos algoritmos de ordenação já vistos. Este é um método rápido, mas o tamanho do arquivo a ser ordenado fica limitado pelo tamanho da memória da máquina.

O segundo método é chamado *Merge Sort*. O *Merge Sort* divide o arquivo a ser ordenado em dois arquivos do mesmo tamanho. O programa, então, lê um elemento de cada arquivo, ordena este par, e grava os dois elementos num terceiro arquivo. Quando, neste terceiro arquivo, estiverem gravados todos os elementos, ele é dividido em dois arquivos do mesmo tamanho, e o processo se repete até que o arquivo esteja ordenado. Este *Merge Sort* requer, portanto, que três arquivos estejam ativos ao mesmo tempo.

Para entender como o Merge Sort trabalha, observe a seguinte seqüência:

1 4 3 8 6 7 2 5

O Merge Sort divide a seqüência a ser produzida

1 4 3 8
6 7 2 5

ordena as duas partes

1 6 - 4 7 - 2 3 - 5 8

divide novamente

1 6 - 4 7
2 3 - 5 8

O merge seguinte produz

1 2 3 6 - 4 5 7 8

a divisão final é

1 2 3 6
4 5 7 8

com o resultado

1 2 3 4 5 6 7 8

No *Merge Sort*, cada arquivo é lido $\log_2 n$, onde n é o número de elementos a serem ordenados.

A seguir, é dada uma versão simples do *Merge Sort*. Ela assume que o arquivo inicial tem o dobro de seu tamanho real, de modo a necessitar de apenas um arquivo ativo. O método é o mesmo. Neste exemplo, **tipo_arq** é definido como um arquivo de tipo **Matriz_Dados.**

```
Function Localiza(var arq: tipo_arq; i:integer) : Dado;
var
   t: Dado;
begin
   Seek(arq,i-1);
   Read(arq,t);
   Localiza:=t;
end; {Localiza}

procedure MergeSort(var arq: tipo_arq; conta: integer);
var
   i,j,k,l,t,h,m,p,q,r: integer;
   d1,d2: Dado;
   up: boolean;

begin
   up:=TRUE;
   p:=1;
   repeat
      h:=1; m:=conta;
      if up then
      begin
         i:=1; j:=conta; k:=conta+1; l:=2*conta;
      end else
      begin
         k:=1; l:=conta; i:=conta+1; j:=2*conta;
      end;
```

```
      repeat
         if m >= p then q:=p else q:=m;
         m:=m-q;
         if m >= p then r:=p else r:=m;
         m:=m-r;
         while (q <> 0) and (r <> 0) do
         begin
            if Localiza(arq,i) < Localiza(arq,j) then
               begin
                  Seek(arq,i-1); Read(arq,d2);
                  Seek(arq,k-1); Write(arq,d2);
                  k:=k+h; i:=i+1; q:=q-1;
               end else
               begin
                  Seek(arq,j-1); Read(arq,d2);
                  Seek(arq,k-1); Write(arq,d2);
                  k:=k+h; j:=j-1; r:=r-1;
               end;
            end;
            while r <> 0 do
            begin
               Seek(arq,j-1); Read(arq,d2);
               Seek(arq,k-1); Write(arq,d2);
               k:=k+h; j:=j-1; r:=r-1;
            end;
            while q <> 0 do
            begin
               Seek(arq,i-1); Read(arq,d2);
               Seek(arq,k-1); Write(arq,d2);
               k:=k+h; i:=i+1; q:=q-1;
            end;
            h:=-1; t:=k;
            k:=l;
            l:=t;
         until m=0;
         up:=not up;
         p:=p*2;
      until p>conta;
      if not up then
         for i:=1 to conta do
         begin
            Seek(arq,i-1+conta); Read(arq,d2);
            Seek(arq,i-1); Write(arq,d2);
         end;
end;
```

BUSCA DE INFORMAÇÃO

Bancos de dados existem para que, de tempos em tempos, o usuário possa localizar e utilizar uma certa informação, desde que a chave seja conhecida. Só existem dois métodos

de busca de informação, um para arquivos ou matrizes já ordenados e outro para arquivos ou matrizes não-ordenados.

MÉTODOS DE BUSCA

Achar um dado elemento numa matriz não-ordenada requer uma *busca seqüencial*, começando do primeiro elemento e parando quando o elemento procurado for encontrado ou quando a matriz terminar. Com dados previamente ordenados, uma *busca binária* pode ser empregada, resultando em grande aumento na velocidade do processo.

A BUSCA SEQÜENCIAL A busca seqüencial é fácil de ser programada. A seguinte função percorre uma matriz de tamanho conhecido até que um elemento com a chave procurada seja encontrado.

```
function Busca_Seq(item: Matriz_Dados; conta: integer;
chave: Dado) : integer;
var
   t: integer;
begin
   t:=1;
   while(chave <> item[t]) and (t <= conta) t:=t+1;
   if t > conta then Busca_Seq:=0;
   else Busca_Seq:=t;
end; {Busca_Seq}
```

Esta função dá como resultado o número do elemento procurado, ou 0, se tal elemento não existir.

Uma busca seqüencial testará, em média, $n/2$ elementos. No melhor dos casos, apenas 1 elemento será testado, e no pior, n. Se a informação estiver gravada em disco, o tempo necessário poderá ser longo, mas este é o único método de busca disponível para dados não-ordenados.

A BUSCA BINÁRIA Se os dados entre os quais está aquele elemento a ser encontrado estiverem ordenados, então um método melhor, chamado *busca binária*, poderá ser usado. Este método é uma aplicação da máxima "divide e conquista". Ele primeiro testa o elemento central; se a chave deste elemento for maior que a chave procurada, ele então testa o elemento central da primeira metade. Caso contrário, ele testa o elemento central da segunda metade. O processo se repete até que o elemento procurado seja encontrado, ou até que não existam mais elementos para serem testados.

Por exemplo, para encontrar o número 4 na matriz 123456789, a busca binária testará primeiro o elemento central, isto é, o 5. Como este elemento é maior que 4, a busca continuará na primeira metade,

12345

Aqui, o elemento central é 3. Como este é menor que 4, a busca continua com

45

Desta vez, o elemento procurado é encontrado.

Na busca binária, o número de comparações no pior caso é $\log_2 n$. Nos casos médios, o número é um pouco melhor. No melhor caso, o número, naturalmente, é 1.

O programa a seguir realiza uma busca binária em uma matriz de caracteres. Ele pode ser usado com qualquer outra estrutura de dados, bastando, para isto, modificar a rotina de comparação, e a definição de tipo em **Dado:**

```
function Busca_Bin(item: Matriz_Dados; conta: integer;
chave: Dado) : integer;

var
   menor, maior, medio: integer;
   encontrou: boolean;
begin
   menor:=1; maior:=conta;
   encontrou:=conta;
   while (menor <= maior) and (not encontrou) do
   begin
      medio:=(menor + maior) div 2;
      if chave < item[medio] then maior:=medio-1
      else if chave > item[medio] then menor:=medio+1
      else encontrou:=true;      {encontrou}
   end;
   if encontrou then Busca_Bin:=medio
   else Busca_Bin:=0;  {nao encontrou}
end; {Busca_Bin}
```

O capítulo seguinte aborda os diferentes métodos de armazenamento e recuperação de dados, os quais muitas vezes tornam a ordenação e a busca tarefas muito mais simples.

CAPÍTULO 3

FILAS, PILHAS, LISTAS ENCADEADAS E ÁRVORES

Programas consistem em *algoritmos* e *estruturas de dados*. Um bom programa é uma mistura de ambos. A escolha e implementação de uma estrutura de dados é tão importante como as rotinas que os manipulam. A maneira como a informação é organizada e acessada é normalmente determinada pela natureza do problema. Portanto, como programador, você deve ter na sua "cartola" os métodos certos de armazenamento e recuperação para cada situação.

A representação de dados no computador é construída "de baixo para cima", começando com os tipos de dados básicos como **char, integer** e **real.** No nível seguinte estão as matrizes, que são coleções de tipos de dados organizados. Em seguida, estão os registros, que são dados de diferentes tipos, acessados sob um mesmo nome. Transcendendo estes aspectos físicos dos dados, o nível final concentra-se na determinação da forma como os dados serão *armazenados* e *recuperados*. Basicamente, os dados estão ligados às "máquinas de dados" que controlam o modo como seu programa acessa as informações. Existem quatro dessas máquinas:

- Filas
- Pilhas
- Listas encadeadas
- Árvores binárias

Cada método fornece solução para um conjunto de problemas; cada um é essencialmente um "dispositivo" que executa uma determinada operação de armazenamento e recuperação de uma determinada informação solicitada. Os métodos têm duas operações em comum:

armazenar um item e *recuperar um item* no qual o item é uma unidade informacional. Este capítulo mostra como desenvolver esses métodos para uso nos seus próprios programas.

FILAS

Uma *fila* é uma lista linear de informação acessada na ordem "primeiro-dentro primeiro-fora" (*first-in first-out*, algumas vezes chamada FIFO). O primeiro item colocado na fila será o primeiro a ser recuperado, o segundo item será o segundo a ser recuperado e assim por diante. Esta ordem é o único modo de armazenagem e recuperação possível na fila; ela não permite acesso aleatório de nenhum item específico.

As filas são comuns na vida diária. Por exemplo, uma seqüência de pessoas num banco ou lanchonete é uma fila – exceto quando os fregueses resolvem furá-la. Para visualizar como funciona a fila, considere as rotinas **Guarda** e **Pega. Guarda** coloca um item no fim da fila, e **Pega** recupera o primeiro item da fila, retornando seu valor. A Figura 3-1 mostra o efeito de uma série dessas operações. Tenha em mente que uma operação de recuperação remove um item da fila e, se esse item não estiver armazenado em algum outro lugar, ele será efetivamente destruído – o item não poderá ser acessado de novo.

Ação	Conteúdo da fila
Guarda(A)	A
Guarda(B)	A B
Guarda(C)	A B C
Pega retorna **A**	B C
Guarda(D)	B C D
Pega retorna **B**	C D
Pega retorna **C**	D

Figura 3-1 Uma fila em ação.

Filas são usadas em muitos tipos de situações de programação, como em simulações (discutidas mais tarde no seu próprio capítulo), listas de eventos (como em PERT) e *buffering* de entrada/saída.

Como exemplo, considere um programa simples escalador de eventos que permita a entrada de uma série de eventos. À medida que cada evento é concluído, ele é tirado da lista e é mostrado o seguinte. Você poderia utilizar um programa como este para organizar eventos tais como os compromissos de um dia. Para simplificar os exemplos, o programa utiliza uma matriz de *strings* no armazenamento dos eventos. Limitamos o número de eventos em 100 e a descrição de cada um deles em 80 caracteres. Primeiro, há o procedimengo **Guarda** e a função **Pega** que serão usados no programa. Eles são mostrados aqui como as variáveis globais necessárias e as definições de tipo.

```
const
   MAXIMO = 100;
type
   Tipo_Evento = string[80];

var
   evento: array[0..MAXIMO] of Tipo_Evento;
   livre, proximo: integer;

procedure Guarda(q:Tipo_Evento);
begin
   if livre=MAXIMO then
      WriteLn('Fila lotada.')
   else
   begin
      evento[livre]:=q;
      livre:=livre+1;
   end;
end;  {Guarda}

function Pega:Tipo_Evento;
begin
   if proximo=livre then
   begin
      WriteLn('Nenhum evento na fila.');
      Pega:='';
   end else
   begin
      proximo:=proximo+1;
      Pega:=evento[proximo-1];
   end;
end;  {Pega}
```

Estas funções usam três variáveis globais: **livre**, que armazena o índice das posições de armazenamento livres que se seguem; **proximo** que indica o próximo item a ser recuperado, e **evento** que é a matriz de *strings* que armazena as informações. Antes que o programa possa chamar tanto **Guarda** quanto **Pega**, as variáveis **livre** e **proximo** devem ser inicializadas em zero.

Neste programa, o procedimento **Guarda** coloca novos eventos no fim da lista e verifica se ela está cheia. Enquanto houver eventos a serem executados é a função **Pega** que as retira da fila. Quando um novo evento estiver escalado, **livre** é incrementado.

Basicamente, **proximo** "caça" **livre** através da fila. A Figura 3-2 mostra como este processo ocorre na memória, à medida que o programa é executado. Se **livre** igualar-se a **proximo**, isso significa que não há mais nenhum elemento restante na lista. Tenha em mente que apesar da informação armazenada na fila não ser realmente destruída pela função **Pega**, ela nunca mais poderá ser acessada novamente e terá sido efetivamente perdida.

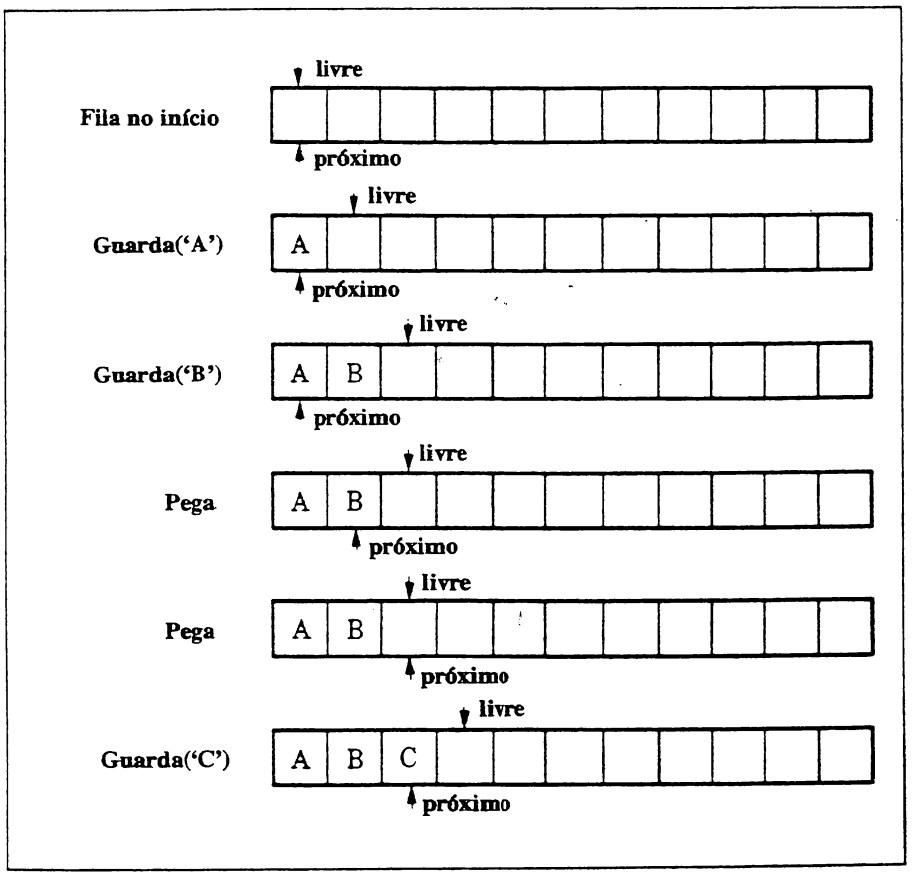


Figura 3-2 O índice de recuperação caçando o índice de armazenamento.

Aqui está o programa completo para este escalador de eventos simples. Você pode querer ampliá-lo para seu próprio uso.

```
program Mini_Agenda;

const
   MAXIMO = 100;

type
   Tipo_Evento = string[80];

var
   evento: array[0..MAXIMO] of Tipo_Evento;
   livre, proximo,t: integer;
   feito: boolean;
   car: char;

procedure Guarda(q:Tipo_Evento);
begin
   if livre=MAXIMO then
      WriteLn('Fila lotada.')
   else
   begin
      evento[livre]:=q;
      livre:=livre+1;
   end;
end;  {Guarda}

function Pega:Tipo_Evento;
begin
   if proximo=livre then
   begin
      WriteLn('Nenhum evento na fila.');
      Pega:='';
   end else
   begin
      proximo:=proximo+1;
      Pega:=evento[proximo-1];
   end;
end;  {Pega}

procedure Entrada;
var
   s: string[80];

begin
   repeat
      Write ('Entre evento ', livre+1, ':');
      Read(s);
      WriteLn;
      if Length(s) <> 0 then Guarda(s);
   until Length(s)=0;
end;  {Entrada}

procedure Mostra;
var
   t: integer;

begin
   for t:=proximo to livre-1 do WriteLn(t, ':', evento[t]);
end;  {Mostra}
```

```
procedure Prox_Compromisso;
var
   s:string[80];
begin
   s:=Pega;   {le o proximo evento}
   if Length(s) <> 0 then WriteLn(s);
end;  {Prox_Compromisso}

begin {agenda}
   for t:=1 to MAXIMO do evento[t]:=''; {inicializa eventos}
   livre:=0; proximo:=0; feito:=FALSE;

   repeat
      Write('Entrar, Mostrar, Prox_Compromisso, Fim: ');
      Read(car);
      WriteLn;
      case upcase(car) of
         'E': Entrada;
         'M': Mostra;
         'P': Prox_Compromisso;
         'F': feito:=TRUE;
      end;
   until feito=TRUE;
end.
```

A FILA CIRCULAR

Na seção anterior, talvez você tenha pensado em um melhoramento para o programa **Mini_Agenda.** Em vez do programa parar quando alcança o limite da matriz que armazena a fila, você pode fazer com que tanto o índice de armazenamento **livre** como o índice de recuperação **proximo** voltem ao início da matriz. Este método permitiria que um número qualquer de dados fosse colocado na fila, desde que os itens também fossem retirados. Essa implementação de fila é chamada *fila circular* pois é utilizada uma matriz armazenadora parecida com um círculo, em vez de uma lista linear.

Para criar uma fila circular no programa **Mini_Agenda**, você deve mudar os subprogramas **Guarda** e **Pega,** como mostrado a seguir:

```
procedure Guarda(q:Tipo_Evento);
begin
   if livre+1=proximo then
      WriteLn('Fila lotada.')
   else
   begin
      evento[livre]:=q;
      livre:=livre+1;
      if livre=MAXIMO then livre:=1; {fecha o circulo}
   end;
end;  {Guarda}
```

```
function Pega:Tipo_Evento;
begin
   if proximo=MAXIMO then proximo:=1;  {volta ao inicio}
   if proximo=livre then
   begin
      WriteLn('Fim da Fila.');
      Pega:=';';
   end else
   begin
      proximo:=proximo+1;
      Pega:=evento[proximo-1];
   end;
end;  {Pega}
```

Na verdade, a fila só estará cheia quando tanto o índice de armazenamento quanto o de recuperação forem iguais; de outro modo, a fila ainda terá lugar para outro evento. Entretanto, isso quer dizer que quando o programa começar, o índice de recuperação **proximo** não poderá ser zerado mas ajustado para **MAXIMO**, de modo que a primeira chamada de **Guarda** não produza a mensagem **Fila lotada.** Note que a fila reterá apenas **MAXIMO–1** elementos pois **proximo** e **livre** devem estar separados sempre por um elemento; do contrário, seria impossível saber se a fila está cheia ou vazia. A Figura 3-3 mostra a matriz usada para versão circular do programa **Mini_Agenda.** O uso mais comum para uma fila circular pode ser na operação de sistemas que "retêm" a informação lida e escrita em um arquivo de disco ou console (*buffers*). Um outro uso comum ocorre na aplicação de programas em tempo-real, nos quais, por exemplo, o usuário pode continuar a introduzir dados pelo teclado enquanto o programa executa uma outra tarefa. Muitos

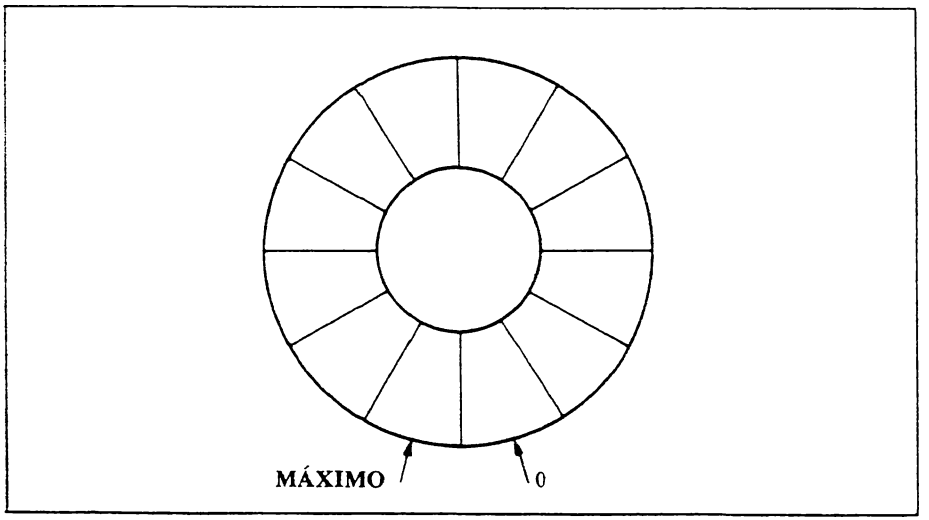


Figura 3-3 A matriz circular do programa **Mini_Agenda.**

processadores de texto fazem isso quando reformatam um parágrafo ou justificam uma linha. Durante um curto intervalo, o que é digitado não é mostrado na tela; este intervalo dura até que o programa complete o outro processo no qual ele trabalhava. Para realizar isto, o programa de aplicação deve continuar verificando entradas do teclado durante a execução do outro processo. Se uma tecla tiver sido digitada, ela será rapidamente colocada na fila e o processo continuará. Depois de o processo ter sido completado, os caracteres serão recuperados da fila e tratados de maneira normal.

Para ver como isto pode ser feito com uma fila circular, estude o programa simples a seguir, que contém os dois processos. O primeiro processo conta até 32.000. O segundo processo coloca caracteres em uma fila circular à medida que eles vão sendo digitados, sem ecoá-los na tela, até que seja encontrado um ponto-e-vírgula. Os caracteres que você digitar não serão mostrados, pois o primeiro processo será priorizado até que seja digitado um ponto-e-vírgula ou até que a contagem termine. Os caracteres serão recuperados e impressos.

```
program Buffer_Teclado;

const
   MAXIMO=10;
type
   Tipo_Evento= char;
   regs= record
        al: char;
        ah: byte;
        bx, cx, dx, bp, si, di, es, flags: integer
   end;

var
   evento: array[0..MAXIMO] of Tipo_Evento;
   livre, proximo, t: integer;
   car: char;
   dos: regs;

procedure Guarda(q:Tipo_Evento);
begin
   if livre+1=proximo then
      WriteLn('Fila lotada.')
   else
   begin
      evento[livre]:=q;
      livre:=livre+1;
      if livre=MAXIMO then livre:=0; {fecha o circulo}
   end;
end;  {Guarda}

function Pega:Tipo_Evento;
begin
   if proximo=MAXIMO then proximo:=0;  {volta ao inicio}
   if proximo=livre then
   begin
      WriteLn;
```

```
      WriteLn('Fim da Fila.');
      Pega:=';';
   end else
   begin
      proximo:=proximo+1;
      Pega:=evento[proximo-1];
   end;
end;  {Pega}

begin    {buffer_teclado}
   proximo:=MAXIMO; livre:=0;
   dos.ah:=7; dos.al:=' ';  {inicializa}
   t:=1;
   repeat
      if KeyPressed then
      begin
         MsDos(dos); {leitura sem eco no video}
         Guarda(dos.al);
      end;
      t:=t+1;
   until (t=32000) or (dos.al=';');
   repeat
      car:=Pega;
      if car <> ';' then Write(car);
   until car=';';
end.
```

A rotina **KeyPressed** usa uma função de chamada ao sistema operacional, esta retorna **TRUE** se alguma tecla for pressionada ou **FALSE**, em caso contrário. A chamada do **MsDos** lê uma tecla do teclado sem ecoá-la na tela. Essas chamadas só funcionam para o IBM PC; se você tiver um computador diferente, deverá consultar o *Manual do usuário do Turbo Pascal* para encontrar os procedimentos corretos a usar. (No Capítulo 5 você aprenderá com profundidade a usar esta e outras chamadas ao sistema operacional.)

PILHAS

Uma *pilha* é o oposto de uma fila, pois utiliza acesso "último-dentro, primeiro-fora" (*last-in, first-out*, às vezes chamado LIFO). Imagine uma pilha de pratos: o prato do fim da pilha é o último a ser usado e o prato do topo (o último colocado na pilha) é o primeiro a ser usado. Pilhas são bastante usadas em *software* de sistemas, incluindo interpretadores e compiladores.

Por razões históricas, as duas primeiras operações de pilha – *armazenamento* e *recuperação* – são em geral chamadas, respectivamente, *push* e *pop*. Portanto, para implementar uma pilha, você precisa de duas funções: **Push**, que coloca um valor no topo

da pilha e **Pop**, que recupera um valor da pilha. Você também precisará de uma região de memória para ser usada como pilha. Isto pode ser feito utilizando-se uma matriz ou através da alocação de uma região da memória, o que é conseguido pelo uso de uma função de alocação de memória dinâmica no Turbo Pascal. Como a fila, a função de recuperação toma um valor da lista e se este valor não estiver armazenado em outro lugar, ele a destrói. Aqui estão as formas gerais de **Push** e **Pop** que usam uma matriz de inteiros:

```
const
   MAX=100;

var
   pilha: array[1..100] of integer;
   topo: integer;  {indica o topo da pilha}
                   {deve comecar com 1}

procedure Push(i: integer);
begin
   if topo >= MAX then WriteLn('Pilha cheia.')
   else
   begin
      pilha[topo]:=i;
      topo:=topo+1;
   end;
end;  {Push}

function Pop: integer;
begin
   topo:=topo-1;
   if topo < 1 then
   begin
      WriteLn('Pilha vazia!');
      topo:=topo+1;
      Pop:=0;
   end
   else Pop:=pilha[topo];
end;  {Pop}
```

A variável **topo** é o índice da próxima posição aberta na pilha. Quando implementar estas funções, lembre-se *sempre* de evitar *overflow* e *underflow*. Nestas rotinas, se **topo** é 0, a pilha está vazia; se **topo** for maior ou igual ao último local de armazenamento disponível, a pilha estará cheia. A Figura 3-4 mostra como funciona uma pilha.

Um excelente exemplo do uso de pilhas é uma calculadora de quatro funções. A maioria das calculadoras hoje aceita uma forma padrão de expressão chamada *notação infixa (infix notation)*, que adota a forma geral *operando-operador-operando*. Por exemplo, para somar 100 a 200, você introduziria **100**, digitaria +, introduziria **200**, e digitaria =. Entretanto, algumas calculadoras usam a chamada *notação posfixa (postfix notation)*, na qual ambos os operandos são introduzidos antes do operador. Por exemplo, para somar 100 a 200, usando notação posfixa, você introduziria primeiro 100, depois 200 e então digitaria +. À medida que os operandos vão sendo introduzidos, vão sendo

Ação	Conteúdo da pilha
Push(A)	A
Push(B)	B A
Push(C)	C B A
Pop retorna C	B A
Push(F)	F B A
Pop retorna F	B A
Pop retorna B	A
Pop retorna A	vazia

Figura 3-4 Uma pilha em ação.

colocados numa pilha; quando um operador é introduzido, dois operandos são removidos da pilha e o resultado é colocado de volta na pilha. A vantagem da forma posfixa é que expressões muito complexas podem ser calculadas facilmente pela calculadora, sem muita programação.

O programa calculadora é mostrado inteiro aqui:

```
program Calculadora_Quatro_Operacoes;

const
   MAX=100;

var
   pilha: array[1..100] of integer;
   topo: integer;  {indica o topo da pilha}
                   {deve comecar com 1}
   a, b: integer;
   s: string[80];

procedure Push(i: integer);
begin
   if topo >= MAX then WriteLn('Pilha cheia.')
   else
   begin
      pilha[topo]:=i;
      topo:=topo+1;
   end;
end;  {Push}

function Pop: integer;
begin
   topo:=topo-1;
   if topo < 1 then
   begin
      WriteLn('Pilha vazia!');
```

```
      topo:=topo+1;
      Pop:=0;
   end
   else Pop:=pilha[topo];
end;  {Pop}

begin  {Calculadora}
   topo:=1;
   WriteLn('Calculadora de Quatro Operacoes (F = Fim)');
   repeat
      Write(': ');
      Read(s);
      WriteLn;
      Val(s,a,b);
      if (b=0) and ((Length(s) > 1) or (s[1] <> '-')) then Push(a)
      else
      case s[1] of
           '+': begin
                     a:=Pop;
                     b:=Pop;
                     WriteLn(a+b);
                     Push(a+b);
                end;
           '-': begin
                     a:=Pop;
                     b:=Pop;
                     WriteLn(b-a);
                     Push(b-a);
                end;
           '*': begin
                     a:=Pop;
                     b:=Pop;
                     WriteLn(a*b);
                     Push(a*b);
                end;
           '/': begin
                     a:=Pop;
                     b:=Pop;
                     if a=0 then WriteLn('Divisao por zero!')
                     else begin
                          WriteLn(b div a);
                          Push(b div a);
                     end;
                end;
      end;
   until UpCase(copy(s,1,1))='F';
end.
```

Embora esta versão seja capaz de operar apenas com números inteiros, seria simples alterá-la para operações de ponto-flutuante pela mudança dos tipos de dados da pilha e conversão do operador **div** para operador de ponto-flutuante (*/*).

LISTAS ENCADEADAS

Filas e pilhas possuem dois traços em comum. Primeiro, ambas possuem regras estritas para a referência de dados nelas armazenados. Segundo, as operações de recuperação são, por natureza, *destrutivas*; ou seja, o acesso a um item de uma pilha ou fila requer sua remoção e, a menos que ele esteja armazenado em outro lugar qualquer, sua destruição. Tanto as pilhas quanto as filas requerem, pelo menos teoricamente, uma região contígua da memória para poderem operar.

Ao contrário de uma pilha ou de uma fila, uma *lista encadeada* pode acessar sua memória de uma maneira aleatória, pois cada fragmento de informação carrega com ele um *elo (link)* para o próximo item de dado na cadeia. Uma lista encadeada requer uma estrutura de dados complexa, enquanto uma pilha ou fila podem operar tanto com itens de dados complexos quanto com simples. Uma operação de recuperação em lista encadeada não remove nem destrói itens da lista; para que isto seja feito deve ser adicionada uma *operação de remoção*.

Listas encadeadas são usadas para dois propósitos. O primeiro é criar matrizes de tamanhos desconhecidos na memória. Se você souber, de antemão, o volume de armazenamento, poderá utilizar uma matriz simples; mas se não conhecer o tamanho real da lista, então você deverá usar uma lista encadeada. O segundo propósito é o da armazenagem de bancos de dados em disco. A lista encadeada permite que você insira e delete itens, rapidamente, sem rearranjar todo o arquivo de disco. Por essas razões, as listas encadeadas são usadas extensivamente em software de gerenciamento de banco de dados.

Listas encadeadas podem ser tanto de ligação simples quanto de ligação dupla. Uma lista de ligação simples contém uma ligação para o próximo item de dado. Uma lista de ligação dupla possui ligações tanto para o item seguinte quanto para o anterior. O tipo que você vai usar depende da aplicação.

LISTAS ENCADEADAS SIMPLES

Uma lista encadeada simples requer que cada item de informação contenha uma ligação com o item seguinte da lista. Cada item de dado geralmente consiste em um registro que contém tanto campos de informação como um *pointer* de ligação. O conceito de lista encadeada simples é mostrado na Figura 3-5.

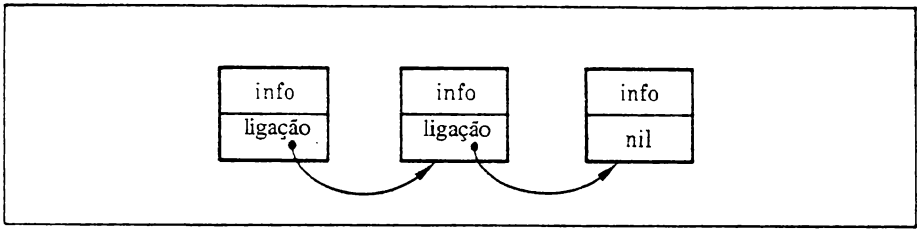


Figura 3-5 Uma lista de memória de ligação simples.

Há dois modos de se construir uma lista de ligação simples. O primeiro consiste simplesmente em adicionar cada novo item ao começo ou ao fim da lista. O outro consiste em colocar itens em lugares específicos da lista (por exemplo, em ordem ascendente).

A maneira como você constrói a lista determina o modo como a *função de armazenamento* será programada, como é mostrado no caso simples de criação de uma lista ligada pela adição de itens no final. Você precisa definir um registro para armazenar a informação e as ligações. Pelo fato das malas diretas serem comuns, este exemplo utiliza uma. O tipo de registro para cada elemento usado na mala direta é definido aqui. (Ele é similar à definição de elementos do Capítulo 2.)

```
Ptr_Endereco = ^address;
endereco = record
        nome: string[30];
        rua: string[40];
        cidade: string[20];
        estado: string[2];
        CEP: string[5];
        proximo: Ptr_Endereco; (aponta para o proximo registro)
end;
var
   primeiro, ultimo: Ptr_Endereco;

{_____________________________________________________________}
```

A função **Armazena_ES** constrói uma lista encadeada simples, colocando cada novo elemento no fim. Um *pointer* para um registro de tipo **endereco** deve ser passado para **Armazena_ES**, como mostrado aqui:

```
procedure Armazena_Es(i: Ptr_Endereco);
begin
   if ultimo=nil then (first item in list)
   begin
      ultimo:=i;
      primeiro:=i;
      i^.proximo:=nil;
   end else
```

```
    begin
      ultimo^.proximo:=i;
      i^.proximo:=nil;
      ultimo:=i;
    end;
end; {Armazena_Es}
```

Embora você possa ordenar a lista criada com **Armazena_ES** numa operação separada, é mais fácil ordenar durante a construção da lista, pela inserção de cada novo item na posição adequada da cadeia. Além disso, se a lista já estiver ordenada, é vantajoso mantê-la ordenada, inserindo os novos itens nos seus devidos lugares. Para fazer isso, a lista é lida seqüencialmente até que a posição certa seja encontrada; o novo endereço então é inserido naquele ponto, e as ligações são rearranjadas de acordo.

Três situações possíveis podem ocorrer quando da inserção de um item em uma lista de ligação simples. Primeiro, o item pode se tornar o novo primeiro item; segundo, ele pode ser inserido entre dois outros itens; ou, terceiro, ele pode se tornar o último item. A Figura 3-6 mostra como as ligações são mudadas em cada caso.

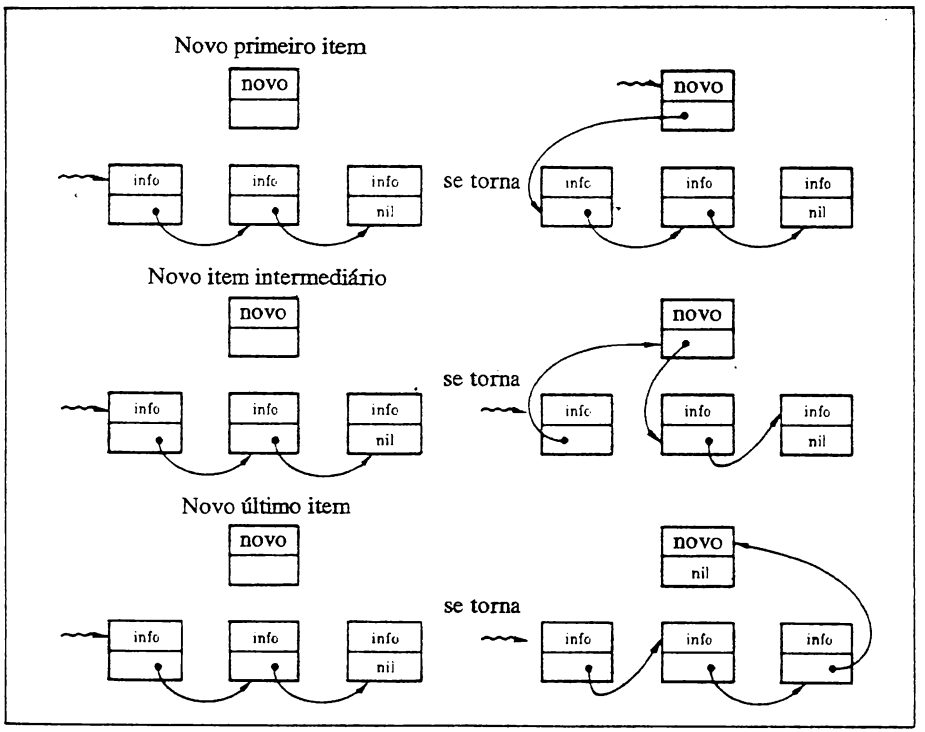


Figura 3-6 Inserindo um item em uma lista de ligação simples.

Se você mudar o primeiro item da lista, deve atualizar o ponto de entrada para ela em outro lugar do seu programa. Para evitar isso, você pode usar um *sentinela* como primeiro item. Um sentinela é um valor especial que sempre será o primeiro da lista. Com esse método, você pode evitar que o ponto de entrada da lista mude. Entretanto, esse método tem a desvantagem de usar uma locação extra para armazenar o sentinela e por isso não será usado aqui.

A função **Faz_lista_ESQ**, mostrada aqui, insere endereços na lista de correspondência em ordem crescente, baseada no campo **nome**. Ela retorna um *pointer* ao

```
function Faz_Lista_ESQ(item, primeiro: ptr_endereco;
                       var ultimo: ptr_endereco): ptr_endereco;
{Armazena dados em lista encadeada simples ordenada}

var
   velho, topo: ptr_endereco;
   feito: boolean;
begin
   topo:=primeiro;
   velho:=nil;
   feito:=FALSE;

   if primeiro=nil then
   begin {primeiro elemento da lista}
      item^.proximo:= nil;
      ultimo:=item;
      Faz_Lista_ESQ:=item;
   end else
   begin
      while (primeiro <> nil) and (not feito) do begin
         if primeiro^.nome < item^.nome then begin
            velho:=primeiro;
            primeiro:=primeiro^.proximo;
         end else begin  {para o meio da lista}
            if velho <> nil then begin
                 velho^.proximo:=item;
                 item^.proximo:=primeiro;
                 Faz_Lista_ESQ:=topo;  {mantem o mesmo inicio}
                 feito:=TRUE;
            end else begin
                 item^.proximo:=primeiro;  {novo primeiro elemento}
                 Faz_Lista_ESQ:= item;
                 feito:=TRUE;
            end;
         end;
      end;  {while}
      if not feito then
      begin
         ultimo^.proximo:=item;   {para o fim da lista}
         item^.proximo:=nil;
         ultimo:=item;
         Faz_Lista_ESQ:=topo;
      end;
   end;
end;  {Faz_Lista_ESQ}
```

primeiro elemento da lista e também requer que os *pointers*, tanto do início quanto do fim da lista, sejam passados para ela.

Em uma lista encadeada é incomum encontrarmos uma função específica dedicada ao *processo de recuperação*, que retorne item após item na ordem da lista. Este procedimento é em geral tão curto que é simplesmente colocado em outra rotina, como a de busca, deleção, ou exibição. Por exemplo, essa rotina exibe todos os nomes em uma lista de correspondência:

```
procedure Mostra(primeiro: ptr_endereco);
begin
   while primeiro <> nil do begin
      WriteLn(primeiro^.nome);
      primeiro:=primeiro^.proximo;
   end;
end;  {Mostra}
```

Aqui, **primeiro** é um *pointer* para o primeiro registro na lista.

Recuperar itens da lista é tão simples quanto seguir uma cadeia. Você poderia escrever uma rotina de busca baseada no campo **nome** como esta:

```
function Busca(primeiro: ptr_endereco; nome: str80): ptr_endereco;
var
   feito: boolean;
begin
   feito:=FALSE;
   while (primeiro <> nil) and (not feito) do
   begin
      if nome=primeiro^.nome then
      begin
         Busca:=primeiro;
         feito:=TRUE;
      end else
         primeiro:=primeiro^.proximo;
   end;
   if primeiro=nil then Busca:=nil;  {nao esta' na lista}
end;  {Busca}
```

Como **Busca** retorna um *pointer* ao item da lista que corresponde ao nome procurado, a função deve ser declarada como um *pointer* para o tipo **endereco.** Se o nome procurado não existir, um *pointer* **nil** será retornado.

O processo de eliminação de um item de uma lista de ligação simples é direto. Como na inserção, há três casos: eliminação do primeiro item, eliminação de item intermediário e eliminação do último item. A Figura 3-7 mostra cada caso.

Esta função apaga um item de uma lista de registros do tipo **endereco.**

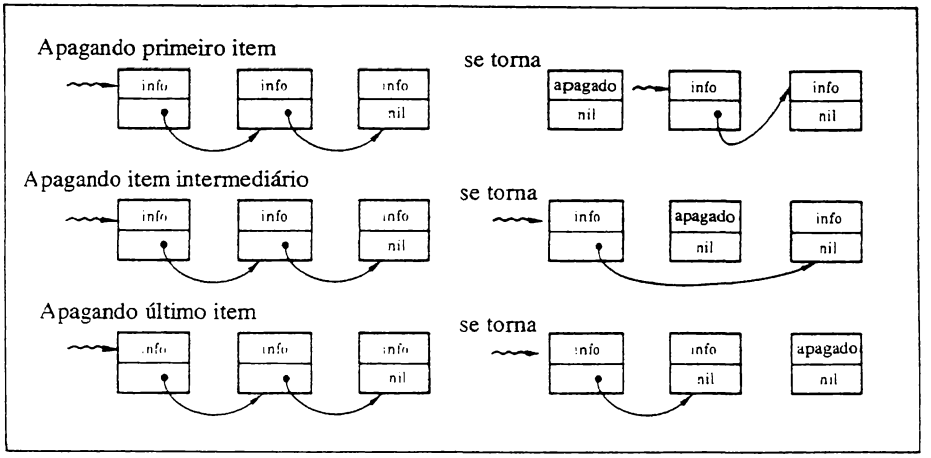


Figura 3-7 Apagando um item de uma lista encadeada um a um.

```
function Apaga_ES(primeiro, item, anterior: ptr_endereco): ptr_endereco;

begin
   if anterior <> nil then
      anterior^.proximo:=item^.proximo
   else primeiro:=item^.proximo;
   Apaga_ES:=primeiro;
end;  {Apaga_ES}
```

Apaga_ES deve receber *pointers* para o item a ser apagado, para o item imediatamente anterior a ele na cadeia e para o início da lista. Se o primeiro item deve ser removido, o *pointer* **anterior** deverá ser **nil.** A função deve retornar um *pointer* ao início da lista no caso em que o primeiro item é apagado – o programa deve saber onde o novo primeiro elemento está alocado.

Listas de ligação simples possuem uma desvantagem maior que impede seu largo emprego: a lista não pode ser seguida em ordem inversa. Por essa razão, listas de ligação dupla geralmente são usadas.

LISTAS DE LIGAÇÃO DUPLA

Listas de ligação dupla consistem em dados ligados tanto ao item anterior quanto ao posterior. A Figura 3-8 mostra como as ligações são organizadas. Uma lista que possua duas ligações em vez de uma tem duas vantagens principais. Primeiro, a lista pode ser lida em qualquer direção. Isso não só simplifica a ordenação da lista como, no caso de um

banco de dados, permite que o usuário pesquise a lista em qualquer direção. Segundo, se uma ligação se tornar inválida, a lista poderá ser reconstituída seguindo-se as ligações no sentido inverso, pois pode-se ler a lista tanto em um sentido como no outro. Isto, entretanto, só é significativo em caso de falha do equipamento.

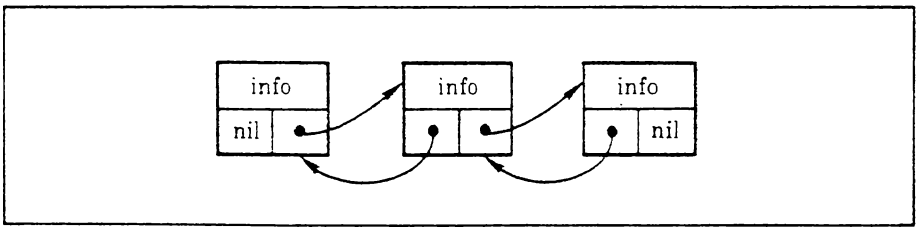


Figura 3-8 Uma lista de ligação dupla.

Três operações primárias podem ser executadas em uma lista de ligação dupla: inserção de um novo primeiro item, inserção de um novo elemento intermediário, e inserção de um novo elemento final. Essas ordenações são mostradas na Figura 3-9.

A construção de uma lista de ligação dupla é similar à construção de uma lista de ligação simples, exceto por ser necessário que o registro tenha lugar para manter duas ligações. Usando novamente o exemplo da mala direta, você pode modificar **endereco**, como mostrado aqui, para acomodar o seguinte:

```
type
   str80 = string[80];
   Ptr_Endereco = ^endereco;
   endereco = record
            nome: string[30];
            rua: string[40];
            cidade: string[20];
            estado: string[2];
            cep: string[5];
            proximo: Ptr_Endereco; {aponta para o proximo registro}
            anterior: Ptr_Endereco; {aponta para o registro anterior}
   end;
```

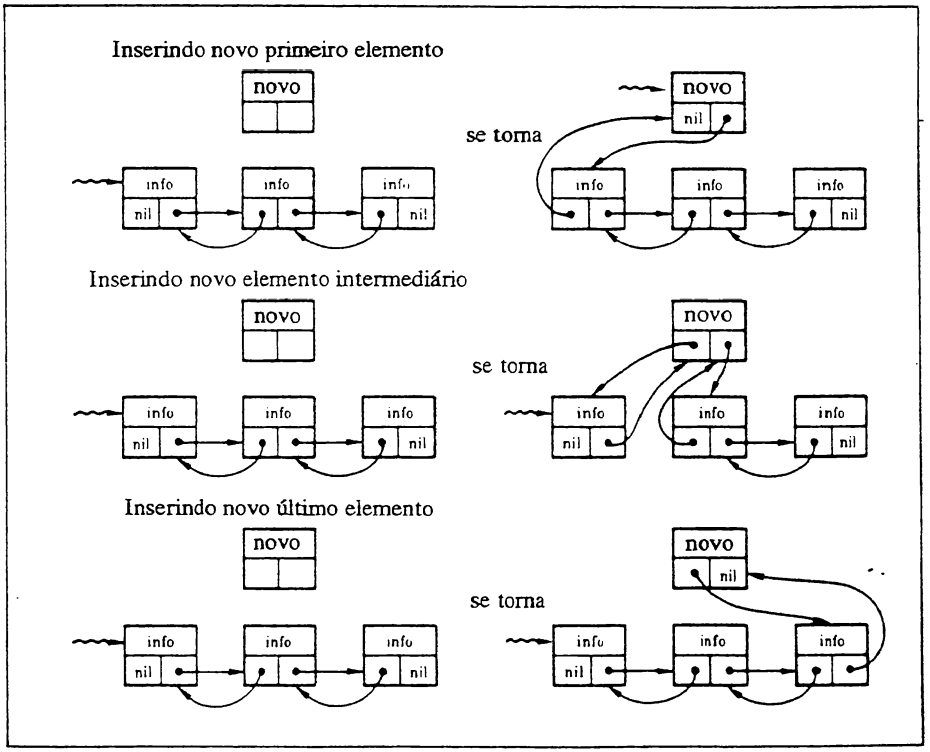


Figura 3-9 Inserindo um item em uma lista de ligação dupla.

Usando o registro **endereco** como item básico, o procedimento **Armazena_ED** constrói uma lista de ligação dupla:

```
procedure Armazena_ED( i: Ptr_Endereco );
begin
   if ultimo=nil then   {primeiro item da lista}
   begin
      ultimo:=i;
      primeiro:=i;
      i^.proximo:=nil;
      i^.anterior:=nil;
   end
   else
       begin
       ultimo^.proximo:=i;
       i^.proximo:=nil;
       i^.anterior:=ultimo;
       ultimo:=i;
   end;
end;  {Armazena_ED}
```

Este procedimento coloca cada nova entrada no final da lista.

Como no caso da lista de ligação simples, uma lista de ligação dupla pode possuir uma função que armazene cada elemento em uma alocação específica na lista enquanto ela é montada, em vez de colocar o novo item sempre no final. A função

```
function Armazena_EDO(item, primeiro: Ptr_Endereco;
                      var ultimo: Ptr_Endereco): Ptr_Endereco;
{armazena dados em lista encadeada dupla ordenada}

var
   velho, topo: Ptr_Endereco;
   feito: boolean;
begin
   topo:=primeiro;
   velho:=nil;
   feito:=FALSE;

   if primeiro=nil then
   begin {primeiro elemento da lista}
      item^.proximo:= nil;
      ultimo:=item;
      item^.anterior:=nil;
      Armazena_EDO:=item;
   end else
   begin
      while (primeiro <> nil) and (not feito) do begin
         if primeiro^.nome < item^.nome then begin
            velho:=primeiro;
            primeiro:=primeiro^.proximo;
         end else begin  {para o meio}
            if velho <> nil then begin
                 velho^.proximo:=item;
                 item^.proximo:=primeiro;
                 primeiro^.anterior:=item;
                 item^.anterior:=velho;
                 Armazena_EDO:=topo;  {mantem o mesmo inicio}
                 feito:=TRUE;
            end else begin
                 item^.proximo:=primeiro;  {novo primeiro elemento}
                 item^.anterior:=nil;
                 Armazena_EDO:= item;
                 feito:=TRUE;
            end;
         end;
      end;  {while}
      if not feito then
      begin
         ultimo^.proximo:=item;    {para o fim}
         item^.proximo:=nil;
         item^.anterior:=ultimo;
         ultimo:=item;
         Armazena_EDO:=topo;
      end;
   end;
end;  {Armazena_EDO}
```

Armazena_EDO cria uma lista que é ordenada em ordem crescente baseada no campo **nome.**

Como um item pode ser inserido no topo da lista, esta função deve retornar um *pointer* ao primeiro item para que as outras partes do programa saibam onde começa a lista. Como na lista encadeada simples, para que o programa recupere um item de dado específico, ele deverá seguir as ligações até que seja encontrado o item correto.

Há três casos a considerar quando da eliminação de um item de uma lista encadeada dupla: eliminação do primeiro item, eliminação de um item intermediário e eliminação do último item. A Figura 3-10 mostra como as ligações são rearranjadas.

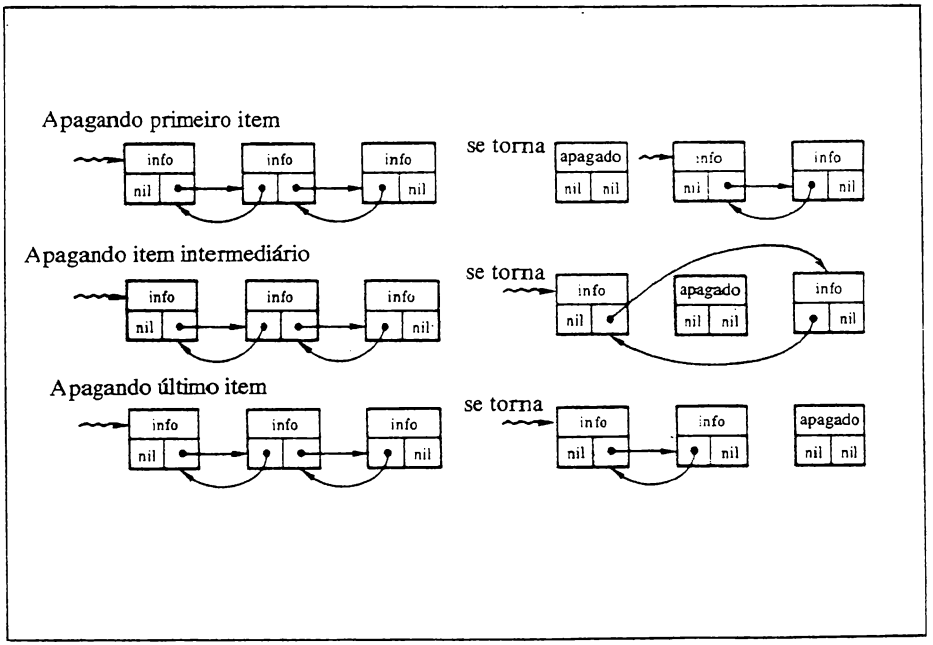


Figura 3-10 Eliminação de um item de uma lista encadeada dupla.

A seguinte função apaga um item do tipo **endereco** de uma lista encadeada dupla.

```
function Apaga_ED(primeiro, item: Ptr_Endereco): Ptr_Endereco;
begin
   if primeiro=item then begin  {delete first in list}
      Apaga_ED:=primeiro^.proximo;
      if item^.proximo <> nil then begin
         item^.proximo^.anterior:=nil;
      end;
      dispose(primeiro);
   end else begin
      item^.anterior^.proximo:=item^.proximo;
      item^.proximo^.anterior:=item^.anterior;
      Apaga_ED:=primeiro;  {ainda o mesmo inicio}
   end;
end;  {Apaga_ED}
```

Esta função, em relação à lista encadeada simples, requer que um *pointer* a menos seja passado para ela; o item de dado que está sendo apagado já carrega uma ligação ao elemento anterior e ao posterior. Como pode haver mudança do primeiro item da lista, o *pointer* do item do topo é passado de volta à rotina de chamada.

UMA MALA DIRETA QUE USA UMA LISTA ENCADEADA DUPLA

Aqui está um programa simples de mala direta que utiliza uma lista encadeada dupla. A lista inteira é mantida na memória enquanto em uso; entretanto, o programa pode ser modificado para armazenar a mala direta em arquivo de disco.

```
program Mala_Direta;

type
   str80 = string[80];
   Ptr_Endereco = ^endereco;
   endereco = record
            nome: string[30];
            rua: string[40];
            cidade: string[20];
            estado: string[2];
            cep: string[5];
            proximo: Ptr_Endereco; {aponta para o proximo registro}
            anterior: Ptr_Endereco; {aponta para o registro anterior}
   end;

   tipo_arq = file of endereco;

var
   t,t2: integer;
   mala_dir: tipo_arq;
   primeiro,ultimo: Ptr_Endereco;
   feito: boolean;
```

```
function Menu:char;  {retorna opcao do usuario}

var
   car: char;

begin
   WriteLn('1. Digitar nomes');
   WriteLn('2. Apagar um nome');
   WriteLn('3. Ver lista');
   WriteLn('4. Procurar um nome');
   WriteLn('5. Gravar lista');
   WriteLn('6. Ler lista');
   WriteLn('7. Fim');
   repeat
      WriteLn;
      Write('Digite a sua opcao: ');
      Read(car); car:=UpCase(car); WriteLn;
   until (car >= '1') and (car <= '7');
   Menu:=car;
end;  {Menu}

function Armazena_EDO(item, primeiro: Ptr_Endereco:
                      var ultimo: Ptr_Endereco): Ptr_Endereco;
{armazena dados em lista encadeada dupla ordenada}

var
   velho, topo: Ptr_Endereco;
   feito: boolean;
begin
   topo:=primeiro;
   velho:=nil;
   feito:=FALSE;

      primeiro=nil then
   begin {primeiro elemento da lista}
      item^.proximo:= nil;
      ultimo:=item;
      item^.anterior:=nil;
      Armazena_EDO:=item;
   end else
   begin
      while (primeiro <> nil) and (not feito) do begin
         if primeiro^.nome < item^.nome then begin
            velho:=primeiro;
            primeiro:=primeiro^.proximo;
         end else begin  {para o meio}
            if velho <> nil then begin
                 velho^.proximo:=item;
                 item^.proximo:=primeiro;
                 primeiro^.anterior:=item;
                 item^.anterior:=velho;
                 Armazena_EDO:=topo;  {mantem o mesmo inicio}
                 feito:=TRUE;
            end else begin
                 item^.proximo:=primeiro;  {novo primeiro elemento
                 item^.anterior:=nil;
                 Armazena_EDO:= item;
                 feito:=TRUE;
            end;
```

```
        end;
      end;  {while}
      if not feito then
      begin
         ultimo^.proximo:=item;   {para o fim}
         item^.proximo:=nil;
         item^.anterior:=ultimo;
         ultimo:=item;
         Armazena_EDO:=topo;
      end;
   end;
end;  {Armazena_EDO}

function Apaga_ED(primeiro: Ptr_Endereco; chave: str80): Ptr_Endereco;
var
   temo, temp2: Ptr_Endereco;
   feito: boolean;
begin
   if primeiro^.nome=chave then begin  {apaga o primeiro da lista}
      Apaga_ED:=primeiro^.proximo;
      if temp^.proximo <> nil then begin
         temp:=primeiro^.proximo;
         temp^.anterior:=nil;
      end;
      dispose(primeiro);
   end else begin
      feito:=FALSE;
      temp:=primeiro^.proximo;
      temp2:=primeiro;
      while (temp <> nil) and (not feito) do
      begin
         if temp^.nome=chave then
         begin
            temo2^.proximo:=temp^.proximo;
            if temp^.proximo <> nil then
               temp^.proximo^.anterior:=temp2;
            feito:=TRUE;
            dispose(temo);
         end else
         begin
            temo2:=temo;
            temo:=temo^.proximo;
         end;
      end;
      Apaga_ED:=primeiro; {ainda o mesmo inicio}
      if not feito then WriteLn('Nao encontrado.');
   end;
end;  {Apaga_ED}

procedure Remove;
var
   nome: str80;
begin
   Write('Digite o nome a apagar: ');
   Read(nome); WriteLn;
   primeiro:=Apaga_ED(primeiro,nome);
end; {Remove}
```

```
procedure Digitacao;
var
   item: Ptr_Endereco;
   feito: boolean;
begin
   feito:=FALSE;
   repeat
      New(item); {faz novo registro}
      Write('Nome: ');
      Read(item^.nome); WriteLn;
      if Length(item^.nome)=0 then feito:=TRUE
      else
      begin
         Write('Rua: ');
         Read(item^.rua); WriteLn;
         Write('Cidade: ');
         Read(item^.cidade); WriteLn;
         Write('Estado: ');
         Read(item^.estado); WriteLn;
         Write('CEP: ');
         Read(item^.cep); WriteLn;
         primeiro:=Armazena_EDO(item,primeiro,ultimo); {armazena}
      end;
   until feito;
end;  {Digitacao}

procedure Mostra(primeiro: Ptr_Endereco);
begin
   while primeiro <> nil do begin
      WriteLn(primeiro^.nome);
      WriteLn(primeiro^.rua);
      WriteLn(primeiro^.cidade);
      WriteLn(primeiro^.estado);
      WriteLn(primeiro^.cep);
      primeiro:=primeiro^.proximo;
   end;
end;  {Mostra}

function Busca(primeiro: ptr_endereco; nome: str80): ptr_endereco;
var
   feito: boolean;
begin
   feito:=FALSE;
   while (primeiro <> nil) and (not feito) do
   begin
      if nome=primeiro^.nome then
      begin
         Busca:=primeiro;
         feito:=TRUE;
      end else
         primeiro:=primeiro^.proximo;
   end;
   if primeiro=nil then Busca:=nil;  {nao esta' na lista}
end;  {Busca}

procedure Procura;
```

```
var
   loc: Ptr_Endereco;
   nome: str80;
begin
   Write('Digite o nome: ');
   Read(nome); WriteLn;
   loc:=Busca(primeiro,nome);
   if loc <> nil then WriteLn(loc^.nome)
   else WriteLn('Nao esta'' na lista.'')
end;  {Procura}

procedure Grava(var f: tipo_arq; primeiro: Ptr_Endereco);
begin
   WriteLn('Gravando arquivo.');
   Rewrite(f);
   while primeiro <> nil do
   begin
      Write(f,primeiro^);
      primeiro:=primeiro^.proximo;
   end;
end;  {Grava}

function Le(var f:tipo_arq; primeiro:Ptr_Endereco): Ptr_Endereco;
{retorna um conteiro para o primeiro elemento da lista}
var
   temp,temp2: Ptr_Endereco;
   first: boolean;
begin
   WriteLn('Lendo arquivo.');
   reset(f);
   while primeiro    nil do
   begin {libera memoria}
      temp:=primeiro^.proximo;
      dispose(primeiro);
      primeiro:=temp;
   end;

   primeiro:=nil; ultimo:=nil;
   if not eof(f) then
   begin
      New(temp);
      Read(f,temp^);
      temp^.proximo:=nil; temp^.anterior:=nil;
      Le:=temp; {aponta para o primeiro elemento da lista}
   end;

   while not eof(f) do
   begin
      New(temp2);
      Read(f,temp2^);
      temp^.proximo:=temp2; {faz lista}
      temp2^.proximo:=nil;
      temp^.anterior:=temp2;
      temp:=temp2
   end;
   ultimo:=temp2;
end;  {Le}
```

```
begin
   primeiro:=nil; {lista inicialmente vazia}
   ultimo:=nil;
   feito:=FALSE;

   Assign(mala_dir, 'mala_dir.dat');

   repeat
      case Menu of
         '1': Digitacao;
         '2': Remove;
         '3': Mostra(primeiro);
         '4': Procura;
         '5': Grava(mala_dir,primeiro);
         '6': primeiro:=Le(mala_dir,primeiro);
         '7': feito:=TRUE;
      end;
   until feito=TRUE;

end.  {mala_dir}
```

ÁRVORES BINÁRIAS

A quarta estrutura de dados é a *árvore binária*. Embora possa haver muitos tipos de árvores, as árvores binárias são especiais, pois quando ordenadas elas se prestam a buscas, inserções e remoções rápidas. Cada item em uma árvore binária consiste em uma informação com uma ligação ao elemento da esquerda e uma ligação ao elemento da direita. A Figura 3-11 mostra uma árvore pequena.

A terminologia necessária para discutir árvores é um caso clássico de metáforas misturadas. A *raiz* é o primeiro item da árvore. Cada item de dado é chamado de nó (ou às vezes de *folha*) da árvore, e qualquer pedaço da árvore é chamado de subárvore. Um nó que não possua subárvores ligadas a ele é chamado de nó terminal. A *altura* da árvore é igual ao número de camadas que sua raiz cresce em profundidade. Ao longo de toda esta discussão, imagine que as árvores binárias aparecem na memória do mesmo jeito que aparecem no papel, mas lembre-se de que uma árvore é apenas uma maneira de estruturar dados na memória, e a memória possui um formato linear.

A árvore binária é uma forma especial de lista encadeada. Itens podem ser inseridos, apagados e acessados em qualquer ordem. Além disso, a operação de recuperação não é destrutiva. Embora sejam fáceis de visualizar, as árvores apresentam difíceis problemas de programação, que serão apenas introduzidos nesta seção.

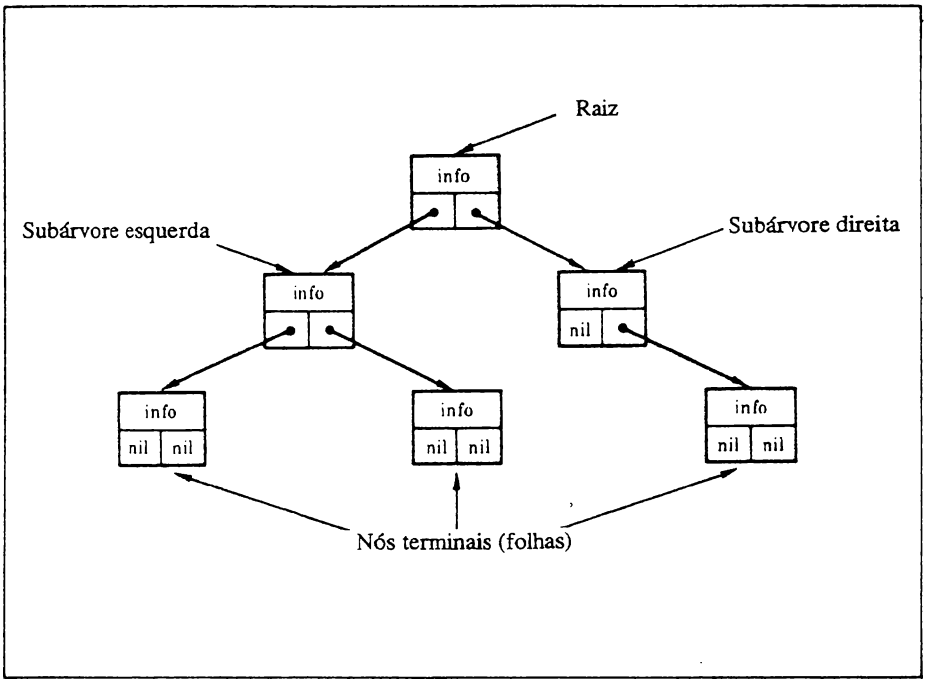
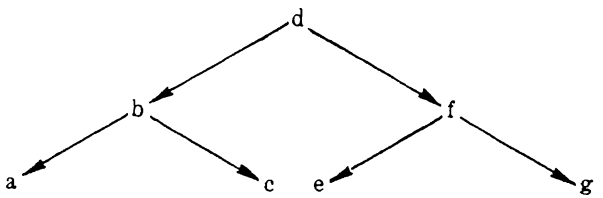


Figura 3-11 Um exemplo de árvore binária.

A maioria das funções que operam com árvores é recursiva, pois a própria árvore é uma estrutura de dados recursiva; ou seja, cada subárvore é uma árvore. Portanto, as rotinas que são desenvolvidas aqui também são recursivas. Versões não-recursivas destas funções também existem, mas são muito mais difíceis de entender.

A ordem de uma árvore depende de como essa árvore será acessada. O processo de acessar cada **nó** em uma árvore é chamado de varredura da árvore *(tree traversal)*. Aqui está um exemplo:



Há três modos de percorrer uma árvore chamados, em inglês: *inorder*, *preorder* e *postorder*. Com *inorder*, você visita a subárvore da esquerda, visita a raiz, e então visita a subárvore da direita. Em *preorder* você visita a raiz, a subárvore da esquerda e em seguida a subárvore da direita. Com *postorder*, você visita a subárvore da esquerda, a subárvore da direita e a raiz. A ordem de acesso para os três processos mostrados, usando cada método, é a seguinte:

inorder	a b c d e f g
preorder	d b a c f e g
postorder	a c b e g f d

Embora uma árvore nem sempre precise ser ordenada, a maior parte das aplicações exige isso. O que constitui uma árvore ordenada depende de como você a percorrerá. Os exemplos do restante deste capítulo acessam a árvore *inorder*. Em uma árvore binária ordenada, a subárvore da esquerda contém nós que são menores ou iguais à raiz, enquanto aquelas à direita são maiores que a raiz. A seguinte função chamada **Armazena_arv** constrói uma árvore binária ordenada.

```
type
   ptr_arvore = ^arvore;
   arvore = record
        dado: char;
        esquerdo: ptr_arvore;
        direito: ptr_arvore;
   end;

function Armazena_arv(raiz,r: ptr_arvore; dado: char): ptr_arvore;
begin
   if r=nil then
   begin
      New(r); {prepara novo no'}
      r^.esquerdo:=nil;
      r^.direito:=nil;
      r^.dado:=dado;
      if raiz <> nil then
         if dado < raiz^.dado then raiz^.esquerdo:=r
         else raiz^.direito:=r;
      Armazena_arv:=r;
   end else
   begin
      if dado < r^.dado then
         Armazena_arv:=Armazena_arv(r,r^.esquerdo,dado)
      else Armazena_arv:=Armazena_arv(r,r^.direito,dado);
   end;
end;  {Armazena_arv}
```

Este algoritmo simplesmente segue as ligações pela árvore, seguindo à direita ou à esquerda, baseado no campo **dado**. Para usar esta função você precisa de uma variável global que contenha a raiz da árvore. Essa variável global deve ser inicializada em **nil**. Um *pointer* para a raiz é atribuído na primeira chamada de **Armazena_arv**. Uma vez que

chamadas subseqüentes não precisarão reatribuir a raiz, a variável **dummy** é usada. Se você admitir que o nome desta global é **rz**, então para chamar a função **Armazena_arv** você usará

```
{ chama Armazena_arv }

if rz = nil then rz := Armazena_arv(rz,rz,item)
   else dummy := Armazena_arv(rz,rz,item)
```

O uso desta chamada permite que sejam inseridos corretamente tanto o primeiro elemento como os subseqüentes.

Armazena_arv é um algoritmo recursivo, assim como a maioria das rotinas para árvores. A mesma rotina seria várias vezes mais longa se fossem usados métodos iterativos comuns. A função deve ser chamada com um *pointer* para a raiz e para a subárvore, e com a informação de que deve ser armazenada. Embora, por simplicidade, aqui tenha sido usado como informação um único caractere, você pode substituí-lo por qualquer tipo de dados que deseje.

Para percorrer a árvore construída, usando **Armazena_arv** pelo método *inorder* e imprimindo o campo dado de cada nó, você poderia usar a função **InOrder:**

```
procedure InOrder(raiz: Ptr_Arvore);
begin
   if raiz <> nil then
   begin
      InOrder(raiz^.esquerdo);
      Write(raiz^.dado);
      InOrder(raiz^.direito);
   end;
end;  {InOrder}
```

Esta função recursiva retorna quando encontra um nó terminal (um *pointer* **nil**). As funções para percorrer a árvore em *preorder* e *postorder* são mostradas aqui:

```
procedure PreOrder(raiz: Ptr_Arvore);
begin
   if raiz <> nil then
   begin
     Write(raiz^.dado);
     preorder(raiz^.esquerdo);
     preorder(raiz^.direito);
  end;
end;  {PreOrder}

procedure PostOrder(raiz: Ptr_Arvore);
begin
   if raiz <> nil then
   begin
```

```
      postorder(raiz^.esquerdo);
      postorder(raiz^.direito);
      Write(raiz^.dado);
   end;
end;  {PostOrder}
```

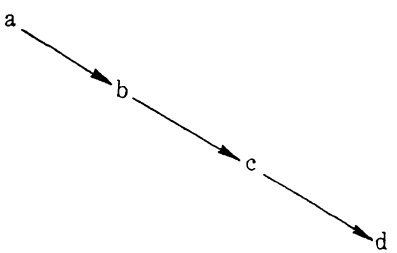
Você pode escrever um programa curto que construa uma árvore binária ordenada e imprima aqueles três caminhos lateralmente na tela do seu computador. Você precisa apenas de uma pequena modificação para o procedimento **Inorder.** O novo programa, chamado **Desenha_Arvore** imprime uma árvore *inorder*.

```
program Desenha_Arvore;

type

   Ptr_Arvore = ^arvore;
   arvore = record
        dado: char;
        esquerdo: Ptr_Arvore;
        direito: Ptr_Arvore;
   end;

var
   raiz,fantasma: Ptr_Arvore;
   car: char;

function Armazena_Arv(raiz,r: Ptr_Arvore; dado: char): Ptr_Arvore;
begin
   if r=nil then
   begin
      New(r); {prepara um novo no'}
      r^.esquerdo:=nil;
      r^.direito:=nil;
      r^.dado:=dado;
      if raiz <> nil then
         if dado < raiz^.dado then raiz^.esquerdo:=r
         else raiz^.direito:=r;
      Armazena_Arv:=r;
   end else
   begin
      if dado < r^.dado then Armazena_Arv:=Armazena_Arv(r,r^.esquerdo,dado)
      else Armazena_Arv:=Armazena_Arv(r,r^.direito,dado);
   end;
end;  {Armazena_Arv}

procedure Desenha_Arv(r: Ptr_Arvore; n: integer);
var
   i: integer;

begin
   if r <> nil then
   begin
      Desenha_Arv(r^.esquerdo,n+1);
      for i:=1 to n do Write('     ');
      WriteLn(r^.dado);
      Desenha_Arv(r^.direito,n+1);
```

```
   end;
end;  {Desenha_Arv}

begin {principal}
   raiz:=nil;
   repeat
      Write('Digite uma letra (F = Fim): ');
      Read(car); WriteLn;
      if raiz=nil then raiz:=Armazena_Arv(raiz,raiz,car)
      else fantasma:=Armazena_Arv(raiz,raiz,car);
      car:=UpCase(car);
   until car='F';

   Desenha_Arv(raiz,0);

end.
```

O programa realmente ordena os dados que você fornecer a ele. Esse processo é uma variante do ordenador por inserção que foi dado no capítulo anterior. Para o caso médio, a performance deste ordenador pode ser bastante boa, mas o *QuickSort* é um método de ordenação para uso geral ainda melhor, pois utiliza menos memória e tem menos trabalho de processamento. Entretanto, se você tiver de construir uma árvore partindo do nada ou se tiver de manter uma árvore já ordenada, deve sempre introduzir novos dados já ordenados usando a função **Armazena_arv.**

Se você já rodou o programa **Desenha_Arvore**, provavelmente deve ter notado que algumas árvores são *equilibradas* – cada subárvore é da mesma ou quase da mesma altura que qualquer outra – enquanto outras árvores são muito desequilibradas. Se você entrasse com a árvore **abcd,** sua construção seria como segue:



Não haveria subárvores à esquerda. Isso é chamado de uma *árvore degenerada,* pois degenerou em uma lista linear. Em geral, se os dados que você usar para construir uma árvore binária forem bem aleatórios, a árvore produzida se aproximará de uma árvore equilibrada. Entretanto, se a informação usada já tiver sido ordenada, aparecerá uma árvore degenerada. (É possível reajustar a árvore a cada inserção de modo a manter a árvore equilibrada. Os algoritmos para fazer isso são muito complexos; se você estiver interessado neles, consulte livros de algoritmos de programação avançada.)

Funções de pesquisa são fáceis de implementar para árvores binárias. Esta função retorna um *pointer* ao nó na árvore que corresponde à chave; caso contrário, ela retornará um **nil:**

```
function Busca(raiz: Ptr_Cel; chave: str9): Ptr_Cel;

begin
   if raiz <> nil then
   begin
      while (raiz^.Nome_Cel <> chave) and (raiz <> nil) do
      begin
         if raiz^.Nome_Cel < chave then raiz:=^.esquerdo
         else raiz:=raiz^.direito;
      end;
   end;
   Busca:=raiz;
end;  {Busca}
```

Infelizmente, apagar um nó de uma árvore não é tão simples como pesquisar a árvore. O nó apagado pode ser tanto a raiz, um nó esquerdo, ou um nó direito. O nó pode ter também de zero a dois subnós ligados a ele. O rearranjo dos *pointers* presta-se a um algoritmo recursivo, como mostrado aqui:

```
function Apaga_Arv(raiz: Ptr_Arvore; chave: char): Ptr_Arvore;
var
   temp,temp2: Ptr_Arvore;
begin
   if raiz^.dado=chave then
   begin {apaga a raiz}
      if raiz^.esquerdo=raiz^.direito then
      begin {arvore vazia}
         Dispose(raiz);
         Apaga_Arv:=nil;
      end
      else if raiz^.esquerdo=nil then
      begin
         temp:=raiz^.direito;
         Dispose(raiz);
         Apaga_Arv:=temp;
      end
      else if raiz^.direito=nil then
      begin
         temp:=raiz^.esquerdo;
         Dispose(raiz);
         Apaga_Arv:=temp;
      end
      else begin {ambos os ramos presentes}
         temp2:=raiz^.direito;
         temp:=raiz^.esquerdo;
         while temp^.esquerdo <> nil do temp:=temp^.esquerdo;
         temp^.esquerdo:=raiz^.esquerdo;
         Dispose(raiz);
         Apaga_Arv:=temp2;
      end;
```

```
    end
    else begin
       if raiz^.dado < chave then raiz^.direito:=Apaga_Arv(raiz^.direito,chave)
       else raiz^.esquerdo:=Apaga_Arv(raiz^.esquerdo,chave);
       Apaga_Arv:=raiz;
    end;

end;  {Apaga_Arv}
```

Lembre-se de atualizar o *pointer* da raiz no resto do seu programa, pois o nó apagado pode ser a raiz da árvore.

Quando usado com programas de gerenciamento de bancos de dados, as árvores binárias oferecem poder, flexibilidade e eficiência. A informação para esses bancos de dados deve estar em disco, e os tempos de acesso são importantes. Por ter $\log_2 n$ comparações a executar, no pior caso, uma árvore binária equilibrada funciona muito melhor que uma lista encadeada, que depende de uma busca seqüencial.

CAPÍTULO 4

ALOCAÇÃO DINÂMICA

Escrever um programa pode ser como construir um prédio, com diversas considerações de ordem tanto estética como funcional, influindo no aspecto final. Assim, alguns programas são tão funcionalmente rígidos quanto uma casa, que tem um certo número de quartos, uma cozinha, dois banheiros etc. Outros têm a arquitetura aberta de centro de convenções, com paredes móveis e compartimentos modulares, permitindo sua adaptação a diferentes finalidades. Este capítulo descreve métodos de armazenamento que facilitam a confecção de programas flexíveis, adaptáveis às necessidades do usuário e à capacidade de cada máquina.

Este capítulo usa os termos *matriz lógica* e *matriz física*. A matriz lógica é a que existe apenas virtualmente no computador. Uma planilha eletrônica é uma matriz lógica. A matriz física é a que existe de fato. São as rotinas de apoio da matriz dispersa que tornam estas duas matrizes a mesma. Veremos quatro técnicas diferentes para se criar uma matriz dispersa: a lista encadeada, a árvore binária, uma matriz de ponteiros e *hashing*. Serão também apresentados exemplos dos modos pelos quais a alocação dinâmica pode ser usada para melhorar o desempenho de um programa.

O Pascal pode armazenar informações na memória do computador, de duas maneiras diversas. A primeira é pelo uso de *variáveis globais* e *locais*, inclusive matrizes e registros, definidas na linguagem Pascal. As variáveis globais têm armazenamento fixo durante a execução do programa. Para as variáveis locais é usado o espaço do *stack*. O único inconveniente apresentado por este método é a necessidade de conhecimento prévio da quantidade de memória a ser usada em qualquer caso possível. O segundo modo de

armazenamento, mais eficiente, faz uso das funções de alocação dinâmica do Pascal: **New**, associado ao par **Mark/Release** ou a **Dispose.**

Na alocação dinâmica, é usada a área livre da memória, localizada entre a área ocupada permanentemente pelo programa e o *stack* (usado para o armazenamento de variáveis locais). Esta área é chamada *heap*.

A Figura 4-1 mostra o esquema de um programa em Pascal, ocupando a memória. O stack cresce de cima para baixo. A quantidade de memória que ele usa depende da estrutura do programa a ser executado. Um programa com muitas funções recursivas requer mais memória para o *stack* que um programa sem função recursiva alguma, isto porque cada chamada recursiva usa o *stack*. A memória necessária para conter o corpo do programa e as informações globais é fixa durante a execução. A memória requerida por um **New** é tomada da área de memória livre, começando acima das variáveis globais e crescendo em direção ao *stack*. Em casos extremos, pode ocorrer uma colisão entre o *stack* e o *heap*.

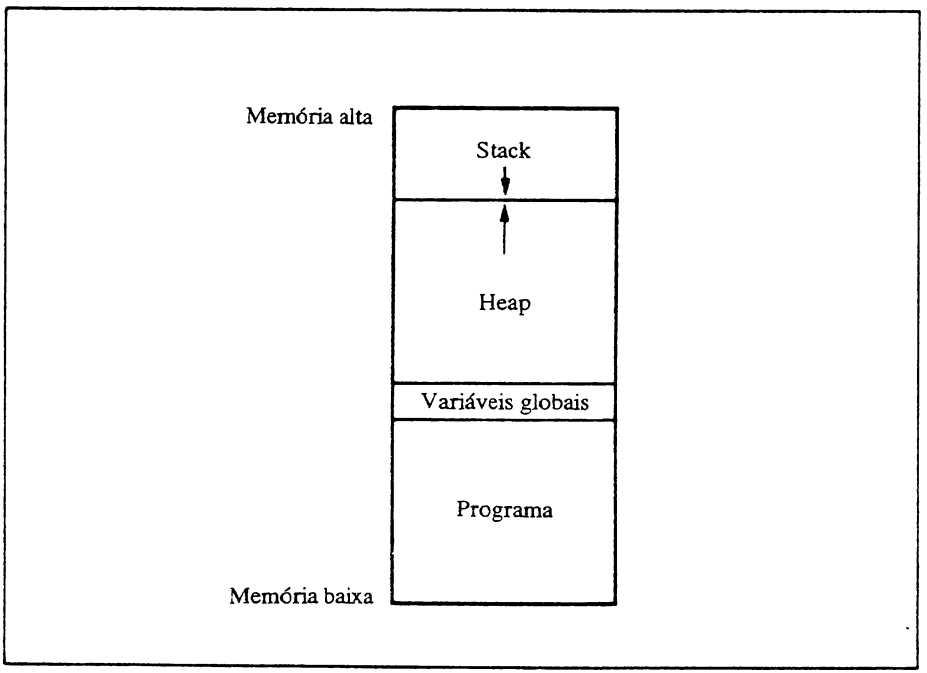


Figura 4-1 Uso da memória por um programa em Pascal.

O *heap* é controlado através dos procedimentos **Dispose** ou **Mark/Release.** Como diz o *Manual do Turbo Pascal,* **Dispose** e **Mark/Release** nunca podem ser usados juntos, num mesmo programa. Você deve, então, decidir de início qual dos dois procedimentos será usado. Para ajudá-lo nessa decisão, segue uma rápida revisão de **New, Dispose** e **Mark/Release.**

NEW

New aloca memória do *heap*. Este procedimento predefinido tem como argumento um ponteiro, e aloca, no *heap,* memória suficiente para conter o tipo de variável à qual o ponteiro em questão se refere. Após a chamada de **New,** o ponteiro estará indicando a memória alocada. Por exemplo, para alocar espaço no *heap* para um **real,** você poderia fazer assim:

```
type
  ptr_real=^ real;
var
  p:ptr_real;
begin
  New(p);
    .
    .
    .
```

Se não houver memória livre no *heap,* o erro de tempo de execução FF *(heap/stack collision)* ocorrerá. Isto pode ser evitado fazendo com que cada **New** seja precedido de uma chamada de **maxavail,** a qual retorna o número de bytes (nos sistemas de 8 bits) ou parágrafos (nos sistemas de 16 bits) disponíveis no *heap*. Os exemplos deste capítulo não incluem este passo, mas podem ser úteis em programas reais.

DISPOSE

Uma das principais razões em favor do uso da alocação dinâmica da memória é a possibilidade, oferecida por este método, de reutilização da memória. **Dispose** é um dos modos pelos quais memória pode ser liberada no *heap*. **Dispose** tem por argumento um ponteiro usado previamente numa chamada de **New.** Assim, **Dispose** contém um ponteiro que indica uma região válida de alocação do *heap*. Após a execução de um **Dispose,** a região

que havia sido alocada para aquele ponteiro está livre para nova alocação. No exemplo a seguir, memória é alocada para um ponteiro referente a uma matriz de 40 elementos e, após o uso, liberada:

```
program Amostra; {Exemplo de New e Dispose juntos}

type
   pont= ^Registro;
   Registro= array[1..40] of integer;

var
   p: pont;
   t: integer;

begin
   New(p);
   for t:=1 to 40 do p^[t]:=t*2;
   for t:=1 to 40 do Write(p^[t],' ');
   WriteLn;
   Dispose(p);
end.
```

MARK E RELEASE

Mark e **Release** também podem ser usados para liberar memória do *heap* após seu uso pelo programa. Resumindo, **Mark** é chamado antes do uso de um **New.** Quando for o momento de liberar a memória, uma chamada a **Release** liberará toda a memória alocada desde o **Mark.** Este método devolve ao *heap* blocos de memória, enquanto o **Dispose** libera apenas a memória usada por um único ponteiro.

Mark tem como argumento um ponteiro de qualquer tipo, cuja única função é marcar o ponto inicial de uma região no *heap*. **Release** deve ser chamado com o mesmo ponteiro, que não pode ser modificado. O programa seguinte aloca memória para uma matriz de 40 elementos e libera esta área, usando **Mark** e **Release.**

```
program Aloca; {esta versao usa Mark e Release}

type
   pont= ^Registro;
   Registro= array[1..40] of integer;

var
   p: pont;
   t: integer;
   q: ^integer;
```

```
begin
   Mark(q);
   New(p);
   for t:=1 to 40 do p^[t]:=t*2;
   for t:=1 to 40 do Write(p^[t],' ');
   WriteLn;
   Release(q);
   {neste momento toda a memoria foi devolvida ao sistema}
end.
```

A escolha do método a ser usado dependerá sempre do programa em questão. Se apenas uma parte da memória alocada no *heap* deve ser liberada, então **Dispose** é o melhor método. Se toda a memória alocada tiver de ser liberada, então **Mark** e **Release** devem ser usados. Os exemplos deste capítulo usam **Dispose**, pois ele oferece maior flexibilidade. Entretanto, sinta-se livre para usar **Mark** e **Release**, sempre que achar melhor.

PROCESSAMENTO DE MATRIZES DISPERSAS

Um dos principais usos da alocação dinâmica é no *processamento de matrizes dispersas*. Numa matriz dispersa, nem todos os elementos potenciais da matriz estão presentes na memória da máquina ao mesmo tempo. A matriz dispersa é usada quando a matriz requerida por um dado programa ultrapassa, por seu tamanho, a capacidade de memória da máquina, e quando nem todos os elementos da matriz serão usados ao mesmo tempo. Matrizes podem ocupar grandes áreas da memória, pois o espaço por elas ocupado cresce polinomialmente em relação a suas dimensões. Por exemplo, enquanto uma matriz de 10 x 10 ocupa apenas 100 bytes, e uma matriz de 100 x 100 apenas 10.000 bytes, uma matriz de 1.000 x 1.000 ocupará 1.000.000 de bytes de memória.

Uma planilha de cálculo é um ótimo exemplo de matriz dispersa. Mesmo que a matriz pareça grande, 999 x 999 digamos, apenas uma parte dela poderá ser usada a cada momento. Nas planilhas, a matriz contém fórmulas, valores e seqüências de caracteres associados a cada endereço. Numa matriz dispersa, o espaço para cada elemento é alocado do *heap*, sempre que necessário. Apesar de apenas uma pequena parte dos elementos poder ser usada ao mesmo tempo, a matriz parecerá grande, maior do que caberia na memória do computador.

Existem três técnicas distintas para se criar uma matriz dispersa: uma lista encadeada, uma árvore binária e uma matriz de ponteiros. Os exemplos assumem que a planilha está organizada do seguinte modo:

```
----A---- ----B---- ----C----...
1
2                X
3
4
5
6
7
```

Neste exemplo, X está na célula B2.

O USO DE LISTAS ENCADEADAS NA CRIAÇÃO DE MATRIZES DISPERSAS

Numa matriz dispersa, criada por meio de uma lista encadeada, um registro é usado para manter informações sobre cada elemento da matriz, informações estas que incluem a posição lógica do elemento na matriz e as ligações com o elemento anterior e o próximo elemento. Cada registro é colocado na lista, que é ordenada de acordo com o índice da matriz. Para acessar a matriz, basta seguir as ligações entre os elementos.

O seguinte registro poderia ser usado na criação de uma matriz dispersa para uma planilha eletrônica:

```
type

   str128= string[128];
   str9= string[9];

   Ptr_Cel= ^cel;

   cel= record
           nome_cel: str9;  {contem o nome da celula}
           formula: str128;
           proximo: Ptr_Cel; {aponta para o proximo registro}
           anterior: Ptr_Cel; {aponta para o registro anterior}
        end;
```

Neste exemplo, o campo **Nome_Cel** contém a seqüência de caracteres que indica o nome da célula, como A1, B34 ou Z19. A seqüência **formula** mantém a **formula** contida em cada posição da planilha. A seguir, serão apresentadas algumas funções que poderiam ser usadas numa planilha eletrônica, formada por uma matriz dispersa de lista encadeada.

(Existem muitas maneiras diferentes de fazer planilha eletrônica; os registros de dados e rotinas usadas aqui são apenas exemplos de técnicas de manipulação de matrizes dispersas.) As seguintes variáveis globais indicam o início e o fim da lista encadeada da matriz:

```
primeiro,ultimo: Ptr_Cel;
```

Quando você digita uma fórmula numa célula de uma planilha eletrônica qualquer, está, na realidade, criando um novo elemento na matriz dispersa. Se a planilha usar uma lista encadeada, então a nova célula será inserida na lista pela rotina **Armazena_EDO**, desenvolvida no capítulo anterior. (Como as rotinas em Pascal são entidades discretas, reutilizáveis, quase nenhuma mudança é necessária.) Aqui, a lista é ordenada pelo nome da célula, isto é, A12 vem antes de A13.

```
function Armazena_EDO(item, primeiro: Ptr_Cel;
                      var ultimo: Ptr_Cel): Ptr_Cel;
{armazena dados em lista encadeada dupla ordenada;
 retorna um ponteiro para o inicio da lista}

var
   velho, topo: Ptr_Cel;
   feito: boolean;
begin
   topo:=primeiro;
   velho:=nil;
   feito:=FALSE;

   if primeiro=nil then
   begin {primeiro elemento da lista}
      item^.proximo:= nil;
      ultimo:=item;
      item^.anterior:=nil;
      Armazena_EDO:=item;
   end else
   begin
      while (primeiro <> nil) and (not feito) do begin
         if primeiro^.nome < item^.nome then begin
            velho:=primeiro;
            primeiro:=primeiro^.proximo;
         end else begin  {para o meio}
            if velho <> nil then begin
                  velho^.proximo:=item;
                  item^.proximo:=primeiro;
                  primeiro^.anterior:=item;
                  item^.anterior:=velho;
                  Armazena_EDO:=topo;  {mantem o mesmo inicio}
                  feito:=TRUE;
            end else begin
                  item^.proximo:=primeiro;  {novo primeiro elemento}
                  item^.anterior:=nil;
                  Armazena_EDO:= item;
                  feito:=TRUE;
            end;
```

```
        end;
    end;  {while}
    if not feito then
    begin
       ultimo^.proximo:=item;   {para o fim}
       item^.proximo:=nil;
       item^.anterior:=ultimo;
       ultimo:=item;
       Armazena_EDO:=topo;
    end;
  end;
end;  {Armazena_EDO}
```

Para remover uma célula da planilha, você deve remover o registro correspondente da lista e liberar a área da memória, antes ocupada por aquela célula com o uso de **Dispose**. A função **Apaga_DE** remove uma célula da lista, dado o nome da célula.

```
function Apaga_DE(primeiro: Ptr_Cel; chave: str9): Ptr_Cel;

var
   temp, temp2: Ptr_Cel;
   feito: boolean;
begin
   if primeiro^.nome_cel=chave then begin  {apaga o primeiro da lista}
      Apaga_DE:=primeiro^.proximo;
      if temp^.proximo <> nil then begin
         temp:=primeiro^.proximo;
         temp^.anterior:=nil;
      end;
      Dispose(primeiro);
   end else begin
      feito:=FALSE;
      temp:=primeiro^.proximo;
      temp2:=primeiro;
      while (temp <> nil) and (not feito) do
      begin
         if temp^.nome_cel=chave then
         begin
            temp2^.proximo:=temp^.proximo;
            if temp^.proximo <> nil then
               temp^.proximo^.anterior:=temp2;
            feito:=TRUE;
            ultimo:=temp^.anterior;
            Dispose(temp);
         end else
         begin
            temp2:=temp;
            temp:=temp^.proximo;
         end;
      end;
      Apaga_DE:=primeiro; {ainda o mesmo inicio}
      if not feito then WriteLn('Nao encontrada.');
   end;
end;  {Apaga_DE}
```

A função **Acha** localiza qualquer célula específica. Esta função é importante, pois muitas das fórmulas da planilha podem fazer referência a outras células; tais células devem ser encontradas e seu conteúdo atualizado. **Acha** realiza uma busca linear para cada item. Como vimos no Capítulo 3, o número médio de comparações numa busca linear é $n/2$, onde n é o número de elementos da lista. Além disto, uma perda de eficiência ainda maior ocorre, pois cada célula pode conter referências a outras células da fórmula, que também deverão ser encontrados. Aqui está um exemplo de **Acha:**

```
function Acha(celula: Ptr_Cel): Ptr_Cel;
var
   c: Ptr_Cel;
begin
   c:=primeiro;
   while c <> nil do begin
      if c^.Nome_Cel=celula^.Nome_Cel then acha:=c
      else c:=c^.proximo;
   end;
   WriteLn('Celula nao encontrada.');
   Find:=nil;
end; {Find}
```

O processo de criação, manutenção e processamento de matrizes dispersas através de listas encadeadas possui esta grande desvantagem: uma busca linear é necessária para acessar cada célula da lista. Para possibilitar uma busca binária, precisaria ser acrescentada mais informação, aumentando o já crítico problema de falta de espaço na memória. Mesmo a rotina de armazenamento usa uma busca linear para encontrar a posição correta, onde a nova célula deve ser inserida. Estes problemas podem ser resolvidos pelo uso de uma árvore binária no lugar da lista encadeada.

O USO DE ÁRVORES BINÁRIAS NA CRIAÇÃO DE MATRIZES DISPERSAS

Uma árvore binária é basicamente uma lista duplamente encadeada. Sua maior vantagem em relação à lista está no aumento da velocidade das buscas, tornando muito mais rápidas as inserções e verificações. Se você quiser usar um registro de lista encadeada mas precisar também de buscas velozes, então o que procura é uma árvore binária.

Para usar uma árvore binária no exemplo da planilha eletrônica, o registro célula deve ser modificado, do seguinte modo:

```
Ptr_Cel= ^celula;
str9= string[9];
str128= string[128];
```

```
celula= record
    Nome_Cel: str9;
    formula: str128;
    esquerdo: Ptr_Cel;
    direito: Ptr_Cel;
end;
```

Você pode modificar a função **Armazena_arv**, vista no Capítulo 3, para que ela construa uma árvore baseada nos nomes das células. A função assume que o parâmetro **New** é um ponteiro para uma nova entrada na árvore.

```
function Armazena_arv(raiz,r,New: Ptr_Cel): Ptr_Cel;
begin
   if r=nil then
   begin
      Novo^.esquerdo:=nil;
      Novo^.direito:=nil;
      if Novo^.Nome_Cel < raiz^.Nome_Cel then raiz^.esquerdo:=Novo
      else raiz^.direito:=Novo;
      Armazena_arv:=Novo;
   end else
   begin
      if Novo^.Nome_Cel < r^.Nome_Cel then
         Armazena_arv:=Armazena_arv(r,r^.esquerdo,dado)
      else Armazena_arv:=Armazena_arv(r,r^.direito,dado);
   end;
end;  {Armazena_arv}
```

Os dois primeiros parâmetros de **Armazena_arv** servem de ponteiro para o nó **raiz**, e o terceiro parâmetro como ponteiro para a nova célula. **Armazena_arv** produz um ponteiro indicando a raiz.

Para apagar uma célula da planilha, você deve modificar a função **Apaga_Arv**, para que esta passe a aceitar o nome da célula como chave:

```
function Apaga_Arv(raiz: Ptr_Cel; chave: str9): Ptr_Cel;
var
   temp,temp2: Ptr_Cel;
begin
   if raiz^.Nome_Cel=chave then
   begin {apaga a raiz}
      if raiz^.esquerdo=raiz^.direito then
      begin {arvore vazia}
         Dispose(raiz);
         Apaga_Arv:=nil;
      end
      else if raiz^.esquerdo=nil then
      begin
         temp:=raiz^.direito;
         Dispose(raiz);
         Apaga_Arv:=temp;
      end
      else if raiz^.direito=nil then
```

```
      begin
         temp:=raiz^.esquerdo;
         Dispose(raiz);
         Apaga_Arv:=temp;
      end
      else begin {ambos os ramos presentes}
         temp2:=raiz^.direito;
         temp:=raiz^.esquerdo;
         while temp^.esquerdo <> nil do temp:=temp^.esquerdo;
         temp^.esquerdo:=raiz^.esquerdo;
         Dispose(raiz);
         Apaga_Arv:=temp2;
      end;
   end
   else begin
      if raiz^.Nome_Cel < chave then
         raiz^.direito:=Apaga_Arv(raiz^.direito,chave)
      else raiz^.esquerdo:=Apaga_Arv(raiz^.esquerdo,chave);
      Apaga_Arv:=raiz;
   end;

end;  {Apaga_Arv}
```

Por fim, a função **Busca** modificada pode ser usada para localizar rapidamente qualquer célula da planilha:

```
function Busca(raiz: Ptr_Cel; chave: str9): Ptr_Cel;

begin
   if raiz <> nil then
   begin
      while (raiz^.Nome_Cel <> chave) and (raiz <> nil) do
      begin
         if raiz^.Nome_Cel < chave then raiz:=^.esquerdo
         else raiz:=raiz^.direito;
      end;
   end;
   Busca:=raiz;
end;  {Busca}
```

A grande vantagem da árvore binária sobre a lista encadeada, como já foi dito, é o grande aumento na velocidade das buscas. A busca seqüencial requer, em seu caso médio, $n/2$ comparações, onde n é o número de elementos da lista; a busca binária requer apenas $\log_2 n$ comparações.

O USO DE MATRIZES DE PONTEIROS NA CRIAÇÃO DE MATRIZES DISPERSAS

Suponha uma planilha eletrônica, cujas dimensões fossem 26 x 100 (A1 a Z100), num total

de 2.600 elementos. Então, em teoria, a seguinte matriz de registros poderia ser usada para conter as entradas da planilha:

```
str9= string[9];
str128= string[128];


Ptr_Cel= ^celula;
celula= record
    Nome_Cel: str9;
    formula: str128;
end;
var
   mp: array[1..2600] of celula;
```

Entretanto, 2.600 células multiplicadas por 128 (o tamanho do campo **formula**) vão requerer 332.800 bytes de memória, e isto numa planilha bastante pequena. Obviamente, isto não é nada prático. Uma alternativa seria criar uma matriz de ponteiros de registros. Este método requer muito menos armazenamento permanente que a criação de uma matriz inteira, e oferece uma performance muito superior aos dois outros métodos vistos até aqui. A declaração necessária para esta técnica é a seguinte:

```
const
   TAMANHO= 2600;

type
   str9= string[9];
   str128= string[128];

   celula= record
      Nome_Cel: str9;
      formula: str128;
   end;

   Ptr_Cel= ^celula;

var
   mp: array[1..TAMANHO] of Ptr_Cel;
```

Esta matriz menor será usada para conter ponteiros que indiquem os dados digitados na planilha pelo usuário. A cada nova entrada de dados, um ponteiro, indicando para a informação da célula, é armazenado na matriz. A Figura 4-2 mostra qual seria a aparência deste processo na memória, com a matriz de ponteiros servindo de alicerce para a matriz dispersa.

Antes de usar a matriz de ponteiros, você deve inicializar cada um de seus elementos para **nil**, o qual indica que não há célula preenchida naquela posição. O procedimento a seguir faz isto:

```
procedure InicMat;
var
   t: integer;

begin
   for t=1 to TAMANHO do mp[t]:=nil;
end;  {InicMat}
```

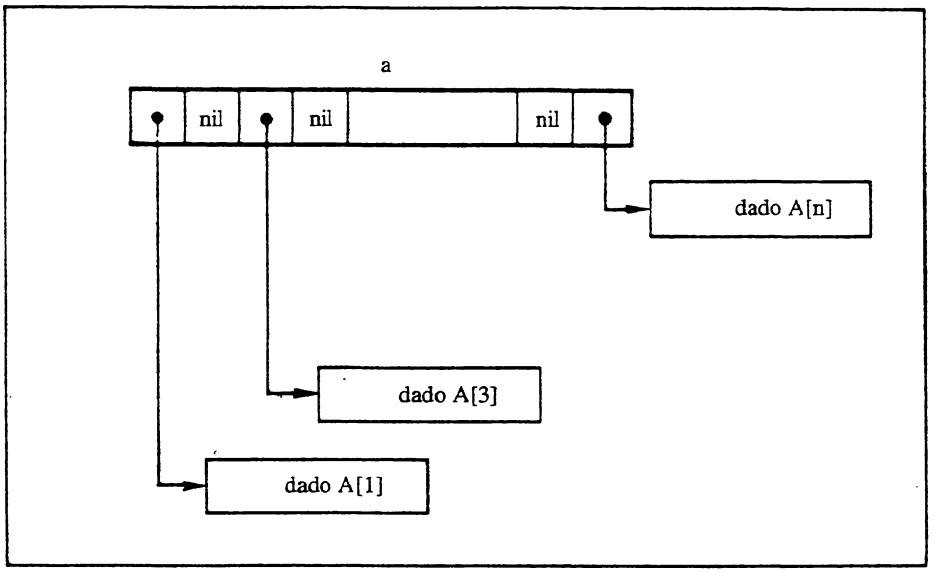


Figura 4-2 Uma matriz de ponteiros servindo de alicerce para a matriz dispersa.

Antes de poder escrever o procedimento **Armazena**, você precisará de uma função chamada **Acha_Ind**, que retorna o índice da matriz de ponteiros, dado o nome da célula. **Acha_Ind** assume que o nome da célula é formado por uma letra maiúscula, seguida de um número inteiro (B34, C19 etc.). **Acha_Ind** é mostrada abaixo:

```
function Acha_Ind(i: Ptr_Cel): integer;
var
   loc,temp,codigo: integer;
   t: str9;

begin
   loc:=ord(i^.Nome_Cel[1])-ord('A');
   t:=copy(i^.Nome_Cel,2,9);
   val(t,temp,codigo);
   Acha_Ind:=loc+(temp*26);
end;  {Acha_Ind}
```

Esta função informa ao procedimento **Armazena** qual endereço da matriz de ponteiros deve ser usado para cada célula. Como você pode notar, o cálculo do índice é simples e rápido, sem necessitar de buscas ou verificações. Este processo é, às vezes, chamado de *hash* direto, porque o dado a ser armazenado produz diretamente o índice do endereço de armazenamento. Quando um valor é digitado numa determinada célula, o nome da célula produz um índice para a matriz **mt.** Este índice é convertido num número por **Acha_Ind**, e armazenado por **Armazena:**

```
procedure Armazena(Novo: Ptr_cel);
var
   loc: integer;

begin
   loc:= Acha_Ind(Novo);
   if loc > TAMANHO then WriteLn('Fora dos limites.')
   else mp[loc]:=Novo;
end;  {Armazena}
```

Como cada nome de célula é único, cada índice é único. Como uma seqüência ASCII é usada, cada ponteiro é armazenado na posição correta da matriz. Comparado à lista encadeada, este método é muito menor e mais simples.

A função **Apaga** também fica menor. Quando chamada com um índice de célula, ela apaga o ponteiro correspondente ao elemento e libera a memória daquele ponteiro para o sistema:

```
procedure Apaga(r_celula: Ptr_Cel);
var
   loc: integer;
begin
   loc:=Acha_Ind(r_celula_);
   if loc > 10000 then WriteLn('Fora dos limites.')
   else
   begin
      Dispose(r_celula);
      mp[loc]:=nil;
   end;
end;  {Apaga}
```

Outra vez, se comparada à lista encadeada ou à árvore binária, esta rotina é muito mais rápida e simples.

Entretanto, lembre-se de que a matriz de ponteiros ocupa espaço na memória para cada posição, seja esta posição usada ou não, o que pode representar uma séria limitação para algumas aplicações.

HASHING

Hashing é o nome dado ao processo de se extrair, da própria informação a ser armazenada, o elemento correspondente da matriz índice. O índice assim gerado é chamado *hash*. *Hashing* tem sido aplicado a arquivos de disco, para reduzir o tempo de acesso. Entretanto, o mesmo método básico pode ser aplicado na criação de matrizes dispersas. No método de matrizes de ponteiros, apresentado no exemplo anterior, uma forma de *hashing* chamada *hashing direto* estava envolvida. No *hashing direto*, cada chave aponta para uma e apenas uma posição da matriz. Isto é, cada chave é transformada num índice único. (Na verdade, o método de matriz de ponteiros não requer um esquema de indexação direta; porém, no problema de planilha eletrônica este era o caminho mais óbvio.) Na prática, poucos são os esquemas de indexação direta existentes, e um método mais flexível é necessário. Nesta seção, veremos como o processo de *hashing* pode ser generalizado, tornando-se um instrumento flexível e poderoso.

A esta altura, deve estar claro que, mesmo no mais rigoroso dos ambientes, nem todas as células de uma planilha eletrônica serão usadas. Neste exemplo, assumiremos que, na imensa maioria dos casos, não mais de 10% das posições potenciais serão realmente ocupadas. Isto quer dizer que, se as dimensões da matriz lógica da planilha forem de 26 x 100 (2.600 posições), apenas 260 posições serão usadas num dado momento. Portanto, a maior matriz física necessária para conter todas as células ocupadas precisará ter apenas 260 elementos. O problema então é mapear e acessar a matriz lógica a partir da matriz física. Isto pode ser feito através de uma cadeia de *hash*.

Quando uma fórmula é digitada numa célula da planilha (a matriz lógica), a posição da célula, dada por seu nome, é usada para produzir um índice (um *hash*) na matriz física. Digamos que o nome da matriz física seja **mt**. O índice é obtido convertendo o nome da célula num número, exatamente como no exemplo da matriz de ponteiros. Este número é então dividido por 10 para produzir um ponto inicial de entrada na matriz (pois esta só tem 260 posições). Se a posição dada pelo índice estiver vazia, o índice e o valor serão armazenados ali. Caso contrário, ocorre uma colisão. Uma colisão ocorrerá quando o nome de duas células produzir o mesmo *hash*. Isto fará com que os índices produzidos por ambas aponte para o mesmo elemento da matriz física. Neste caso, um elemento vago será procurado na matriz **mt**. Uma vez encontrada uma posição vazia, a informação é armazenada ali e um ponteiro é colocado na posição original, indicando a posição real da célula. A Figura 4-3 exemplifica a situação.

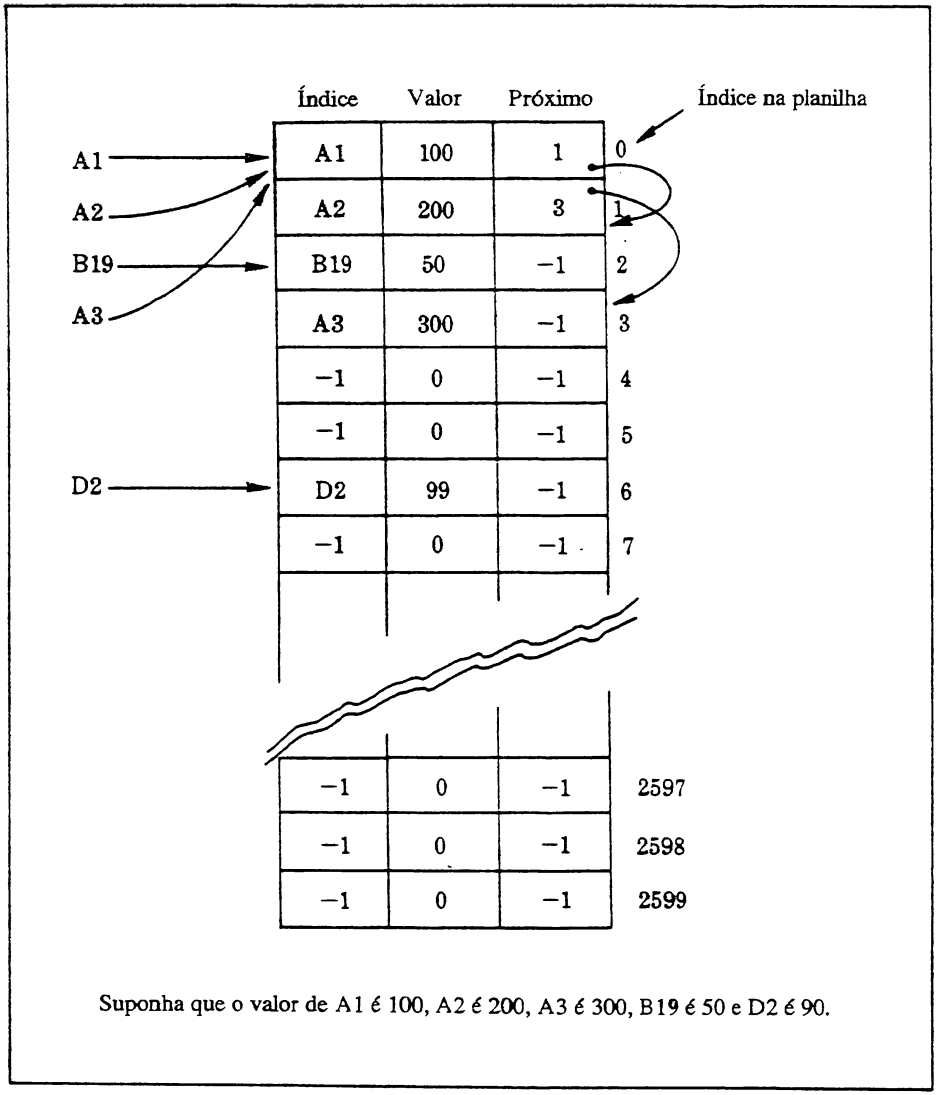


Figura 4-3 Um exemplo de *hashing*.

Para encontrar um elemento na matriz física, dado o nome da célula na matriz lógica, deve-se primeiro transformar este nome no *hash*. A posição gerada pelo *hash* é então verificada na matriz física. Se o índice da matriz lógica, ali armazenado, for o que se está procurando, a busca termina. Caso contrário, a cadeia de *hash* é seguida até o índice lógico correto ser encontrado ou o final da cadeia atingido.

A aplicação deste processo ao exemplo da planilha requer, inicialmente, a definição da seguinte matriz de registros, que funcionará como a matriz física.

```
const
   TAMANHO= 260;

type
   str9= string[9];
   str128= string[128];

   celula= record
      Nome_Cel: str9;
      formula: str128;
      proximo: integer;
   end;


var
   Mat: array[1..TAMANHO] of Ptr_Cel;
   nome: celula
```

Esta matriz deve ser previamente inicializada. O procedimento seguinte faz com que todos os campos **Nome_Cel** passem a conter a palavra "vazia" (nome o qual, por definição, nenhuma célula terá), indicando um elemento vazio. O –1 no campo **proximo** indica o fim da cadeia de *hash*.

```
procedure IniciMat;
var
   t: integer;

begin
   for t=0 to TAMANHO do
   begin
      Mat[t].Nome_Cel:= 'vazia';
      Mat[t].proximo:=-1;
   end;
end;  {IniciMat}
```

O procedimento **Armazena** chama o procedimento **Ind_Hash**, que, por sua vez, obtém o índice correto para a matriz **mt.** Se a posição indicada pelo valor remodelado estiver ocupado, a rotina procurará a primeira posição livre existente. Ela faz isto seguindo a cadeia de *hash* até o fim, e encontrando então a primeira posição livre. Quando tal posição é encontrada, o valor do elemento e sua posição na matriz lógica são ali armazenados. O índice lógico do elemento é armazenado porque será necessário quando o elemento for novamente acessado.

```
{calcula e armazena o hash }

function Ind_Hash(i: str9): integer;
var
   loc,temp,codigo: integer;
   t: str9;

begin
   loc:=ord(i[1])-ord('A');
   t:=copy(i,2,9);
   val(t,temp,codigo);
   Ind_Hash:=(loc*26+temp) div 10;
end;  {Ind_Hash}


procedure Armazena(Novo: celula);
var
  loc,i: integer;

begin
   loc:=Ind_Hash(Novo.Nome_Cel);
   if loc > TAMANHO then WriteLn('Fora dos limites.')
   else
   begin  {armazena em loc, se este estiver
           livre ou no caso de atualizacao de dado

      if ((Mat[loc].Nome_Cel= 'vazia') or
          (Mat[loc].Nome_Cel= Novo.Nome_Cel)) then
      begin
         Mat[loc].Nome_Cel:=Novo.Nome_Cel;
         Mat[loc].formula:=Novo.formula;
      end else  {acha uma entrada livre}
      begin
         {segue qualquer cadeia existente ate o fim}
         while(Mat[loc].proximo <> -1) do
            loc:=Mat[loc].proximo;
         {agora acha um endereco livre}
         i:=loc;
         while((i < TAMANHO) and (Mat[i].Nome_Cel <> 'vazia'))
            do i:=i+1;
         if(i=TAMANHO) then
         begin
            WriteLn('Nao pode ser colocado na matriz de hash.');
         end else
         begin  {armazena num endereco livre e atualiza a cadeia}
            Mat[i].Nome_Cel:=Novo.Nome_Cel;
            Mat[i].formula:=Novo.formula;
            Mat[loc].proximo:=i; {cadeia}
         end;
      end;
   end;
end;   {Armazena}
```

Achar o conteúdo de um elemento requer primeiro que o *hash* deste elemento seja calculado. Então, o índice lógico armazenado na posição da matriz física indicada pelo *hash* é comparado ao índice da matriz lógica pedido. Se forem iguais, o conteúdo

procurado foi encontrado. Caso contrário, a cadeia de *hash* é seguida até que o elemento procurado ou um –1 sejam encontrados. Um -1 indicaria que o elemento procurado não se encontra na matriz física. A função **Acha** realiza esta busca:

```
{Da a localizacao fisica da celula}
function Acha(cnome: celula): integer;
var
   loc: integer;
begin
   loc:= AchaIndex(cnome.Nome_Cel);
   while((Mat[loc].Nome_Cel <> cnome.Nome_Cel) and
         (loc <> -1)) do loc:=Mat[loc].next;
   if(loc=-1) then begin
      WriteLn('Nao encontrada');
      Acha:=-1;
   end else Acha:=loc;
Write('A celula esta em  '); WriteLn(loc);
end;  {Acha}
```

O esquema de *hashing*, apresentado aqui, é extremamente simples. Na prática usam-se métodos muito mais sofisticados. É comum, por exemplo, que *hashes* secundários e até terciários sejam calculados nos casos de colisão, antes que a cadeia de *hash* seja usada. O conceito básico, porém, será sempre o mesmo.

A ANÁLISE DO HASHING

No melhor caso do *hashing* (muito raro), cada índice físico dado pelo *hash* é único, e o tempo de acesso é próximo do tempo de acesso da indexação direta. Isto é, nenhuma cadeia de *hash* é criada e as verificações são basicamente acessos diretos. Raramente será este o caso, pois os índicês lógicos estarão distribuídos por todo o espaço lógico de indexação. No pior caso (também muito raro), o esquema de *hash* degenerará numa lista encadeada. Isto ocorrerá se todos os *hashes* tiverem valores idênticos. No caso médio, o mais comum, o tempo de acesso será igual ao tempo de acesso da indexação direta multiplicado por uma constante qualquer que seja proporcional ao tamanho médio das cadeias de *hash*. A vantagem do uso do *hashing* na manutenção de uma matriz dispersa é o fato do algoritmo de *hashing* impedir a formação de cadeias de *hash* muito longas. O *hashing* pode ser melhor explorado quando o número de posições da matriz a serem usadas for previamente conhecido.

DECIDINDO SOBRE O MÉTODO A SER UTILIZADO

A decisão sobre o método (lista encadeada, árvore binária, matriz de ponteiros ou *hashing*)

de criação de uma matriz dispersa a ser usado deve levar em consideração a eficiência no uso da memória e a velocidade.

Quando a matriz for muito dispersa, os melhores métodos serão a lista encadeada e a árvore binária, pois eles só alocam memória para elementos em uso. Os encadeamentos requerem uma pequena quantidade de memória para serem mantidos, em geral desprezível. O método da matriz de ponteiros implica a existência de um ponteiro alocado para cada elemento da matriz lógica, esteja o elemento sendo usado ou não. Ou seja, não só toda a matriz de ponteiros deve estar armazenada na memória como também deve haver espaço o bastante para uso do programa. Esta pode ser uma séria restrição para alguns programas. Calculando a memória livre restante, você poderá decidir se este método pode ser usado por seu programa ou não.

Em termos de uso de memória, o *hashing* se encontra entre os métodos de lista e o de matriz de ponteiros. Apesar de exigir a presença de toda a matriz física todo o tempo, esta matriz física será sempre menor que a matriz de ponteiros, que requer um ponteiro para cada posição da matriz lógica.

Se a matriz lógica estiver bastante tomada, entretanto, a matriz de ponteiros fará um uso melhor da memória. Tanto a lista encadeada como a árvore binária usam dois ponteiros, enquanto a matriz de ponteiros usa apenas um. Por exemplo: se uma matriz lógica de 1.000 elementos estiver completa, e se cada ponteiro ocupar 2 bytes na memória, tanto a lista encadeada como a árvore binária usarão 4.000 bytes para seus ponteiros. No mesmo caso, a matriz de ponteiros usará apenas 2.000 bytes, economizando 2.000 bytes.

O método mais rápido é o da matriz de ponteiros. Como no exemplo da planilha, quase sempre existe um método fácil de associar a matriz de ponteiros à matriz lógica. Por este método, o acesso a uma matriz dispersa fica quase tão rápido quanto o acesso a uma matriz normal. A lista encadeada é muito mais lenta, pois utiliza buscas lineares para localizar elementos na matriz. E mesmo se mais informação fosse adicionada à lista encadeada, para permitir acesso mais veloz, tal acesso ainda seria mais lento que o acesso quase direto da matriz de ponteiros. Mesmo a árvore binária parece uma tartaruga quando comparada à matriz de ponteiros.

Se o algoritmo de geração do *hash* for bem escolhido, o *hashing* poderá se tornar mais rápido que a árvore binária, mas nunca será mais rápido que a matriz de ponteiros.

Sempre que possível, o método da matriz de ponteiros deve ser usado. Se, entretanto, a memória for um fator crítico, você poderá ser obrigado a usar a lista encadeada ou a árvore binária.

BUFFERS REUTILIZÁVEIS

Quando houver pouca memória disponível, a alocação dinâmica pode ser usada no lugar de variáveis normais. Imagine um programa que contenha dois processos, **A** e **B**. Assumiremos que **A** requer 60% da memória livre e **B** requer 55% da memória livre para serem executados. Se **A** e **B** alocarem memória a partir de variáveis locais, **A** nunca poderá chamar **B** ou vice-versa, pois mais de 100% da memória seria necessária. Se **A** nunca chama **B**, não há problema; isto até você querer que **A** chame **B**. A única maneira de fazer isto é usando armazenamento dinâmico para ambos, **A** e **B**, e liberando a memória antes que um processo chame o outro. Ou seja, se tanto **A** quanto **B** usam mais da metade da memória livre disponível e **A** deve chamar **B**, ambos devem usar alocação dinâmica. Deste modo, ambos os processos terão a memória necessária no momento desejado.

Imagine que existam 100.000 bytes de memória livre restantes num computador que esteja executando um programa onde existam os seguintes procedimentos:

```
procedure B; foward;
procedure A;
var
      a:array[1..60000] of char;


begin
        .
        .
        .
        B;
        .
        .
        .
end;

procedure B;
var
    b:array[1..55000] of char;
begin


end;
```

A e **B** têm variáveis locais, cada uma exigindo mais da metade da memória disponível. **B** não poderá ser executado, pois não há espaço para alocar os 55.000 bytes necessários à matriz **b**.

Este tipo de problema muitas vezes é insolúvel, mas em algumas situações ele pode ser contornado. Se **A** não precisar manter o conteúdo da matriz **a** enquanto **B** estiver sendo executado, **A** e **B** poderão dividir o mesmo espaço da memória. Isto pode ser feito alocando espaço para as matrizes **a** e **b** dinamicamente. **A** então liberaria a memória antes de chamar **B** e a realocaria mais tarde. Os procedimentos teriam o seguinte formato:

```
procedure B;foward;
procedure A;
var
        a:^array[1..60000] of char;
begin
        New(a);
        .
        .
        .
        Dispose(a); {Libera memoria para B}
        B;
        New(a); {recupera a memoria}
        .
        .
        .
        Dispose(a);
end;

procedure B;
var
        b:^array[1..55000] of char;
begin
        New(b);


        .
        Dispose(b);
end;
```

Apenas o ponteiro **a** existe enquanto **B** é executado. Apesar de só ser usada ocasionalmente, esta técnica é, quase sempre, o único modo de resolver este tipo de problema.

O DILEMA DA "MEMÓRIA DESCONHECIDA"

Se você é um programador profissional, provavelmente já se encontrou frente ao dilema da "memória desconhecida". Ele ocorre quando você escreve um programa que tenha parte de sua performance baseada na quantidade de memória existente em qualquer computador onde o programa possa ser executado. Planilhas eletrônicas, malas diretas na RAM e programas de ordenação são o tipo de programa que pode ter este problema. Por exemplo,

um programa de ordenação, capaz de manipular 10.000 endereços num computador de 256K, pode ser capaz de trabalhar com apenas 5.000 endereços numa máquina de 128K. Se o programa deve funcionar em computadores de memória previamente desconhecida, será muito difícil determinar o tamanho ótimo da matriz que conterá a informação a ser ordenada, isto por duas razões: ou o programa não funcionará em máquinas cuja memória seja muito pequena para conter a matriz, ou você criará a matriz para o pior caso, não permitindo a usuários que tivessem máquinas com memória maior tirar proveito de tal memória. A solução é utilizar alocação dinâmica para conter a informação.

Editores de texto são bons exemplos do dilema da memória e de sua solução. Na maioria dos editores de texto, o número de caracteres que pode ser mantido não é fixo. A memória disponível do computador é usada para armazenar o texto digitado pelo usuário. Por exemplo, para cada linha digitada, é alocada a memória necessária e uma lista encadeada é mantida. Quando uma linha é apagada, a memória é devolvida ao sistema. Este editor poderia ser criado com o uso do seguinte registro para cada linha:

```
Ptr_Lin= ^linha;
str80= string[80];

linha= record
       texto: str80; {contem as linhas}
       num: integer;
       proximo: Ptr_Lin; {ponteiro para o proximo registro}
       anterior: Ptr_Lin; {ponteiro para o registro anterior}
end;
```

Este registro sempre aloca memória o bastante para que cada linha possa ter até 80 caracteres de comprimento. Num ambiente real, apenas o comprimento exato de cada linha seria alocado, e o restante usado apenas se a linha fosse alterada. O elemento **num** mantém o número de cada linha do texto. Isto permite que a função **Armazena_EDO** seja usada para criar e manter o arquivo de texto como uma lista encadeada.

O programa completo de um editor de texto simples é apresentado a seguir. Ele permite que linhas sejam inseridas ou apagadas em qualquer ponto, dado o número da linha. O texto pode também ser listado e gravado em arquivo de disco.

O editor tem sua operação baseada numa lista encadeada e ordenada de linhas de texto. A chave de ordenação é o número de cada linha. Texto pode ser inserido e apagado facilmente, bastando para isto especificar o número da linha. A única função mais difícil de entender é **Renumera**, que renumera o elemento **num** quando uma linha é inserida ou apagada.

Neste exemplo, a quantidade de texto que o editor pode conter está diretamente baseada na quantidade de memória livre no sistema do usuário. Assim, o editor usa

automaticamente qualquer memória adicional, sem precisar ser reprogramado. Esta é talvez a razão mais importante a favor do uso de alocação dinâmica para resolver dilemas de memória.

O programa, como mostrado aqui, é bastante limitado. Porém, suas rotinas de apoio à edição de texto são sólidas. Você pode expandi-lo para uso pessoal.

```
program Ed_Texto;

type
Ptr_Lin= ^linha;
str80= string[80];

linha= record
       texto: str80; {contem as linhas}
       num: integer;
       proximo: Ptr_Lin; {ponteiro para o proximo registro}
       anterior: Ptr_Lin; {ponteiro para o registro anterior}
end;
Dado= linha;
arq= file of linha;
var
   texto: arq;
   primeiro,ultimo: Ptr_Lin;
   feito: boolean;
   fnome: str80;

function Menu: char; {retorna a opcao do usuario}
var
   car: char;

begin
   WriteLn('1. Escrever');
   WriteLn('2. Apagar uma linha');
   WriteLn('3. Mostrar o arquivo');
   WriteLn('4. Gravar');
   WriteLn('5. Ler');
   WriteLn('6. Fim');
   repeat
      WriteLn;
      Write('Digite sua opcao: ');
      Read(car); car:=UpCase(car); WriteLn;
   until (car >= '1') and (car <= '6');
   Menu:=car;
end;  {Menu}

function Acha(lnum: integer): Ptr_Lin;
var
   i: Ptr_Lin;
```

```
begin
   i:=primeiro;
   Acha:=nil;
   while(i <> nil) do begin
      if lnum= i^.num then Acha:=i;
      i:= i^.proximo;
   end;
end;  {Acha}

procedure Renumera(lnum, incr: integer);
var
   i: Ptr_Lin;

begin
   i:=Acha(lnum);
   while(i <> nil) do begin
      i^.num:=i^.num+incr;
      i:=i^.proximo;
   end;
end;  {Renumera}

function Armazena_EDO(item, primeiro: Ptr_Lin;
                      var ultimo: Ptr_Lin): Ptr_Lin;
{armazena dados em lista ordenada}
var
   velho, topo: ^linha;
   feito: boolean;
begin
   topo:=primeiro;
   velho:=nil;
   feito:=FALSE;

   if primeiro=nil then
   begin {primeiro elemento da lista}
      item^.proximo:= nil;
      ultimo:=item;
      item^.anterior:=nil;
      Armazena_EDO:=item;
   end else
   begin
      while (primeiro <> nil) and (not feito) do begin
         if primeiro^.num < item^.num then begin
            velho:=primeiro;
            primeiro:=primeiro^.proximo;
         end else begin  {no meio}
            if velho <> nil then begin
                  velho^.proximo:=item;
                  item^.proximo:=primeiro;
                  primeiro^.anterior:=item;
                  item^.anterior:=velho;
                  Armazena_EDO:=topo;  {mesmo inicio}
                  feito:=TRUE;
            end else begin
                  item^.proximo:=primeiro;  {novo primeiro elemento}
                  item^.anterior:=nil;
                  Armazena_EDO:= item;
                  feito:=TRUE;
            end;
         end;
```

```
      end;  {while}
      if not feito then
      begin
         ultimo^.proximo:=item;   {no fim}
         item^.proximo:=nil;
         item^.anterior:=ultimo;
         ultimo:=item;
         Armazena_EDO:=topo;
      end;
   end;
end;  {Armazena_EDO}

function Apaga_DE(primeiro: Ptr_Lin; key: integer): Ptr_Lin;

var
   temp, temp2: Ptr_Lin;
   feito: boolean;
begin
   if primeiro^.num=key then
   begin  {apaga o primeiro da lista}
      Apaga_DE:=primeiro^.proximo;
      if temp^.proximo <> nil then begin
         temp:=primeiro^.proximo;
         temp^.anterior:=nil;
      end;
      Dispose(primeiro);
   end else begin
      feito:=FALSE;
      temp:=primeiro^.proximo;
      temp2:=primeiro;
      while (temp <> nil) and (not feito) do
      begin
         if temp^.num=key then
         begin
            temp2^.proximo:=temp^.proximo;
            if temp^.proximo <> nil then
                temp^.proximo^.anterior:=temp2;
            feito:=TRUE;
            ultimo:=temp^.anterior;
            Dispose(temp);
         end else
         begin
            temp2:=temp;
            temp:=temp^.proximo;
         end;
      end;
      Apaga_DE:=primeiro; {mesmo inicio}
      if not feito then WriteLn('Nao encontrada')
      else Renumera(key+1,-1);
   end;
end;  {Apaga_DE}

procedure Remove;
var
   num: integer;
begin
   Write('Linha a ser apagada: ');
   Read(num); WriteLn;
   primeiro:=Apaga_DE(primeiro,num)
end;  {Remove}
```

```
procedure Digitacao;
var
   item: Ptr_Lin;
   num: integer;
   feito: boolean;
begin
   feito:=FALSE;
   Write('Numero da primeira linha: ');
   Read(num); WriteLn;
   repeat
      New(item); {abre um novo registro}
      item^.num:=num;
      Write(item^.num, ':');
      Read(item^.texto); WriteLn;
      if length(item^.texto)=0 then feito:=TRUE
      else begin
         if Acha(num) <> nil then Renumera(num,1);
         primeiro:=Armazena_EDO(item,primeiro,ultimo);
      end;
      num:=num+1;
   until feito;
end; {Enter}

procedure Mostra(primeiro: Ptr_Lin);
begin
   while primeiro <> nil do
   begin
      Write(primeiro^.num, ':');
      WriteLn(primeiro^.texto);
      primeiro:=primeiro^.proximo;
   end;
   WriteLn;
end;  {Mostra}

procedure Grava(var f:arq; primeiro: Ptr_Lin);
begin
   WriteLn('saving file');
   ReWrite(f);
   while primeiro <> nil do
   begin
      Write(f,primeiro^);
      primeiro:=primeiro^.proximo;
   end;
end;  {Grava}

function Le(var f: arq): Ptr_Lin;
{retorna um ponteiro para o primeiro da lista}
var
   temp: Ptr_Lin;
begin
   WriteLn('Le file');
   Reset(f);
   while primeiro <> nil do
   begin  {libera memoria}
      temp:=primeiro^.proximo;
      Dispose(primeiro);
      primeiro:=temp;
```

```
    end;
    ultimo:=nil; primeiro:=nil;
    while not eof(f) do
    begin
       New(temp);
       Read(f,temp^);
       primeiro:=Armazena_EDO(temp,primeiro,ultimo);
    end;
    Le:=primeiro;
end;  {Le}

begin
    primeiro:=nil; {lista inicialmente vazia}
    ultimo:=nil;
    feito:=FALSE;

    Write('Nome do arquivo: ');
    Read(fnome);
    WriteLn;
    assign(texto,fnome);

    repeat
       case Menu of
          '1': Digitacao;
          '2': Remove;
          '3': Mostra(primeiro);
          '4': Grava(texto,primeiro);
          '5': primeiro:=Le(texto);
          '6': feito:=TRUE;
       end;
    until feito=TRUE;
end.
```

FRAGMENTAÇÃO

Fragmentação ocorre quando pedaços de memória livre ficam presos entre blocos de memória alocada. Apesar da quantidade de memória livre ser, em geral, suficiente para dar conta de alocações, pode ocorrer de os pedaços individuais serem pequenos demais, ainda que representem memória suficiente se somados. A Figura 4-4 mostra como uma seqüência de **News** e **Disposes** pode gerar este problema.

Alguns tipos de fragmentação são evitados porque as funções de alocação dinâmica associam regiões adjacentes da memória. Por exemplo, se as regiões da memória A, B, C e D (mostradas a seguir) forem alocadas e então as regiões B e C forem liberadas, B e C teoricamente poderão ser combinadas, pois estão lado a lado. Porém, se B e D forem liberadas, não haverá como associá-las, pois C está entre elas.

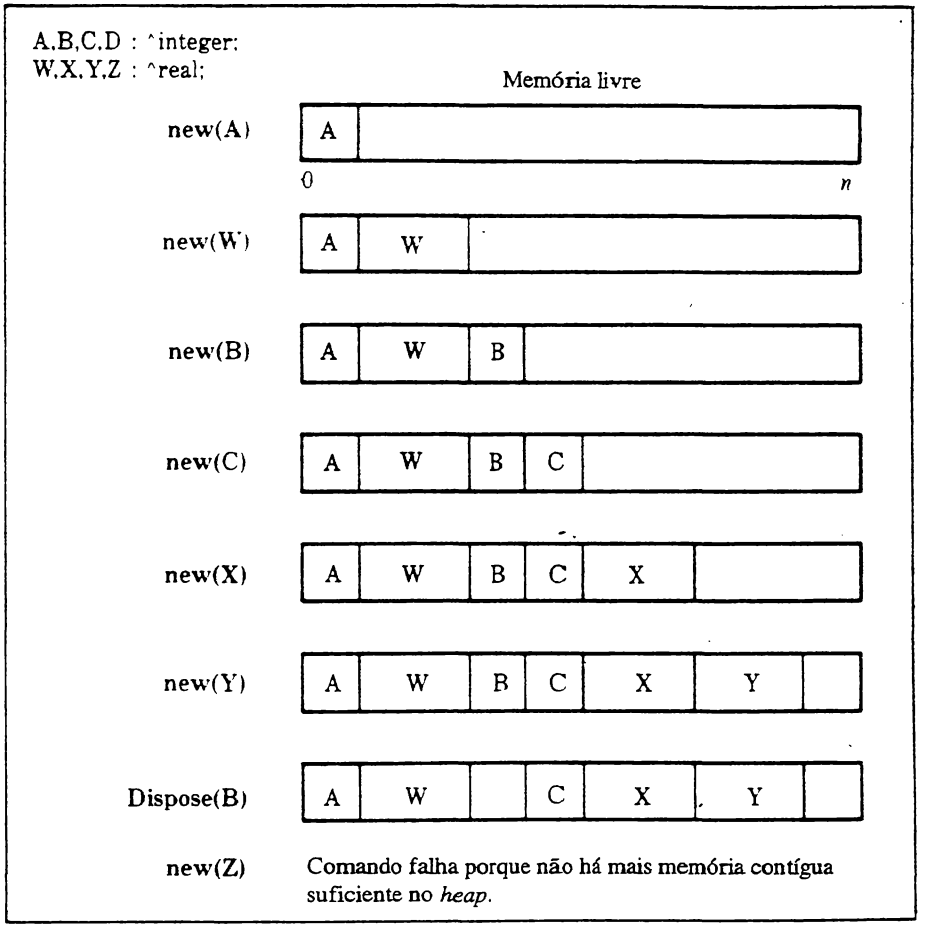
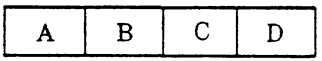


Figura 4-4 Fragmentação numa alocação dinâmica.



Como B e D foram liberadas enquanto C estava alocada, você pode imaginar que basta mover o conteúdo de C para D, e então combinar B e C. O problema é que seu programa não teria meios de saber que o conteúdo de C foi movido para D.

Um modo de evitar fragmentação excessiva é sempre alocar quantidades iguais de memória; deste modo, todas as regiões liberadas poderão ser realocadas posteriormente,

e toda a memória livre poderá ser usada. Se isto não for possível, tente limitar-se a uns poucos tamanhos diferentes. Isto pode, algumas vezes, ser conseguido pela compactação de várias pequenas alocações numa só grande alocação. Nunca aloque mais memória que o necessário, apenas para evitar a fragmentação; a memória assim perdida é muito maior que o ganho obtido. Existe uma outra solução: durante a execução do programa, grave todas as informações num arquivo de disco temporário, libere toda a memória e leia as informações de volta. Os espaços existentes serão eliminados na leitura das informações a partir do disco.

ALOCAÇÃO DINÂMICA E INTELIGÊNCIA ARTIFICIAL

Apesar de não ser uma linguagem específica de inteligência artificial (IA), o Pascal pode ser usado em algumas experiências. Um traço comum a muitos programas de inteligência artificial é a existência de uma lista que o próprio programa cuida de aumentar, quando ele "aprende" algo novo. Numa linguagem como LISP, considerada a principal linguagem de inteligência artificial, a própria linguagem mantém a lista. Em Pascal, é necessário programar estes procedimentos, usando listas encadeadas e alocação dinâmica. Apesar de simples, o exemplo mostrado aqui pode ser aplicado a outros programas "inteligentes" mais sofisticados.

Uma área interessante da IA é aquela dedicada à criação de programas que parecem comportar-se como seres humanos. O famoso programa ELIZA, por exemplo, parecia ser um psiquiatra. Seria ótimo ter um programa de computador capaz de conversar sobre qualquer assunto; um programa para rodar nos momentos de cansaço e solidão! O exemplo usado nesta seção é uma versão extremamente simples de tal programa. Ele utiliza palavras e as definições destas palavras para manter uma conversa simples com o usuário. Um artifício muito comum nos programas de IA é a ligação entre um item de informação e seu significado; neste caso, o programa liga palavras a suas definições. O registro a seguir mantém cada palavra, sua definição, sua função gramatical e sua conotação:

```
Ptr_Vocab= ^Vocab;
str80= string[80];
str30= string[30];

vocab= record
       tipo: char;
       conotacao: char;
```

```
      palavra: str30;
      def: str80;
      proximo: Ptr_Vocab; {aponta para o proximo registro}
      anterior: Ptr_Vocab; {aponta para o registro anterior}
end;
```

No programa, você digita uma palavra, sua definição, sua função e sua conotação, boa, má, ou indiferente. Para manter este minidicionário, uma lista encadeada é criada, usando alocação dinâmica. **Armazena_EDO** cria e mantém uma lista duplamente encadeada e ordenada no dicionário. Após preencher o dicionário com algumas palavras, você pode começar a conversar com o computador. Você digita uma sentença, por exemplo, "O dia está bonito". O programa procura na sentença um substantivo conhecido. Se achar, ele então faz um comentário sobre aquele substantivo, baseado na definição existente. Quando o programa encontra uma palavra desconhecida, ele pede a você que digite a palavra e sua definição. Você sai do modo de conversação, digitando "fim".

O procedimento **Falar** é a parte do programa que mantém a conversação. A função de apoio **Dissecar** verifica a frase por você digitada, palavra por palavra. A variável **sentenca** mantém a sentença digitada. **Dissecar** remove uma palavra de cada vez de **sentenca** e coloca em **palavra.** Aqui estão as funções **Falar** e **Dissecar:**

```
procedure Dissecar(var s: str80; var p: str30);
var
   t,x: integer;
   temp: str80;
begin
   t:=1;
   while(s[t]=' ') do t:=t+1;
   x:=t;
   while(s[t] <> ' ') and (t <= length(s)) do t:=t+1;
   if t <= length(s) then t:=t-1;
   p:=Copy(s,x,t-x+1);
   temp:=s;
   s:=Copy(temp,t+1,length(s));
end;  {Dissecar}

procedure Falar;
var
   sentenca: str80;
   palavra: str30;
   p: Ptr_Vocab;
begin
   WriteLn('Modulo de conversacao (Digite  Fim  para voltar ao menu)');
   repeat
      Write(': ');
      Read(sentence); WriteLn;
      repeat
         Disseca(sentenca,palavra);
         p:=Busca(primeiro,palavra);
         if p <> nil then
```

```
         begin
            if p^.tipo='n' then
            begin
               case p^.conotacao of
                  'b': Write('Eu gosto de ');
                  'm': Write('Eu nao gosto de ');
               end;
               WriteLn(p^.def);
            end  {if}
            else WriteLn(p^.def);
         end
         else if palavra <> 'fim' then
         begin
            WriteLn(palavra,' e'' desconhecida.');
            Digitacao(TRUE);
         end;
      until length(sentenca)=0;
   until palavra='fim';
end;  {Falar}
```

Eis o programa completo:

```
program Conversacao;

type

Ptr_Vocab= ^Vocab;
str80= string[80];
str30= string[30];

vocab= record
       tipo: char;
       conotacao: char;
       palavra: str30;
       def: str80;
       proximo: Ptr_Vocab; {aponta para o proximo registro}
       anterior: Ptr_Vocab; {aponta para o registro anterior}
end;

Item= vocab;
Matriz= array[1..100] of Ptr_Vocab; {contem os ponteiros para
                                     os registros vocab}

arq= file of vocab;
 var
    teste: Matriz;
    conversa: arq;
    primeiro,ultimo: Ptr_Vocab;
    feito: boolean;

function Menu: char; {retorna a opcao do usuario}
var
   car: char;

begin
   WriteLn('1. Digitar palavras');
   WriteLn('2. Apagar uma palavra');
   WriteLn('3. Mostrar lista');
```

```
   WriteLn('4. Procurar uma palavra');
   WriteLn('5. Gravar arquivo');
   WriteLn('6. Ler arquivo');
   WriteLn('7. Conversar');
   WriteLn('8. Fim');
   repeat
      WriteLn;
      Write('Digite sua opcao: ');
      Read(car); car:=UpCase(car); WriteLn;
   until (car >= '1') and (car <= '8');
   Menu:=car;
end;  {Menu}

function Armazena_EDO(item, primeiro: Ptr_Vocab;
                      var ultimo: Ptr_Vocab): Ptr_Vocab;
{armazena dados em lista ordenada}
var
   velho, topo: ^vocab;
   feito: boolean;
begin
   topo:=primeiro;
   velho:=nil;
   feito:=FALSE;

   if primeiro=nil then
   begin {primeiro elemento da lista}
      item^.proximo:= nil;
      ultimo:=item;
      item^.anterior:=nil;
      Armazena_EDO:=item;
   end else
   begin
      while (primeiro <> nil) and (not feito) do begin
         if primeiro^.palavra < item^.palavra then begin
            velho:=primeiro;
            primeiro:=primeiro^.proximo;
         end else begin  {no meio}
            if velho <> nil then begin
                 velho^.proximo:=item;
                 item^.proximo:=primeiro;
                 primeiro^.anterior:=item;
                 item^.anterior:=velho;
                 Armazena_EDO:=topo;  {mesmo inicio}
                 feito:=TRUE;
            end else begin
                 item^.proximo:=primeiro;  {novo primeiro elemento}
                 item^.anterior:=nil;
                 Armazena_EDO:= item;
                 feito:=TRUE;
            end;
         end;
      end;  {while}
      if not feito then
      begin
         ultimo^.proximo:=item;   {no fim}
         item^.proximo:=nil;
         item^.anterior:=ultimo;
         ultimo:=item;
```

```
        Armazena_EDO:=topo;
      end;
  end;
end;  {Armazena_EDO}


function Apaga_DE(primeiro: Ptr_Vocab; chave: str80): Ptr_Vocab;

var
   temp, temp2: Ptr_Vocab;
   feito: boolean;
begin
   if primeiro^.palavra=chave then
   begin  {apaga o primeiro da lista}
      Apaga_DE:=primeiro^.proximo;
      if temp^.proximo <> nil then begin
         temp:=primeiro^.proximo;
         temp^.anterior:=nil;
      end;
      Dispose(primeiro);
   end else begin
      feito:=FALSE;
      temp:=primeiro^.proximo;
      temp2:=primeiro;
      while (temp <> nil) and (not feito) do
      begin
         if temp^.palavra=chave then
         begin
            temp2^.proximo:=temp^.proximo;
            if temp^.proximo <> nil then
               temp^.proximo^.anterior:=temp2;
            feito:=TRUE;
            ultimo:=temp^.anterior;
            Dispose(temp);
         end else
         begin
            temp2:=temp;
            temp:=temp^.proximo;
         end;
      end;
      Apaga_DE:=primeiro; {mesmo inicio}
      if not feito then WriteLn('Nao encontrada')
   end;
end;  {Apaga_DE}

procedure Remove;
var
   nome: str80;
begin
   Write('Palavra a ser apagada: ');
   Read(nome); WriteLn;
   primeiro:=Apaga_DE(primeiro,nome)
end;  {Remove}
```

```
procedure Digitacao(one: boolean);
var
   item: Ptr_Vocab;
   feito: boolean;
begin
   feito:=FALSE;
   repeat
      New(item); {abre um novo registro}
      Write('Palavra: ');
      Read(item^.palavra); WriteLn;
      if Length(item^.palavra)=0 then feito:=TRUE
      else
      begin
         Write('Tipo(s,v,a): ');
         Read(item^.tipo); WriteLn;
         Write('Conotacao(b,m,n): ');
         Read(item^.conotacao); WriteLn;
         Write('Definicao: ');
         Read(item^.def); WriteLn;
         primeiro:=Armazena_EDO(item,primeiro,ultimo);
      end;
   until feito;
end; {Digitacao}

procedure Mostra(primeiro: Ptr_Vocab);
begin
   while primeiro <> nil do
   begin
      WriteLn('Palavra: ',primeiro^.palavra);
      WriteLn('Tipo: ',primeiro^.tipo);
      WriteLn('Conotacao: ',primeiro^.conotacao);
      WriteLn('Definicao');
      WriteLn(primeiro^.def);
      WriteLn;
      primeiro:=primeiro^.proximo;
   end;
   WriteLn;
end;  {Mostra}

function Procura(primeiro: Ptr_Vocab; palavra: str30): Ptr_Vocab;
var
   feito: boolean;
begin
   feito:=FALSE;
   while (primeiro <> nil) and (not feito) do
   begin
      if palavra=primeiro^.palavra then begin
         Procura:=primeiro;
         feito:=TRUE;
      end else
         primeiro:=primeiro^.proximo;
   end;
   if primeiro=nil then Procura:=nil; {nao esta na lista}
end;  {Procura}
```

```
procedure Acha;
var
   loc: Ptr_Vocab;
   palavra: str30;
begin
   Write('Palavra a ser procurada:
   Read(palavra); WriteLn;
   loc:=Procura(primeiro,palavra);
   if loc <> nil then
   begin
      WriteLn('Palavra: ',loc^.palavra);
      WriteLn('Tipo: ',loc^.tipo);
      WriteLn('Conotacao: ',loc^.conotacao);
      WriteLn('Definicao');
      WriteLn(loc^.def);
   end
   else WriteLn('Nao esta na lista.');
end;  {Acha}


procedure Grava(var f:arq; primeiro: Ptr_Vocab);
begin
   WriteLn('Gravando arquivo.');
   ReWrite(f);
   while primeiro <> nil do
   begin
      Write(f,primeiro^);
      primeiro:=primeiro^.proximo;
   end;
end;  {Grava}

function Le(var f: arq; primeiro: Ptr_Vocab): Ptr_Vocab;
var
   temp: Ptr_Vocab;
begin
   WriteLn('Lendo arquivo.');
   Reset(f);
   while primeiro <> nil do
   begin  {libera memoria}
      temp:=primeiro^.proximo;
      Dispose(primeiro);
      primeiro:=temp;
   end;
   ultimo:=nil; primeiro:=nil;
   while not eof(f) do
   begin
      New(temp);
      Read(f,temp^);
      primeiro:=Armazena_EDO(temp,primeiro,ultimo);
   end;
   Le:=primeiro;
end;  {Le}


procedure Dissecar(var s: str80; var p: str30);
var
```

```
   t,x: integer;
   temp: str80;
begin
   t:=1;
   while(s[t]=' ') do t:=t+1;
   x:=t;
   while(s[t] <> ' ') and (t <= length(s)) do t:=t+1;
   if t <= length(s) then t:=t-1;
   p:=Copy(s,x,t-x+1);
   temp:=s;
   s:=Copy(temp,t+1,length(s));
end;  {Dissecar}

procedure Falar;
var
   sentenca: str80;
   palavra: str30;
   p: Ptr_Vocab;
begin
   WriteLn('Modulo de conversacao (Digite ''Fim'' para voltar ao menu)');
   repeat
      Write(': ');
      Read(sentenca); WriteLn;
      repeat
         Dissecar(sentenca,palavra);
         p:=Procura(primeiro,palavra);
         if p <> nil then
         begin
            if p^.tipo='s' then
            begin
               case p^.conotacao of
                  'b': Write('Eu gosto de ');
                  'm': Write('Eu nao gosto de ');
               end;
               WriteLn(p^.def);
            end  {if}
            else WriteLn(p^.def);
         end
         else if palavra <> 'fim' then
         begin
            WriteLn(palavra,' e'' desconhecida.');
            Digitacao(TRUE);
         end;
      until length(sentenca)=0;
   until palavra='fim';
end;  {Falar}



begin
   primeiro:=nil;  {lista inicialmente vazia}
   ultimo:=nil;
   feito:=FALSE;
```

```
   Assign(conversa, 'conversa.dic');
   repeat
      case Menu of
         '1': Digitacao(FALSE);
         '2': Remove;
         '3': Mostra(primeiro);
         '4': Acha;
         '5': Grava(conversa,primeiro);
         '6': primeiro:=Le(conversa,primeiro);
         '7': Falar;
         '8': feito:=TRUE;
      end;
   until feito=TRUE;
end.
```

Este programa é engraçado e fácil de escrever. Ele pode ser melhorado, para parecer bem mais esperto. Uma maneira seria fazer o programa procurar os verbos da sentença e substituí-los por sinônimos, em seu comentário. O programa também poderia fazer perguntas.

CAPÍTULO 5

INTERFACEAMENTO COM ROTINAS EM LINGUAGEM DE MÁQUINA E COM O SISTEMA OPERACIONAL

Embora o Turbo Pascal seja muito poderoso, às vezes é necessário escrever uma rotina em *assembly* ou usar um serviço do sistema operacional. Isto pode ser necessário para obter maior velocidade de execução ou para controlar algum dispositivo especial não previsto no Turbo Pascal. Seja qual for a razão, o Turbo Pascal foi projetado com flexibilidade para lidar com estas complementações em linguagem de máquina.

Cada processador tem uma linguagem de montagem *(assembly language)* diferente, e cada sistema operacional tem uma estrutura de interfaceamento própria. Além disso, a convenção para as chamadas ao sistema, que define como os subprogramas recebem e devolvem as informações, varia um pouco nas versões CP/M, CP/M-86 e MS-DOS do Turbo Pascal. Este capítulo é baseado no sistema operacional PC-DOS e no processador 8086, que são os mais usados hoje. Mesmo que você use um equipamento diferente, a discussão a seguir serve de orientação.

INTERFACEAMENTO COM LINGUAGEM DE MÁQUINA

Há várias razões para se usar uma rotina em *assembly:*

- Para aumentar a velocidade e eficiência.
- Para efetuar uma operação específica na máquina, não disponível através do Turbo Pascal.

– Para aproveitar uma rotina já pronta em *assembly*.

Apesar de todos os compiladores de Turbo Pascal produzirem programas-objeto rápidos e compactos, não há um compilador capaz de criar programas mais rápidos ou compactos do que um programador competente pode fazer com um *assembly*. A diferença em geral não vale o tempo extra necessário para se escrever um programa em *assembly*. Entretanto, há casos especiais em que uma determinada função ou procedimento têm de ser escritos em *assembly* para rodar mais rápido. Isto é necessário quando um subprograma é usado muito freqüentemente, e portanto afeta muito a velocidade do programa como um todo. Um bom exemplo é um pacote de aritmética de ponto-flutuante. Além disso, alguns dispositivos especiais de *hardware* necessitam de programas capazes de responder em intervalos de tempo muito precisos.

Muitos computadores, inclusive máquinas baseadas no 8086/8088, têm características úteis que não podem ser acessadas pelo Turbo Pascal. Por exemplo, você não pode mudar o segmento de dados do programa, nem modificar o conteúdo de um registrador de CPU usando um comando do Pascal.

Em ambientes de programação profissionais, é comum a compra de bibliotecas de sub-rotinas para funções como aritmética com ponto-flutuante ou gráficos. Muitas vezes, este tipo de rotina tem de ser usado na forma de objeto porque o autor não fornece o programa-fonte. Às vezes, basta "linkar" estas rotinas ao programa em Pascal; outras vezes, é preciso criar um módulo de interface entre o programa e a rotina, a fim de corrigir qualquer problema de interfaceamento entre os dois.

Há duas maneiras de integrar módulos em linguagem de máquina com programas em Pascal. A primeira é produzir a rotina em separado, montá-la com um *assembly* e ligá-la ao resto do programa através do comando **external**. A segunda maneira é usar o comando **inline** para incluir as rotinas em linguagem de máquina no corpo do programa.

Não é meta deste livro ensinar programação em *assembly*. Este capítulo pressupõe familiaridade com a linguagem de máquina do seu computador. Os exemplos apresentados servem apenas como guias.

FORMATOS INTERNOS DE DADOS E CONVENÇÕES DE CHAMADA DO TURBO PASCAL

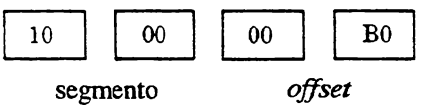
Antes de usar uma rotina em *assembly* com o Turbo Pascal você deve saber como os dados são armazenados num programa e como eles passam de um subprograma para outro. Na

versão MS-DOS, todas as variáveis globais são guardadas no segmento de dados e acessadas pelo registrador DS. Todas as variáveis locais são guardadas no *stack* e acessadas pelo registrador BP. Constantes tipadas, que são armazenadas no segmento de código, são referenciadas através do registrador CS. A Figura 5-1 mostra como cada tipo de dado é armazenado.

Tipo	Comprimento	Comentários
byte	1 byte	
char	1 byte	
integer	2 bytes	
real	6 bytes	O primeiro byte é o expoente; o próximo byte é o byte menos significativo; o último byte é o mais significativo.
string	varia	O primeiro byte contém o comprimento atual, e o próximo contém o primeiro caractere.
set	varia	Uma vez que cada elemento usa um bit, e o número máximo de elementos é 256, o maior conjunto ocupará 32 bytes.
pointer	4 bytes	Dois bytes para segmento; dois para *offset*. Guardados em formato de byte reverso. **nil** é 00 00 00 00.
array	varia	Elementos de índice menor em endereços mais baixos; índice maior em endereços mais altos.
record	varia	Primeiro campo no endereço mais baixo; último no mais alto.

Figura 5-1 Armazenagem de tipos predefinidos.

Lembre-se: ponteiros *(pointers)* são armazenados em formato de byte reverso. Desta forma, um *pointer* contendo *offset* $B000 e segmento $0010 é armazenado assim:



A *convenção de chamada* é o método usado pelo compilador Turbo Pascal para passagem de informações para os subprogramas e para retornar os valores das funções. O Turbo Pascal usa o *stack* para passar os parâmetros para os subprogramas e para retornar o resultado de certas funções. (O registrador AX retorna resultados de um ou dois bytes.) O conteúdo exato do *stack*, quando um subprograma é chamado, depende tanto do tipo do dado como da natureza de sua declaração (ou seja, se a passagem do parâmetro é feita por *valor* ou por *referência*).

PARÂMETROS POR VALOR Parâmetros por valor são unidirecionais: a informação é passada para o subprograma, mas qualquer modificação que ela sofra não alterará a variável usada na chamada do subprograma. Em vez de trabalhar com aquela variável, uma cópia de seu valor é feita e passada, deixando assim a variável intacta. Em resumo, só um valor é passado para o procedimento ou função.

Os parâmetros de tipo **integer, char, byte** e **boolean** são passados através do *stack* em uma palavra (dois bytes). Outros tipos escalares ordinais declarados também são passados assim. Como no caso de variáveis **booleanas**, quando apenas um byte é necessário, a metade mais significativa da palavra é zerada. Um parâmetro do tipo **real** usa seis bytes. Uma seqüência de caracteres (*string*) usa um byte além do seu tamanho declarado. Este byte extra contém o comprimento atual da *string*, e fica acima da *string* no *stack*. Todos os conjuntos (tipo **set**) usam 32 bytes no *stack* quando passados como parâmetro de valor.

Os *pointers* consistem em duas palavras: o segmento e o *offset*. Se o pointer é **nil**, então ambas as palavras são zero.

PARÂMETROS POR REFERÊNCIA Ao contrário dos parâmetros passados por valor, os parâmetros por referência têm os endereços das respectivas variáveis passados para o *stack*. Isto significa que o subprograma trabalha diretamente com elas. Estes parâmetros, declarados com a palavra **var**, são bidirecionais: eles passam informações para a sub-rotina chamada e podem também retornar informações para a rotina que fez a chamada, pois estes parâmetros podem ser alterados. Duas palavras são passadas para o *stack*, contendo o segmento e o *offset* da variável, não importando seu tipo.

VALORES DE RETORNO DE FUNÇÕES Quando uma função em Turbo Pascal encerra sua execução, ela passa um valor para a rotina que chamou. Para os tipos escalares ordinais (**integer, char** etc.), o valor retorna através do registrador RX. Valores do tipo **boolean** também mudam o *zero flag*: 1 é **TRUE**, 0 é **FALSE**. Funções que retornam ponteiros colocam o segmento no registrador DX e o *offset* no AX.

Quando uma função retorna uma variável de vários bytes, a variável é colocada num "local de retorno de função" do *stack*. Como o tipo da variável retornada por uma função é conhecido no momento da chamada, Pascal aloca espaço suficiente no *stack* para armazená-la. No caso de variáveis do tipo **real**, este espaço é de seis bytes, com o expoente no endereço mais baixo. No caso de *strings*, matrizes e registros, o primeiro byte da

variável ocupa o endereço mais baixo. O local de retorno da função é logo abaixo do endereço de retorno da função. A Figura 5-2 mostra o *stack* no momento de uma chamada de função.

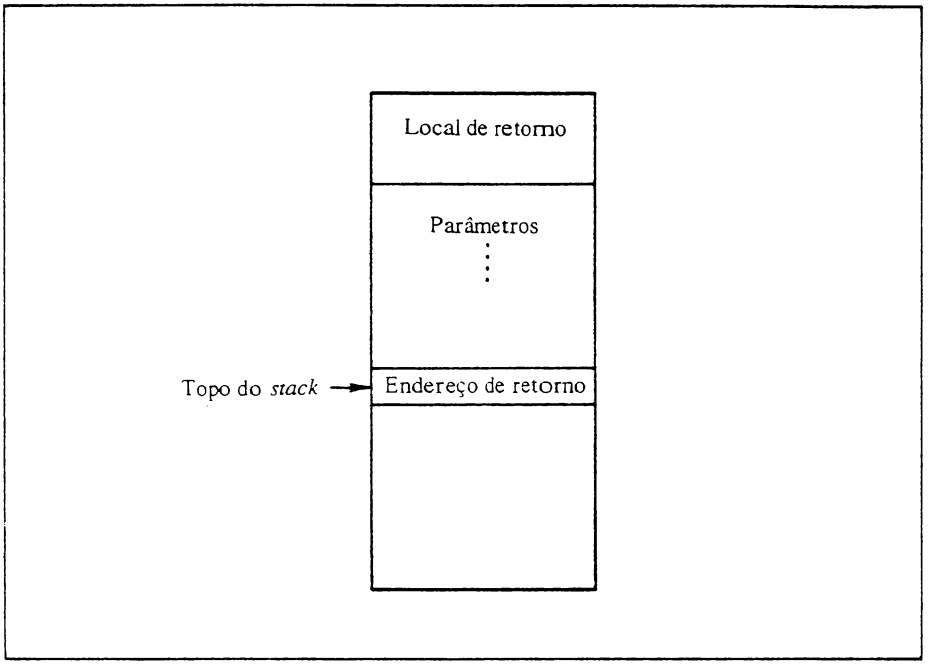


Figura 5-2 O *stack* quando uma função é chamada.

GUARDANDO REGISTRADORES Todo subprograma em *assembly* deve preservar os registradores BP, CS, DS e SS. Isto geralmente é feito com as instruções **push** e **pop.**

Ao escrever um módulo em *assembly* para uso com Turbo Pascal, siga as convenções descritas acima. Só assim suas rotinas em linguagem de máquina se comunicarão corretamente com seus programas em Pascal.

CRIAÇÃO DE UMA ROTINA EXTERNA EM ASSEMBLY

Agora que você conhece os princípios, eis um subprograma externo. Para este exemplo, vamos supor que seja necessário refazer a seguinte função em *assembly:*

```
function mul(a,b: integer): integer;
begin
   a:=a*b;
   mul:=a;
end;
```

Já que esta função será chamada por um programa em Turbo Pascal, você sabe que os dois argumentos do tipo **integer** serão passados para o *stack* na forma de duas palavras. Portanto, se **mul** for chamada assim:

```
mul(10,20);
```

o valor 20 deverá ser armazenado primeiro e o valor 10 depois, em ordem reversa. Lembre-se de que os argumentos são escalares, seus valores vão para o *stack*. No caso de matrizes e registros, os endereços é que vão para o *stack*.

O valor de retorno deve ser colocado em AX. Eis a função **mul** em *assembly:*

```
code            segment 'code'
                assume cs:code
xmul            proc near   ; deve ser near porque o Turbo Pascal
                            ; usa o modelo 8088
                push bp
                mov bp,sp

; primeiro parametro
                mov ax,[bp]+4
; multiplica pelo segundo
                mul [bp]+6
;limpa bp e o stack
;o resultado final ja esta em AX
                pop bp
                ret 4
xmul            endp
code            ends
                end
```

Note que todos os registradores apropriados são preservados com **push** e recuperados com **pop**. Os argumentos são retirados do *stack*. Se você não conhece *assembly* de 8086/8088, observe as instruções a seguir:

```
mov bp,sp
mov ax,[bp]+4
mul [bp]+6
```

Estas instruções colocam o endereço do topo do *stack* no registrador BP e movem o quarto e o quinto byte do *stack*, que é o parâmetro **a** da função, para o registrador AX. A seguir, a multiplicação é efetuada com o sexto e o sétimo byte do *stack*. Os parâmetros estão no quarto e no sexto byte porque o endereço de retorno da função e o registrador BP ocupam os quatro primeiros bytes. Assim, os parâmetros são encontrados a partir do quarto byte do *stack*.

Como o comentário no programa-fonte de **mul** explica, este é um procedimento "near". Isto é necessário para ser compatível com a maneira como o Turbo Pascal usa a memória num sistema baseado no 8086/8088. Se esta função fosse "far", o *stack* seria invalidado no retorno ao programa em Pascal.

Antes de usar a função externa **mul**, ela deve ser montada, "linkada" e transformada num arquivo .COM através do utilitário **exe2bin** fornecido com o MS-DOS. A seqüência correta é a seguinte:

```
masm mul;
link mul,,;
exe2bin mul.exe mul.com
```

O último passo transforma **mul** num arquivo .COM, como exige o Turbo Pascal.

Seus programas em Turbo Pascal agora podem usar a função externa **mul**. Por exemplo,

```
program AsmTest;

var
   a,b,c: integer;

function mul(x,y: integer): integer; external 'mul.com';

begin
   a:=40;
   b:=20;
   c:=mul(a,b);  {multiplica a por b e retorna o resultado}
   WriteLn(c);
end.
```

funciona desde que **mul.com** esteja no acionador em uso.

Lembre-se de que todos os exemplos estão em *assembly* de 8086/8088.

Se você usa a versão CP/M do Turbo Pascal, deve alterar os exemplos de acordo com as instruções do *Turbo Pascal Reference Manual*.

LINGUAGEM DE MÁQUINA VIA INLINE

Ao contrário do Pascal padrão, o Turbo Pascal tem uma extensão que permite programas em linguagem de máquina fazerem parte de um programa em Turbo Pascal. Desta forma, você não tem de usar um subprograma separado. Há duas vantagens nisso: primeiro, o interfaceamento é mais simples; segundo, o programa fica todo em um único arquivo, tornando a manutenção mais fácil. O único problema é que o formato do comando **inline** é trabalhoso.

O comando **inline** permite que um trecho em linguagem de máquina faça parte de um programa em Turbo Pascal. A forma geral do comando é:

inline(*valor1*/*valor2*/.../*valorN*);

onde *valorX* é o código-objeto de uma instrução de máquina ou um dado. Um * pode ser usado numa referência ao valor atual do *program-counter*. Em *assembly* 8088 usa-se $ para isso, mas como o $ marca valor hexadecimal no Turbo Pascal, optou-se pelo sinal *.

Quando o valor puder ser armazenado num único byte, ele o será; senão, dois bytes serão usados. Para contornar esta restrição, use os símbolos < e >. Se um valor começa com <, então apenas o byte menos significativo será considerado. Se um valor do tamanho de um byte for precedido de >, ele resultará em dois bytes, com o mais significativo em zero. Por exemplo, <$1234 resulta num único byte com o valor $34, enquanto >$12 resulta na palavra $0012.

O próximo programa multiplica dois números inteiros através da função **mul**, desta vez colocada num **inline.** Compare esta função **mul** com a sub-rotina externa, mostrada na seção anterior.

Nota do Revisor Técnico: as matrizes **Port** e **PortW** podem ser usadas para enviar ou receber respectivamente bytes e palavras pelas portas de E/S, sem necessidade de recorrer à linguagem de máquina.

```
program InLineAsm;  {este programa exemplifica o usc
                     do procedimento InLine}

var
   a,b,c: integer;

function mul(x,y: integer): integer;
begin
   inline($88/$46/$04/    {mov ax,[bp]+4}
          $F6/$66/$06/    {mul [bp]+6}
          $89/$EC/        {mov sp,bp}
          $5D/            {pop bp}
          $C2/$06/$00);   {ret 6}
end;  {mul}

begin
   a:=10;
   b:=20;
   c:=mul(a,b);
   WriteLn(c);
end.
```

Aqui, o compilador Turbo Pascal providencia as instruções de retorno da função. Quando o compilador compila um **inline**, as instruções deste são codificadas no meio das instruções da função **mul.** (*Nota do Revisor Técnico:* O programa acima funciona, mas o *Manual do Usuário do Turbo Pascal* recomenda que todo **inline** inicie com **pushs** para preservar os registradores, e encerre com **pops** para recuperá-los. Além disso, a instrução RET 6 no final do **inline** acima causa um retorno prematuro da função: quaisquer comandos entre o fim do **inline** e o fim da função não são executados.)

Um uso comum de **inline** é a comunicação com dispositivos não previstos pelo Turbo Pascal. Por exemplo, o subprograma abaixo poderia ser usado para ligar um ventilador quando um sensor de temperatura atingisse 100 graus. Este programa pressupõe que o envio do valor 1 pela porta de E/S (*I/O port*) número 200 acionará o ventilador:

```
procedure Vent(temp: integer);
{if temp >= 35 graus celsius, liga o ventilador }
begin
   if temp >= 100 then
      inline ($B8/00/01/    {mov AX,1}
              $E7/$C8);     {out 200,AX}
end;
```

Lembre-se: o compilador Turbo Pascal toma as providências necessárias para a chamada e retorno de uma função. Você só tem de escrever o corpo da função e seguir a convenção de chamada descrita no manual para acessar os parâmetros.

Seja qual for o método empregado, você estará criando rotinas que dependem das particularidades de uma máquina, tornando difícil a conversão do programa para

outras máquinas ou sistemas operacionais. Entretanto, para as situações extremas que exigem o uso de *assembly*, o esforço valerá a pena.

QUANDO PROGRAMAR EM ASSEMBLY

A maioria dos programadores só programa em *assembly* quando é absolutamente necessário, pois é uma tarefa difícil. Como regra geral, não o faça: causa muitas dores de cabeça. Entretanto, há duas situações em que trabalhar com um *assembly* faz sentido. A primeira situação é quando não há absolutamente nenhum outro jeito de resolver o problema – por exemplo, quando é necessário controlar dispositivos que o Turbo Pascal não pode manipular.

A segunda situação é quando o tempo de execução de um programa tem de ser reduzido. Neste caso, você deve escolher cuidadosamente as funções que devem ser escritas em *assembly*. Se você escolher as funções erradas, o aumento em velocidade será pequeno. Escolhendo as certas, o programa voa! Para determinar quais subprogramas devem ser reescritos, reveja o fluxo de operação do seu programa. Rotinas usadas em *loops* são as que devem ser reescritas, pois são usadas repetidamente. Usar *assembly* para refazer uma rotina usada, uma ou duas vezes, pode causar um aumento desprezível de velocidade, mas refazer uma rotina usada muitas vezes pode aumentar a velocidade significativamente. Por exemplo, observe o procedimento a seguir:

```
procedure ABC;
var
   t: integer;

begin
   inic;
   for t:=0 to 1000 do begin
      fase1;
      fase2;
      if t=10 then fase3;
   end;
   tchau;
end;
```

Reprogramar **inic** e **tchau** pode não afetar sensivelmente a velocidade deste procedimento, pois eles são executados só uma vez. Tanto **fase1** quanto **fase2** são executados 1.000 vezes, e reprogramá-los certamente terá um grande efeito na velocidade; **fase3** está dentro

do *loop*, mas só é executado uma vez, portanto, reprogramá-lo provavelmente não valha a pena.

Com um planejamento cuidadoso, você pode aumentar bastante a velocidade de um programa, refazendo só uns poucos subprogramas em *assembly*.

INTERFACE COM O SISTEMA OPERACIONAL

Muitos programas escritos em Turbo Pascal têm de trabalhar a nível do sistema operacional. Às vezes é necessário usar funções especiais do sistema operacional não disponíveis em Turbo Pascal. Por isso, o Turbo Pascal facilita o acesso a rotinas de baixo nível do sistema operacional.

Vários sistemas operacionais podem ser usados com Turbo Pascal:

- PC-DOS ou MS-DOS
- CP/M
- CP/M-86

Todos eles têm um conjunto de funções que os programas usam para tarefas como abrir arquivos, receber/enviar caracteres de/para o console, alocar memória etc. O modo de acesso a estas rotinas varia de sistema para sistema, mas a tendência geral é usar o conceito de tabela de desvios (*jump table*). Num sistema operacional como CP/M, rotinas do sistema são executadas através de uma instrução CALL para uma certa área da memória, com o código da função desejada num registrador. No PC-DOS, uma interrupção de software é usada. Nos dois casos, uma tabela de desvios leva à função propriamente dita. A Figura 5-3 mostra como o sistema operacional e sua tabela de desvios podem estar dispostos na memória.

Não é possível tratar dos vários sistemas operacionais aqui. Este capítulo concentra-se no PC-DOS, por ser o mais usado. Entretanto, a técnica geral é a mesma nos outros sistemas.

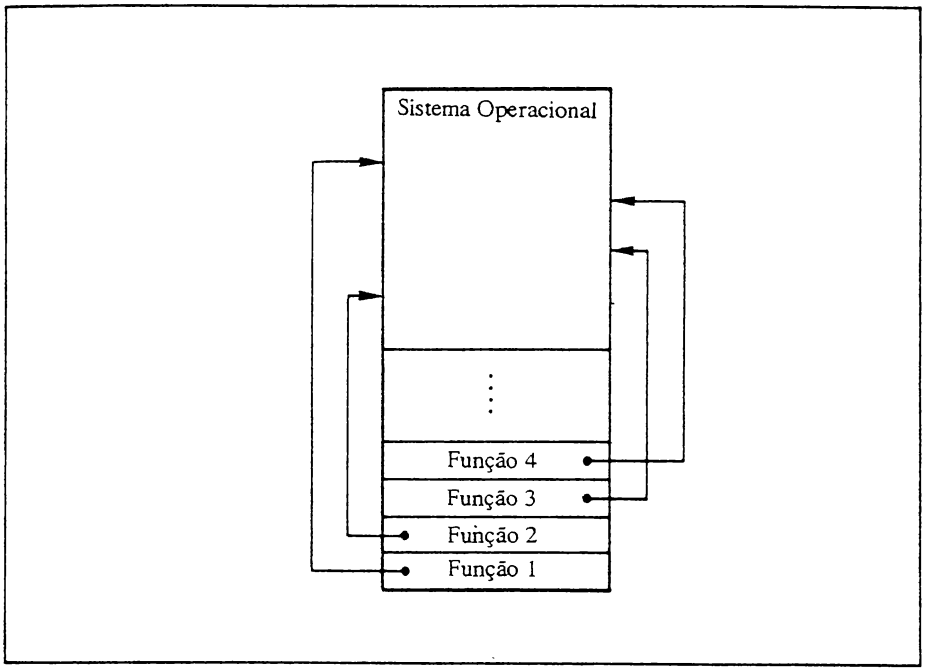


Figura 5-3 Um sistema operacional e sua tabela de desvios.

ACESSO A ROTINAS DO SISTEMA EM PC-DOS

No PC-DOS, as rotinas do sistema são acessadas por meio de interrupções de *software*. Cada interrupção acessa um grupo de funções; o conteúdo do registrador AH determina qual. Se outros parâmetros são necessários, eles são passados através dos registradores AL, BX, CX e DX. O PC-DOS é dividido em BIOS (*Basic I/O System* – sistema básico de E/S) e DOS (*Disk Operating System* – sistema operacional de disco). O BIOS fornece as rotinas de nível mais baixo que o DOS usa em suas rotinas. Há alguma sobreposição de funções, mas ambos são acessados do mesmo modo, basicamente. Segue uma lista de interrupções:

Interrupção	Função
05h	Utilitário de impressão da tela
10h	E/S de vídeo
11h	Listagem da configuração
12h	Tamanho da memória
13h	E/S de disco
14h	E/S da porta serial
15h	Controle do cassete
16h	Teclado
17h	Impressora
18h	BASIC residente
19h	Rotina de *boot*
1Ah	Hora e data

Para uma explicação mais detalhada, consulte o *IBM Technical Reference Manual.*

(*Nota do Tradutor:* Infelizmente, nenhum dos fabricantes nacionais fornece uma documentação tão detalhada quanto a IBM. E em alguns casos, os BIOS nacionais não se comportam de acordo com o BIOS padrão do IBM PC.)

Cada uma destas interrupções está associada a um número de opções que dependem do valor de AH quando a chamada ocorre. A Tabela 5-1 mostra uma lista parcial das opções disponíveis para algumas destas interrupções. Há duas maneiras de acessar as funções da Tabela 5-1. A primeira é através de um procedimento padrão do Turbo Pascal chamado **MsDos** (para o PC-DOS ou MS-DOS). O segundo método é via *assembly*. (*Nota do Revisor Técnico*: Há um terceiro método, através do procedimento **Intr** do Turbo Pascal. Este procedimento, ao contrário do **MsDos,** não se limita a produzir uma interrupção 21h, permitindo acesso a todas as rotinas do BIOS.)

Tabela 5-1 Rotinas do sistema acessadas via interrupções

E/S de vídeo – Interrupção 10h

Registrador AH	Função
0	Define modo de vídeo. AL = 0 : 40x25 P&B 1 : 40x25 cor 2 : 80x25 P&B 3 : 80x25 cor

(*continua na página seguinte*)

(continuação)

4 . 320x200 gráfica cor
5 : 320x200 gráfica monocromática
6 : 640x200 gráfica monocromática

1 Define linhas de pixel do cursor.
CH : bits 0–4 indicam linha de início
bits 5–7 em zero
CL : bits 0–4 indicam linha de fim
bits 5–7 em zero

2 Posiciona cursor.
DH : linha
DL : coluna
BH : número da página de vídeo

3 Verifica posição do cursor.
BH : número da página de vídeo
Retorna:
DH : linha
DL : coluna
CX : modo

4 Lê posição da caneta ótica.
Retorna:
AH = 0 : botão da caneta não pressionado
1 : botão da caneta pressionado
DH : linha
DL : coluna
CH : linha da tela gráfica (0–199)
BX : coluna da tela gráfica (0–319/639)

5 Define página ativa de vídeo.
AL : número da página (0–7)

6 *Scroll* para cima.
AL : número de linhas (0 = todas)
CH : linha do canto superior esquerdo
CL : coluna do canto superior esquerdo
DH : linha do canto inferior direito
DL : coluna do canto inferior direito
BH : atributo para limpar linhas

(continua na página seguinte)

(continuação)

7 *Scroll* para baixo.
(mesmos registradores da rotina 6)

8 Lê caractere na posição do cursor.
BH : número da página de vídeo
Retorna:
AL : caractere lido
AH : atributo do caractere lido

9 Escreve caractere e atributo na posição do cursor.
AL : caractere
BH : número da página de vídeo
BL : atributo do caractere
CX : número de posições a preencher

10 Escreve caractere na posição do cursor.
AL : caractere
BH : número da página de vídeo
CX : número de posições a preencher

11 Define palheta de cores (*palette*).
BH : número da palheta
BL : cor na palheta

12 Acende um pixel.
DX : linha
CX : coluna
AL : cor

13 Verifica um pixel.
DX : linha
CX : coluna
Retorna:
AL : cor

14 Escreve caractere e avança o cursor.
AL : caractere
BL : cor
BH : número da página de vídeo

15 Verifica estado do vídeo.
Retorna:
AL : número do modo

(continua na página seguinte)

(continuação)

AH : número de colunas
BH : número da página de vídeo ativa

Lista configuração – Interrupção 11h

Retorna:
AX : descrição do equipamento
bit 0 = 1 : um ou mais acionadores de disco
bit 1 : sem função
bits 2,3 : RAM presente na placa-mãe; 11 = 64k
bits 4,5 : modo de vídeo inicial;
01 = 40x25 cor
10 = 80x25 cor
11 = 80x25 monocromático
bits 6,7 : número de *drives;* 0 = 1 *drive*
bit 8 = 0 : *chip* de DMA instalado
bits 9, 10, 11 : número de portas RS-232
bit 12 = 1 : saída para *joystick* instalada
bit 13 = 1 : impressora serial (só no PCjr)
bit 14, 15 : número de impressoras

Tamanho da memória – Interrupção 12h

Retorna:
AX : número de parágrafos e RAM instalados no sistema

E/S de disco – BIOS – Interrupção 13h

Registrador AH	Função
0	Reseta sistema de disco.
1	Verifica estado do acionador. Retorna: AL : (v. *IBM Technical Reference Manual)*
2	Lê setores para memória. DL : número do acionador DH : número do cabeçote CH : número da trilha CL : número do setor AL : número de setores a serem lidos

(continua na página seguinte)

(continuação)

ES:BX : endereço do *buffer*
Retorna:
AL : número de setores lidos
AH : status (0 = ok)

3 Escreve setores no disco.
(mesmos registradores da rotina 2)

4 Verifica um setor.
(mesmos registradores da rotina 2)

5 Formata uma trilha.
DL : número do acionador
DH : número do cabeçote
CH : número da trilha
ES:BX : endereço do buffer

E/S do teclado – BIOS – Interrupção 16h

Registrador AH Função

0 Lê código de varredura.
Retorna:
AH : código de varredura da tecla
AL : código do caractere

1 Verifica estado do *buffer*.
Retorna:
ZF = 1 : não há caracteres no *buffer*
0 : há pelo menos um caractere no *buffer* ; o primeiro está no registrador AX (v. rotina 0)

2 Verifica estado do teclado.
(v. *IBM Technical Reference Manual*)

E/S da impressora – BIOS – Interrupção 17h

Registrador AH Função

0 Imprime um caractere.
AL : caractere
DX : número da impressora

(continua na página seguinte)

(continuação)

Retorna:
AH : status

1 Aciona impressora.
DX : número da impressora
Retorna:
AH : status

2 Verifica estado da impressora.
DX : número da impressora
Retorna:
AH : status

Rotinas de alto nível – DOS – Interrupção 21h (lista parcial)

Registrador AH	Função
1	Lê caractere do teclado. Retorna: AL : caractere
2	Coloca caractere na tela. DL : caractere
3	Lê caractere da porta assíncrona. Retorna: AL : caractere
4	Envia caractere para porta assíncrona. DL : caractere
5	Imprime caractere no dispositivo LST:. DL : caractere
7	Lê caractere do teclado sem mostrar no vídeo. AL : caractere
B	Verifica estado do teclado. AL = FF : tecla pressionada 0 : tecla não pressionada
D	Reseta acionador de disco.

(continua na página seguinte)

(continuação)

E — Define acionador de disco ativo.
DL = número do acionador (0=A, 1=B...)

11 (4E até V.2.xx) — Procura arquivo no disco.
DX : endereço do FCB
Nome do arquivo deve estar no endereço de transferência para disco (v. rotina 1A)
Retorna:
AL = 0 : arquivo encontrado
FF : arquivo não-encontrado

12 (4F até V2.xx) — Acha nova ocorrência do nome de arquivo.
(mesmos registradores da rotina 11)

1A — Define endereço de transferência para disco (*disk transfer address*).
DX : endereço

2A — Lê data do sistema.
Retorna:
CX : ano (1980–2099)
DH : mês (1–12)
DL : dia (1–31)

2B — Define data do sistema.
CX : ano (1980–2099)
DH : mês (1–12)
DL : dia (1–31)

2C — Lê hora do sistema.
Retorna:
CH : hora (0–23)
CL : minutos (0–59)
DH : segundos (0–59)
DL : centésimos de segundo (0–99)

2D — Define hora do sistema.
CH : hora (0–23)
CL : minutos (0–59)
DH : segundos (0–59)
DL : centésimos de segundo (0–99)

USO DO PROCEDIMENTO MsDos

O procedimento **MsDos** provoca uma interrupção de software número 21h para acessar uma das funções de nível mais alto do DOS. Uma chamada tem a forma geral:

MsDos(*registradores*)

onde *registradores* é um registro definido de uma destas maneiras:

```
reg_record = record
          AX,BX,CX,DX,BP,SI,DI,DS,ES,flags: integer;
end;

reg_byte = record
        AL,AH,BL,BH,DL,DH: byte;
        BP,SI,DI,DS,ES,flags: integer;
end;
```

A segunda definição é usada quando é preciso manipular bytes. Você pode misturar os tipos **byte** e **integer** para representar os registradores, ou usar um registro variante com **case**. Escolha a definição mais adequada para as suas necessidades.

No restante deste capítulo, estaremos desenvolvendo procedimentos que duplicam funções já existentes em Turbo Pascal. Isto se deve a três fatores. Primeiro, Turbo Pascal já inclui quase tudo em termos de rotina do sistema operacional para a maior parte das aplicações. Segundo, é importante ilustrar tão detalhadamente quanto possível como o interfaceamento ocorre, até que possamos lidar com situações especiais. Terceiro, os exemplos demonstram com maior profundidade como as funções e procedimentos funcionam no Turbo Pascal.

Aqui está um exemplo simples. A função a seguir determina se uma tecla foi digitada. Ela é similar à função **Keypressed** do Turbo Pascal. Esta função, **Tecl**, resulta **TRUE** se uma tecla tiver sido pressionada, e **FALSE** no caso inverso. Ela usa a interrupção 21h, número $B, como mostrado aqui. Lembre-se: números hexadecimais devem ser precedidos de $, que informa ao compilador que um número hexadecimal virá em seguida. O programa imprime pontos na tela até uma tecla ser pressionada.

```
program Kb;  {espera que o teclado seja acionado}
{$C-}
function Tecl: boolean;  {específico do PCDOS}
type
   RegByte = record
        AL,AH,BL,BH,DL,DH: byte;
        BP,SI,DI,DS,ES,flags: integer;
   end;
```

```
var
   registro: RegByte;
begin
   registro.AH:=$B;
   MsDos(registro);
   if registro.AL=0 then Tecl:=FALSE
   else Tecl:=TRUE;
end;

begin
   repeat
      Write('.');
   until Tecl;
end.
```

Note que o resto dos registradores não teve de ser acionado nesta chamada, pois apenas o número da função, $B, era necessário. Geralmente, se um registrador não é usado numa chamada, então a ele não precisa ser atribuído um valor.

A diretiva **$C-** do compilador é usada neste programa para impedir a verificação de CTRL-C, CTRL-S e CTRL-Q. Se a verificação não fosse impedida, estas teclas não seriam passadas ao programa. Elas seriam interceptadas pelo sistema de apoio em tempo de execução do Turbo Pascal. CTRL-C abortaria o programa, CTRL-S suspenderia a execução temporariamente, e CTRL-Q recomeçaria.

INTERFACEAMENTO COM BIOS E DOS VIA ASSEMBLY

Suponha que você deseje mudar o modo da tela durante a execução de um programa. Os 16 modos possíveis da tela do PC são mostrados na tabela a seguir:

Modo	Tipo	Dimensões	Placas
0	Texto, P&B	40 x 25	CGA,EGA
1	Texto, 16 cores	40 x 25	CGA,EGA
2	Texto, P&B	80 x 25	CGA,EGA
3	Texto, 16 cores	80 x 25	CGA,EGA
4	Gráfico, 4 cores	320 x 200	CGA,EGA
5	Gráfico, 4 tons de cinza	320 x 200	CGA,EGA
6	Gráfico, monocromático	640 x 200	CGA,EGA
7	Texto, P&B	80 x 25	Monochrome
8	Gráfico, 16 cores	160 x 200	PCjr
9	Gráfico, 16 cores	320 x 200	PCjr
10	Gráfico, 4 cores	640 x 200	PCjr
	ou Gráfico, 16 cores	640 x 200	EGA

Modo	Tipo	Dimensões	Placas
13	Gráfico, 16 cores	320 x 200	EGA
14	Gráfico, 16 cores	640 x 200	EGA
15	Gráfico, 4 cores	640 x 350	EGA

O procedimento **modo**, mostrado a seguir, executa uma chamada número 1 do BIOS para mudar a tela para o modo gráfico de 640 x 200, escrever a mensagem **oi**, e esperar o usuário teclar RETURN. Depois disto, a tela retorna para o modo de texto colorido de 80 x 25. (Este programa só funciona num IBM PC ou compatível com placa CGA.)

```
program ModoGrafico;

procedure mode(ModeSet: integer); external  mode.com';

begin
   mode(6);
   WriteLn('Oi!'); Read;
   mode(3);
end.
```

O procedimento externo **modo**, em *assembly*, é o seguinte:

```
; este procedimento muda o modo da tela
; com base em um argumento inteiro.
code    segment 'code'
        assume cs:code
mode    proc near   ; tem que ser 'near' porque o Turbo 3.0 usa
                    ; este modo de enderecamento do 8088
        push bp
        mov bp,sp

;muda o modo
        mov ax,[bp]+4
        mov ah,0    ; muda o modo
        int 010h    ; chamada de bios
;limpa e sai
        pop bp
        ret 2
mode    endp
code    ends
        end
```

Agora, um procedimento para limpar a tela, o **Limpatela:**

```
program LimpaTela;

procedure Limpa; external 'clr.com';

begin
   WriteLn('Pressione uma tecla para limpar a tela.');
   ReadLn;
   clr;
   WriteLn('Tela limpa.');
end.
```

O procedimento externo em *assembly*, **lt**, aparece a seguir:

```
;esta rotina limpa a tela usando a interrupcao padrao
;numero 6 do BIOS.
cseg segment 'code'
         assume cs:cseg

clr proc near
;salva os registradores usados
         push ax
         push bx
         push cx
         push dx
         mov cx,0    ; start at 0,0
         mov dh,24   ; end at row 24
         mov dl,79   ; column 79
         mov ah,6    ; set scroll option
         mov al,0    ; clear screen
         mov bh,7
         int 10h
;recupera os registradores e sai
         pop dx
         pop cx
         pop bx
         pop ax
         ret
clr endp
cseg ends
         end
```

Outro exemplo de interfaceamento com o BIOS através de *assembly* é o procedimento **xy**, que posiciona o cursor nas coordenadas x e y dadas. Este procedimento é semelhante ao procedimento **GotoXY** encontrado no Turbo Pascal. Para o IBM PC, 0,0 é o canto superior esquerdo da tela.

```
program Coordenadas;

var
   t: integer;

procedure xy(x,y: integer); external 'xy.com';
```

```
begin
   for t:=0 to 24 do
   begin
      xy(t,t);
      Write(t);
   end;
end.
```

Abaixo o procedimento externo em *assembly:*

```
code    segment 'code'
        assume cs:code
xy      proc near   ; must be near because Turbo uses small
                    ; 8088 model
        push bp
        mov bp,sp

; get first parm
        mov dh,[bp]+4 ; get x
        mov dl,[bp]+6 ; get y
        mov ah,2      ; tell bios to go there
        mov bh,0      ; page number
        int 10h    ;
;
        pop bp
        ret 4
xy      endp
code    ends
        end
```

USANDO OS CÓDIGOS DE VARREDURA DO TECLADO DO PC

Uma das experiências mais frustrantes para quem se inicia no IBM PC é tentar usar as teclas de movimentação do cursor (setas, home, PgUp etc.) ou as teclas de função nos seus programas. Estas teclas não geram um byte como as outras. Ao pressionar uma tecla, o PC gera um valor de 16 bits, chamado código de varredura (*scan code*). O código de varredura consiste em um byte menos significativo, que é o código ASCII (se houver) da tecla pressionada, e um byte mais significativo, contendo a posição da tecla no teclado. Para a maior parte das teclas, este código é convertido num número ASCII de 8 bits pelo sistema operacional. Mas, para as teclas de função e de movimentação do cursor, isto não ocorre pois não há um código para estas teclas (o código do caractere é zero). Isto significa que você tem de usar o código de posição para determinar que tecla foi pressionada. A rotina do número 1 do DOS para leitura do teclado não permite que você leia as teclas especiais.

O modo mais fácil de acessar estas teclas, usando Turbo Pascal, é escrevendo uma rotina em *assembly* para invocar a interrupção 16h e ler o código de varredura. (*Nota do Revisor Técnico:* Mais fácil ainda é usar o procedimento predefinido **intr** para acionar a interrupção 16h.)

```
;esta rotina retorna um valor de 16 bits do teclado
;O byte mais baixo e' um caracter ASCII ou 0.
;Se for 0, o byte mais alto contem um codigo
;de varredura
code    segment 'code'
        assume cs:code
scan    proc near    ; must be near because Turbo uses small
                     ; 8088 model

        push bp
        mov bp,sp

;get fisrt parm
        mov ah,0
        int 16h
        mov [bx+2],ax  ; valor
;restore and exit
        pop bp
        ret 2
scan    endp
code    ends
        end
```

Após a chamada, o código de varredura e o código do caractere estão no registrador AX. No caso de interrupção 16h, função 0, AH indica a posição e AL indica o caractere.

O importante no uso da rotina **scan** é saber que quando uma tecla especial é pressionada, o código do caractere é 0. Neste caso, você identifica a tecla pelo código de posição. O uso de **scan**, para cuidar de toda entrada via teclado, requer que a rotina que faz a chamada tome decisões baseadas no conteúdo de AH e AL. Eis um programa curto que ilustra um modo de fazê-lo:

```
program Flecha;

var
   t: integer;

function scan: integer; external  scan.com ;

begin
   repeat
      t:=scan;
      if Lo(t)=0 then WriteLn('O codigo de varredura e''', Hi(t))
      else WriteLn(Chr(lo(t)));
   until Chr(lo(t))='q';
end.
```

As funções predefinidas **Hi** e **Lo** são usadas para acessar as duas metades do valor de 16 bits devolvido por **scan**. Também é usada a função **Chr** para converter o número resultante em caractere.

O *IBM Technical Reference Manual* traz uma tabela completa dos códigos de varredura do IBM PC. Você também pode descobrir estes códigos, experimentando com um programa como o anteriormente mostrado (este método é mais divertido). Para ajudá-lo a testar o programa, aqui vão alguns códigos de varredura:

Seta esquerda	75
Seta direita	77
Seta para cima	72
Seta para baixo	80

A utilização plena das teclas de função exige a criação de rotinas de entrada de dados especiais, ignorando o **read** do Pascal. Isto é uma pena, mas é o único meio. A recompensa é permitir ao usuário usar o teclado inteiro.

PENSAMENTOS FINAIS SOBRE INTERFACEAMENTO COM O SISTEMA OPERACIONAL

Este capítulo apenas tocou de leve nas possibilidades de uso criativo dos recursos de um computador. A fim de integrar seu programa mais completamente com sistema operacional, você deve ter acesso a informações que descrevam todas as funções com detalhe.

Há várias vantagens em se usar as funções do sistema operacional. A primeira é que elas fazem uso de características especiais do seu computador, tornando seu programa mais profissional na aparência e no desempenho. Segundo, ignorando algumas funções do Turbo Pascal e buscando suas equivalentes no sistema, às vezes leva a um programa mais compacto e rápido. Terceiro, você tem acesso a funções não disponíveis pelo Turbo Pascal.

Entretanto, usar o sistema neste nível tem um preço. Você está construindo uma armadilha para si próprio quando começa a usar rotinas do sistema em vez de rotinas definidas pelo Turbo Pascal, pois seu programa não rodará em outra máquina. Você também poderá ficar dependente de uma certa versão do sistema operacional ou do compilador Turbo Pascal, o que criará problemas na hora de distribuir seus programas. Só você pode decidir se e quando seus programas devem estar à mercê destes fatores externos.

CAPÍTULO 6

ESTATÍSTICA

A maioria das pessoas que possuem ou têm acesso freqüente a um computador, em algum ponto, utiliza-o para executar *análise estatística*. Essa análise pode tomar a forma de monitoração ou tentativa de previsão de alteração de preços de ações de uma carteira, estudar testes clínicos para estabelecer limites seguros para uma nova droga, ou mesmo fornecer a média de passes errados para um time da segunda divisão. O ramo da matemática que lida com condensação, manipulação e extrapolação de dados é chamado *estatística*.

Como disciplina, a análise estatística é bastante jovem. Ela desenvolveu-se no século XVIII como uma conseqüência do estudo dos jogos de azar. De fato, probabilidade e estatística estão proximamente relacionadas. A análise estatística moderna começou por volta da virada do século, quando tornou-se possível amostrar e trabalhar com grandes conjuntos de dados. O computador tornou possível correlacionar e manipular rapidamente volumes de dados ainda maiores e converter esses dados em forma prontamente utilizável. Hoje, devido à sempre crescente massa de informações, criada e utilizada pelo governo e pela mídia, todos os aspectos da vida são ornados com calhamaços de informações estatísticas. É quase impossível ouvir rádio, assistir o noticiário da TV ou ler um artigo de jornal sem ser informado de alguma estatística.

Embora o Turbo Pascal não tenha sido planejado especificamente para programação estatística, ele se adapta bastante bem ao serviço. Chega até mesmo a oferecer uma flexibilidade não encontrada nas linguagens comerciais mais comuns como COBOL ou BASIC. Uma vantagem do Turbo Pascal sobre o COBOL, é a velocidade e facilidade com que os programas de Turbo Pascal podem utilizar as funções gráficas do

sistema para produzir diagramas e gráficos de dados. Além disso, as rotinas matemáticas do Turbo Pascal são muito mais rápidas que aquelas normalmente encontradas em BASIC interpretativo.

Este capítulo enfoca vários conceitos de estatística, incluindo:

- a média;
- a mediana;
- o desvio padrão;
- a equação de regressão (**linha** de melhor ajuste);
- o coeficiente de correlação.

Ele também explora algumas técnicas gráficas simples.

AMOSTRAS, POPULAÇÕES, DISTRIBUIÇÕES E VARIÁVEIS

Antes de usar estatística, você deve entender alguns conceitos-chaves. A informação estatística origina-se a partir da tomada de uma *amostra* de pontos de dados específicos seguida da elaboração de generalizações sobre eles. Cada amostra vem da *população*, que consiste em todas as ocorrências possíveis para a situação em estudo. Por exemplo, se você quisesse medir a produção anual de uma fábrica de caixas, usando apenas os números de produção das quartas-feiras e generalizando a partir deles, então sua amostra consistiria no valor de um ano de quartas-feiras tomadas da propulsão maior da produção de cada dia do ano.

É possível que a amostra iguale a população se for completa. No caso da fábrica de caixas, sua amostra igualaria a população se você usasse os verdadeiros números de produção – cinco dias por semana durante todo o ano. Quando a amostra é menor que a população, sempre há lugar para erro; entretanto, para a maioria dos casos, você pode determinar a probabilidade de ocorrência desse erro. Este capítulo pressupõe que a amostra é igual à população, portanto ele não cobre a questão dos erros de amostragem.

Para projeções eleitorais e enquetes de opinião, uma amostra proporcionalmente pequena é usada para projetar informações sobre a população como um todo. Por exemplo, você poderia utilizar informações estatísticas sobre o índice Bovespa para fazer uma inferência do mercado de ações em geral. Claro que a validade dessas conclusões varia muito. Em outros usos da estatística, para facilitar a manipulação, uma amostra que iguale

ou aproximadamente iguale a população é usada para sintetizar um grande conjunto de números. Por exemplo, uma junta de educação geralmente relata o ponto médio das notas de uma classe, em vez da nota individual de cada aluno.

As estatísticas são afetadas pelo modo como os eventos são distribuídos na população. Das várias distribuições comuns na natureza, a mais importante (e a única usada neste capítulo) é a *curva de distribuição normal*, ou a familiar *curva em forma de sino*, mostrada na Figura 6-1. Como sugere o gráfico da Figura 6-1, os elementos em distribuição normal são encontrados principalmente no meio. De fato, a curva é completamente simétrica em torno desse pico que também é a média de todos os elementos. Quanto mais longe do centro em qualquer direção da curva, menos elementos haverá. Muitas situações da vida real possuem distribuição normal.

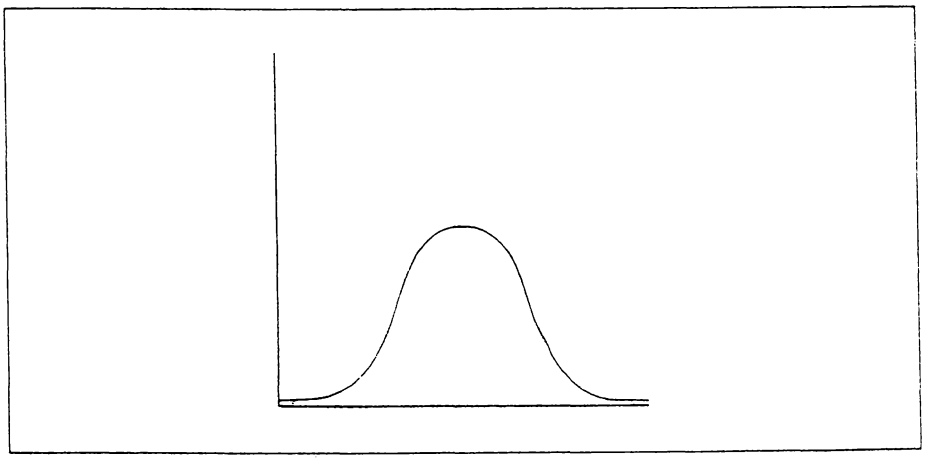


Figura 6-1 A curva de distribuição normal.

Em qualquer processo estatístico, há sempre uma *variável independente*, que é o número em estudo, e uma *variável dependente*, que é o fator que determina a variável independente. Este capítulo usa o *tempo* – incremento gradual da passagem de eventos – como variável dependente. Por exemplo, no exame de uma carteira de ações, você poderá desejar ver o movimento de estoque em base diária. Você estaria, portanto, preocupado com a mudança dos preços das ações em um dado período de tempo, não efetivamente com a data de calendário de cada preço.

Ao longo deste capítulo, serão desenvolvidas funções estatísticas individuais que serão montadas e um único programa gerenciado por menu. Você pode utilizar esse

programa para a execução de uma ampla gama de anâlises estatísticas, assim como para a representação de informações na tela.

Sempre que os elementos de uma amostra são discutidos, serão chamados de D e serão indexados de 1 a N, onde N é o número do último elemento.

A ESTATÍSTICA BÁSICA

Três valores importantes formam a base de muitas análises estatísticas e também são úteis individualmente. Eles são a *média,* a *mediana,* e a *moda.*

A MÉDIA

A média, ou média aritmética, é a mais comum de todas as estatísticas. Esse único valor pode ser usado para representar uma série de dados – a média pode ser chamada de "centro de gravidade" da série. Para computar a média, todos os elementos da amostra são somados e o resultado é dividido pelo número total de elementos. Por exemplo, a soma da série

1 2 3 4 5 6 7 8 9 10

é igual a 55. Quando o número é dividido pelo número de elementos na amostra, que é 10, a média é 5,5.

A fórmula geral para achar a média é

$$M = \frac{D_1 + D_2 + D_3 + \ldots + D_N}{N}$$

ou

$$M = \frac{1}{N} \sum_{i=1}^{N} D_i$$

O símbolo sigma indica a soma de todos os elementos entre 1 e *N*.

Da maneira como as funções estatísticas são desenvolvidas em Turbo Pascal, você deve admitir que todos os dados estão armazenados em uma matriz de números de ponto-flutuante de um tipo definido pelo usuário, chamado **Matriz**, e que o número de elementos da amostra é conhecido. Todas as funções e procedimentos utilizam a matriz **data** para armazenar a amostra e a variável **num** para o número de elementos. A seguinte função computa a média de uma matriz **num** de números reais e retorna a média:

```
function Media(dado: Matriz; num: integer): real;
var
   t: integer;
   med: real;

begin
   med:=0;
   for t:=1 to num do med:=med+dado[t];
   Media:=med/num;
end;  {Media}
```

Por exemplo, se você chamar **Media** com uma matriz de 10 elementos que contenha os números de 1 a 10, então ela retornará o resultado 5,5.

A MEDIANA

A mediana de uma amostra é o valor do meio, baseado na ordem de magnitude. Por exemplo, no conjunto amostra

1 2 3 4 5 6 7 8 9

5 é a mediana, porque está no meio. No conjunto

1 2 3 4 5 6 7 8 9 10

você poderia usar 5 ou 6 como mediana. Em uma amostra bem ordenada que tenha distribuição normal, a média e a mediana são muito similares. Entretanto, à medida que a amostra se afasta da curva de distribuição normal, a diferença entre a mediana e a média aumenta. O cálculo da mediana de uma amostra consiste simplesmente em ordenar a amostra em ordem crescente e selecionar o elemento do meio, que é indexado como *N*/2.

A função **Mediana,** aqui mostrada, retorna o valor do elemento do meio de uma amostra. Uma versão modificada de *QuickSort* (desenvolvido no Capítulo 2) é usada para ordenar a matriz de dados.

```
procedure QuickSort (var item: Matriz; conta: integer);
   procedure qs (l,r: integer; var it: Matriz);
      var
      i,j: integer;
      x,y: Dado;
      begin
         i:=l; j:=r;
         x:=it[(l+r) div 2];
         repeat
            while it[i] < x do i:=i+1;
            while x < it[j] do j:=j-1;
            if i <= j then
            begin
               y:=it[i];
               it[i]:=it[j];
               it[j]:=y;
               i:=i+1; j:=j-1;
            end;
         until i > j;
         if l < j then qs(l,j,it);
         if l < r then qs(i,r,it);
      end;
begin
     qs(1,conta,item);
end; {quicksort}

function Mediana (dado: Matriz; num: integer): real;
var
   dtemp: Matriz;
   t: integer;

begin
   for t:=1 to num do dtemp[t]:=data[t];  {copia daods para ordenacao}
   QuickSort(dtemp,num);
   Median:=dtemp[num div 2];   {elemento central}
end;  {Mediana}
```

A MODA

A moda de uma amostra é o valor do elemento de ocorrência mais freqüente. Por exemplo, na série

1 2 3 3 4 5 6 6 6 7 8 9

a moda seria 6 pois ele ocorre três vezes. Pode haver mais de uma moda, por exemplo, a amostra

10 20 30 30 40 50 60 60 70

tem duas modas – 30 e 60 – pois ambas ocorrem duas vezes.

A seguinte função, **Moda**, retorna a moda de uma amostra. Se houver mais de uma moda, ele retornará a última encontrada.

```
function Moda(dado: Matriz; num: integer): real;
var
   t,w,conta,conta_velho: integer;
   md,md_velha: real;

begin
   md_velha:=0; conta_velho:=0;
   for t:=1 to num do
   begin
      md:=dado[t];
      conta:=1;
      for w:=t+1 to num do
         if md=dado[w] then conta:=conta+1;
      if conta > conta_velho then
      begin
         md_velha:=md;
         conta_velho:=conta;
      end;
   end;
   Moda:=md_velha;
end;  {Moda}
```

USANDO A MÉDIA, A MEDIANA, E A MODA

A média, a mediana e a moda possuem o mesmo propósito: fornecer um valor que seja uma condensação de todos os valores da amostra. Entretanto, cada uma representa a amostra de um modo diferente. A média da amostra geralmente é o valor mais útil. Devido ao fato de usar todos os valores na sua computação, a média reflete todos os elementos da amostra. A principal desvantagem da média é sua sensibilidade a um valor extremo. Por exemplo, em uma companhia imaginária chamada Widget Inc., o salário do proprietário é $ 100.000 por ano, enquanto o salário de cada um dos nove empregados é $ 10.000. O salário médio da Widget é $ 19.500, mas esse número não representa fielmente a situação!

Em casos como o de dispersão de salários na Widget, usa-se a moda em vez da média. A moda dos salários em Widget é $ 10.000 – um número que reflete mais acuradamente a situação real. Entretanto, a moda também pode ser enganadora. Considere uma companhia de carros que fabrica carros em cinco cores diferentes. Em uma dada semana, ela fabricou

100 carros verdes
100 carros laranja
150 carros azuis
200 carros pretos
190 carros brancos

Aqui, a moda da amostra é preta, pois foram fabricados 200 carros pretos, o que é mais do que qualquer outra cor. Entretanto, seria errado sugerir que a companhia de carros fabrica principalmente carros pretos.

A mediana é interessante porque sua validade é baseada na *esperança* de que a amostra reflita a distribuição normal. Por exemplo, se a amostra for

1 2 3 4 5 6 7 8 9 10

então a mediana é 5 ou 6 e a média é 5,5. Neste caso, a mediana e a média são similares. Entretanto, na amostra

1 1 1 1 5 100 100 100 100

a mediana ainda é 5, mas a média é em torno de 46.

Em certas circunstâncias, não se pode contar nem com a média, nem com a moda, nem com a mediana para se obter um valor significativo. Isso nos leva a dois dos números mais importantes em estatística – a *variância* e o *desvio padrão*.

A VARIÂNCIA E O DESVIO PADRÃO

Embora o resumo mononumerário (como a média, moda e mediana) seja muito conveniente, ele pode ser enganador. Pensando um pouco nesse problema, você verá que a causa da dificuldade não é o valor em si, mas o fato dele não conter nenhuma informação sobre as variações dos dados. Por exemplo, na amostra

1 1 1 1 9 9 9 9

a média é 5; entretanto, não há nenhum elemento na amostra que seja próximo de 5. O que você provavelmente gostaria de saber é quão próximo está cada elemento da média; em outras palavras, qual a variabilidade dos dados. Saber a variabilidade dos dados o ajudará

a interpretar melhor a média, a mediana e a moda. Você pode determinar a variabilidade de uma amostra, computando a variância e o desvio padrão.

A variância e sua raiz quadrada, o desvio padrão, são números que expressam o desvio médio da média da amostra. Dos dois, o desvio padrão é o mais importante. Ele pode ser pensado como a média das distâncias que os elementos estão da média da amostra. A variância é computada como:

$$V = \frac{1}{N} \sum_{i=1}^{N} (D_i - M)^2$$

onde N é o número de elementos na amostra, e M é a média da amostra. Você deve elevar ao quadrado a diferença entre a média e cada elemento para produzir apenas números positivos. Se os números não fossem elevados ao quadrado, eles sempre tenderiam a 0.

A variância produzida por essa fórmula, V, é de valor limitado pois é difícil de entender. Entretanto, sua raiz quadrada, o desvio padrão, é o número que você está realmente procurando. O desvio padrão é conseguido primeiro achando-se a variância e depois tomando-se sua raiz quadrada:

$$dp = \sqrt{\frac{1}{N} \sum_{i=1}^{N} (D_i - M)^2}$$

onde N é o número de elementos na amostra e M é a média da amostra.

Como exemplo, para a amostra

11 20 40 30 99 30 50

você computa a variância do seguinte modo:

	D	*D–M*	$(D–M)^2$
	11	–29	–841
	20	–20	–400
	40	0	0
	30	–10	–100
	99	59	3481
	30	–10	–100
	50	10	100
Soma	280	0	5022
Média	40	0	717,42

Aqui, a média das diferenças elevadas ao quadrado é 717,42. Para derivar o desvio padrão, você simplesmente toma a raiz quadrada desse número; o resultado é aproximadamente 26,78. Para interpretar o desvio padrão, lembre-se de que ele é igual à *distância média que os elementos mantêm da média da amostra*.

O desvio padrão nos diz com que aproximação a média representa a totalidade da amostra. Se você possuísse uma fábrica de doces, e seu supervisor de produção relatasse que a média da produção do mês passado foi 2.500 mas o desvio padrão foi 2.000, você saberia que a linha de produção precisa de uma supervisão melhor!

Se sua amostra segue uma distribuição normal padrão, então cerca de 68% da amostra estará dentro de um desvio padrão da média, e cerca de 95% estará dentro de dois desvios padrões.

A seguinte função calcula o desvio padrão de uma dada amostra:

```
function Dev_Pad (dado: Matriz; num: integer): real;
var
   t: integer;
   pad,med: real;

begin
   med:=Media(dado,num);
   pad:=0;
   for t:=1 to num do
      pad:=pad+((dado[t]-med)*(dado[t]-med));

   pad:=pad/num;
   Dev_Pad:=Sqrt(pad);

end;  {Dev_Pad}
```

REPRESENTAÇÃO SIMPLES NA TELA

A vantagem em utilizar gráficos com estatísticas é que, juntos, eles podem veicular o significado clara e acuradamente. Um gráfico também mostra, num relance, como a amostra está realmente distribuída e de que forma os dados variam. Esta discussão está limitada a gráficos bidimensionais, que utilizam o sistema de coordenadas X-Y. (A criação de gráficos tridimensionais é uma disciplina em si mesma e está além do objetivo deste livro.)

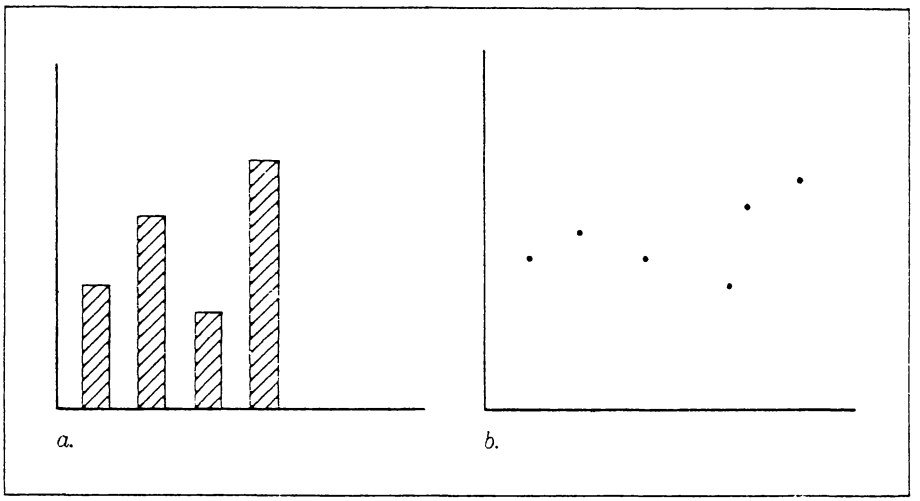


Figura 6-2 Exemplos de um gráfico de barras (*a*) e de pontos (*b*).

Há duas formas comuns de gráficos tridimensionais: o *gráfico de barras* e o *gráfico de pontos*. O gráfico de barras utiliza barras maciças para representar a magnitude de cada elemento; o gráfico de pontos utiliza um ponto único por elemento, localizado nas coordenadas X e Y. A Figura 6-2 mostra um exemplo de cada.

O gráfico de barras geralmente é usado com um conjunto relativamente pequeno de informações, como o produto nacional bruto nos últimos dez anos ou a produção percentual de uma fábrica em base mensal. O gráfico de pontos é geralmente usado para mostrar um grande número de pontos de dados, como o preço diário de estoque através do ano. Uma modificação do gráfico de dados, que conecta os pontos de dados com uma linha cheia, é útil para traçar projeções.

Aqui está um procedimento simples que cria um gráfico de barras no IBM PC. Através de algumas funções externas do Turbo Pascal, a função usa as capacidades

gráficas internas do IBM PC. Os procedimentos **GraphColorMode, Palette, Draw** e **Plot** são fornecidos pelo Turbo Pascal.

```
procedure Graf_Simples(dado Matriz; num: integer);
{Desenha um grafico de barra usando os graficos do IBM-PC.}
var
    t,incr: integer;
    a: real;
    car: char;

begin
    GraphColorMode;
    Palette(0);
    GotoXY(1,25); WriteLn(min);
    GotoXY(1,1); WriteLn(max);
    GotoXY(38,25); WriteLn(num);
    Draw(0,190,200,190,1);  {base}
    for t:=1 to num do
    begin
        a:=dado[t];
        y:=trunc(a);
        incr:=10;
        x:=((t-1)*incr)+20;
        Draw(x,190,x,190-y,2);
    end;
    Read(car);
    TextMode;

end;  {Graf_Simples}
```

No IBM PC, o modo de gráfico colorido, de máxima resolução, oferece uma resolução de 320 x 320 e é instalado pela chamada de **GraphColorMode**. O procedimento **GotoXY** coloca o cursor na posição X-Y desejada. O procedimento **Draw** tem a forma geral

draw(*X_inicial, Y_inicial, X_final, Y_final, cor*);

onde todos os valores devem ser inteiros. Consulte o *Manual do Usuário do Turbo Pascal* para informações adicionais.

Esta rotina simples tem uma limitação séria – ela pressupõe que todos os dados estarão entre 0 e 199, pois os números que podem ser usados para chamar a função gráfica **Draw** estão entre 0 e 199. Esta limitação é boa apenas na improvável hipótese de todos os elementos de dados estarem neste intervalo. Para fazer com que a rotina manipule unidades arbitrariamente dimensionadas, você deve normalizar os dados antes de traçar o gráfico, para reescaloná-los e ajustá-los à escala exigida. O processo de normalização envolve a determinação de uma razão entre a escala real dos valores de dados e o limite físico de resolução da tela. Cada elemento de dado pode então ser multiplicado por essa

razão para produzir um número que se ajuste à escala da tela. A fórmula para fazer isso no eixo Y do PC é

$$Y' = Y * \frac{200}{(máx - mín)}$$

onde Y' é o valor usado quando da chamada da função que desenha o gráfico. Essa mesma fórmula pode ser usada para aumentar a escala quando o intervalo coberto pelos dados é muito pequeno. Isso resulta em um gráfico que preenche a tela tanto quanto possível.

O procedimento **Graf_Bar** escalona os eixos X e Y e traça um gráfico de barras de até 300 elementos. Admite-se que o eixo X seja o tempo e tenha incrementos de uma unidade. Geralmente, o procedimento de normalização encontra o maior e o menor valor na amostra e então calcula sua diferença. Esse número, que representa a distância entre o mínimo e o máximo, é usado para dividir a resolução da tela. Para o IBM PC, esse número

```
procedure Graf_Bar(dado: Matriz; num: integer);
{desenha um grafico de barra usando os graficos do IBM PC }
var
   x,y,max,min,t,incr: integer;
   a,norm,div: real;
   ch: char;

begin
   GraphColorMode;
   Palette(0);
   {acha min e max para normalizar os dados}
   max:=GetMax(dado,num);
   min:=GetMin(dado,num);
   if min > 0 then min=0;
   div:=max-min;
   norm:=190/div;
   GotoXY(1,25); WriteLn(min);
   GotoXY(1,1); WriteLn(max);
   GotoXY(38,25); WriteLn(num);
   for t:=1 to 19 do Plot(0,t*10,1);
   Draw(0,190,320,190,1);
   for t:=1 to num do
   begin
      a:=dado[t]-min;
      a:=a*norm;
      y:=trunc(a);
      incr:=300 div num;
      x:=((t-1)*incr)+20;
      Draw(x,190,x,190-y,2);
   end;
   Read(ch);
   TextMode;

end;  {Graf_Bar}
```

é 190 para o eixo X (porque você precisa de espaço para margens e linha-base). A razão é então usada para converter os dados da amostra na escala adequada.

Essa versão também imprime uma escala ao longo do eixo Y, onde cada intervalo representa 1/20 da diferença entre os valores máximo e mínimo. A Figura 6-3 dá uma amostra de **Graf_Bar** com 20 elementos. De modo algum **Graf_Bar** fornece todas as características que você poderia desejar, mas ele representará uma amostra simples, acuradamente. Você pode achar fácil expandi-lo para que se ajuste às suas necessidades.

Apenas uma ligeira modificação em **Graf_Bar** é necessária para se fazer um procedimento que trace um gráfico de pontos. A maior alteração modifica a função **Draw** para uma que trace apenas um ponto. Essa função é chamada **Plot**. Sua forma geral é

Plot *(x, y, cor);*

onde *x*, *y* e *cor* são inteiros. O procedimento **Graf_Pontos** é mostrado a seguir.

```
procedure Graf_Pontos(dado: Matriz; num,ymin,ymax,xmax: integer);
var
   x,y,t,incr: integer;
   a,norm,div: real;

begin
   {acha max e min para normalizar os dados}
   if ymin > 0 then ymin:=0;
   div:=ymax-ymin;
   norm:=190/div;
   GotoXY(1,25); WriteLn(ymin);
   GotoXY(1,1); WriteLn(ymax);
   GotoXY(38,25); WriteLn(xmax);
   for t:=1 to 19 do Plot(0,t*10,1);
   Draw(0,190,320,190,1);
   for t:=1 to num do
   begin
      a:=dado[t]-ymin;
      a:=a*norm;
      y:=trunc(a);
      incr:=300 div xmax;
      x:=((t-1)*incr)+20;
      Plot(x,190-y,2);
   end;
end;  {Graf_Pontos}
```

Em **Graf_Pontos**, os valores de dados mínimo e máximo são passados para o procedimento, em vez de serem computados pelo procedimento como em **Graf_Bar**. Isso o torna capaz de representar séries de dados múltiplos na mesma tela, sem ter de mudar de escala, criando um efeito de sobreposição. A Figura 6-4, que mostra um gráfico de pontos de uma amostra de 30 elementos de dados, foi produzida por este procedimento.

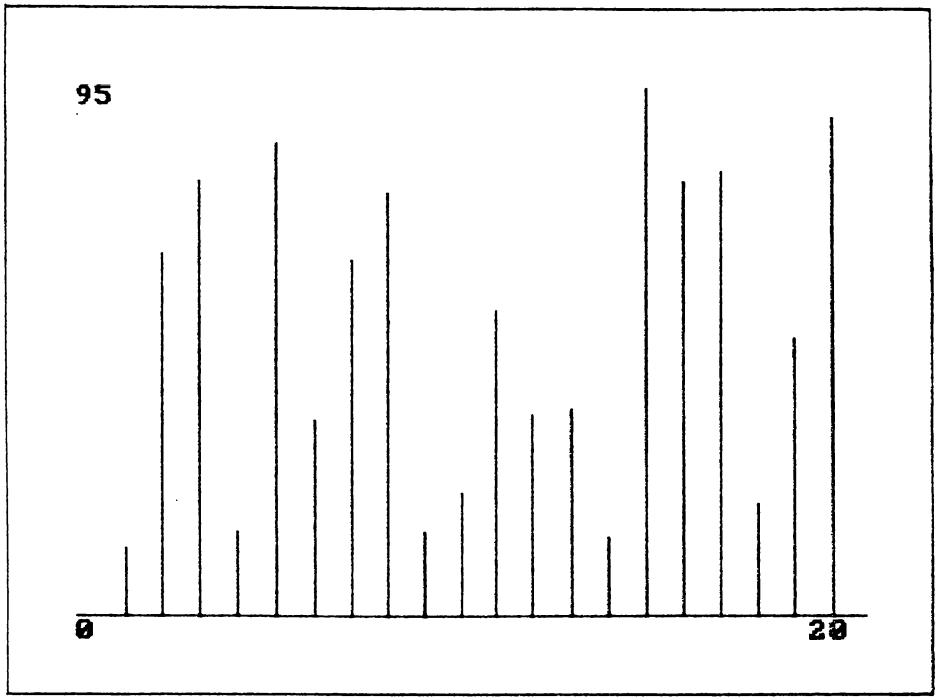


Figura 6-3 Um gráfico de barras de uma amostra, produzido por **Graf_Bar.**

PROJEÇÕES E A EQUAÇÃO DE REGRESSÃO

Informação estatística é freqüentemente usada para realizar "adivinhações informadas" sobre o futuro. Mesmo que todo mundo saiba que o passado não prediz necessariamente o futuro e que existem exceções a toda regra, dados históricos são usados desse modo. Muito freqüentemente, passado e presente tendem a continuar no futuro. Quando isso acontece, você pode tentar determinar valores específicos em pontos futuros no tempo. Esse processo é chamado *projeção,* ou *análise de tendência.*

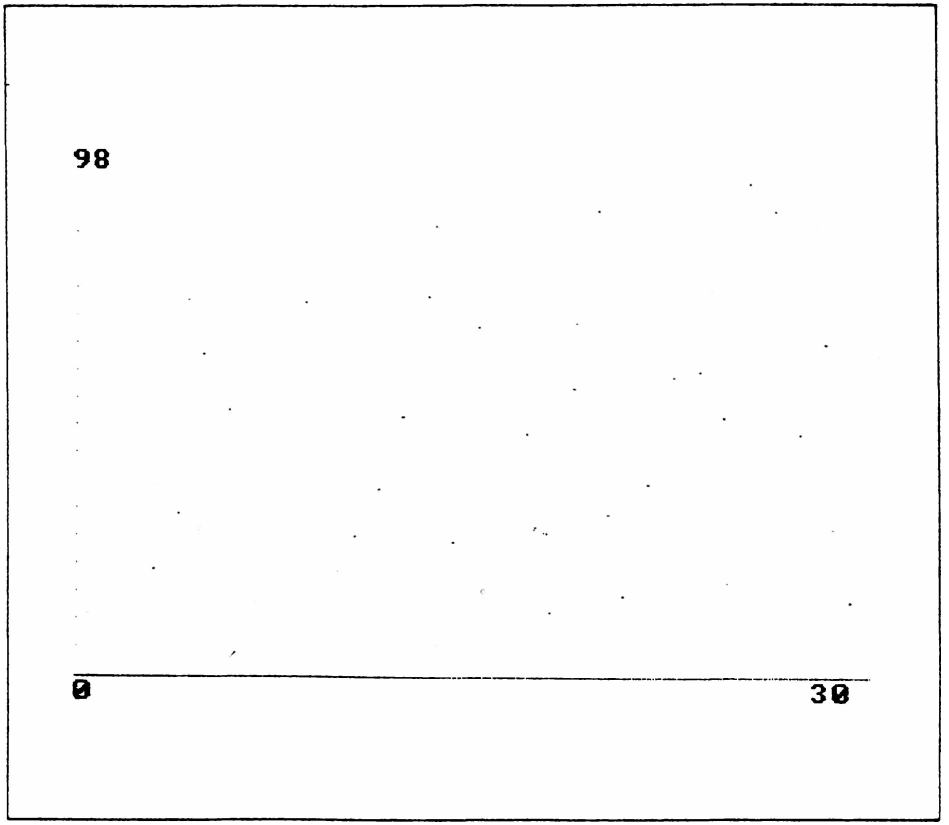


Figura 6-4 Um gráfico de pontos de uma amostra, produzido por **Graf_Pontos.**

Por exemplo, considere um estudo fictício de dez anos de expectativa de vida, que tivesse coletado os seguintes dados:

Ano	**Longevidade**
1970	69
1971	70
1972	72
1973	68
1974	73
1975	71
1976	75

Ano	**Longevidade**
1977	74
1978	78
1979	77

Você poderia se perguntar se aqui há realmente uma tendência. Se houver, você poderá querer saber em que direção ela está indo. Finalmente, se de fato houver uma tendência, você poderá imaginar qual será a expectativa de vida, digamos, em 1985.

Primeiro olhe para o gráfico de barras e para o gráfico de pontos desses dados, como mostrado na Figura 6-5. Pelo exame desses gráficos, você pode concluir que os períodos de vida em geral estão se tornando mais longos. Além disso, se você colocasse uma régua nos gráficos que se ajustasse aos dados e desenhasse uma linha que se estendesse até 1985, você poderia projetar que a expectativa de vida seria em torno de 82 anos. Entretanto, mesmo se você se sentisse confiante na sua análise intuitiva, provavelmente preferiria utilizar um método mais formal e exato para projetar tendências de expectativa de vida.

Dado um conjunto de dados históricos, a melhor maneira de fazer projeções é encontrar a reta *de melhor ajuste* em relação aos dados. Foi isso que você fez com a régua. Uma reta de melhor ajuste representaria com maior aproximação cada ponto dos dados e sua tendência. Embora alguns ou até mesmo todos os pontos reais possam não estar na reta, a linha os representa melhor. A validade da reta é baseada na proximidade com que vêm os pontos de dados da amostra.

Uma reta em espaço bidimensional possui a seguinte equação básica:

$$Y = a + b\,X$$

onde Y é a variável independente, X a variável dependente, a é o intercepto de Y, e b é a inclinação da reta. Portanto, para determinar a reta que melhor se ajuste a uma amostra, você deve determinar a e b.

Vários métodos podem ser usados para determinar os valores de a e b, mas o mais comum (e geralmente o melhor) é chamado de método do menor quadrado. Ele tenta minimizar a distância entre os verdadeiros pontos de dados e a reta. O método envolve duas etapas. A primeira computa b, a inclinação da reta, a segunda encontra a, o intercepto de Y. Para encontrar, use a fórmula:

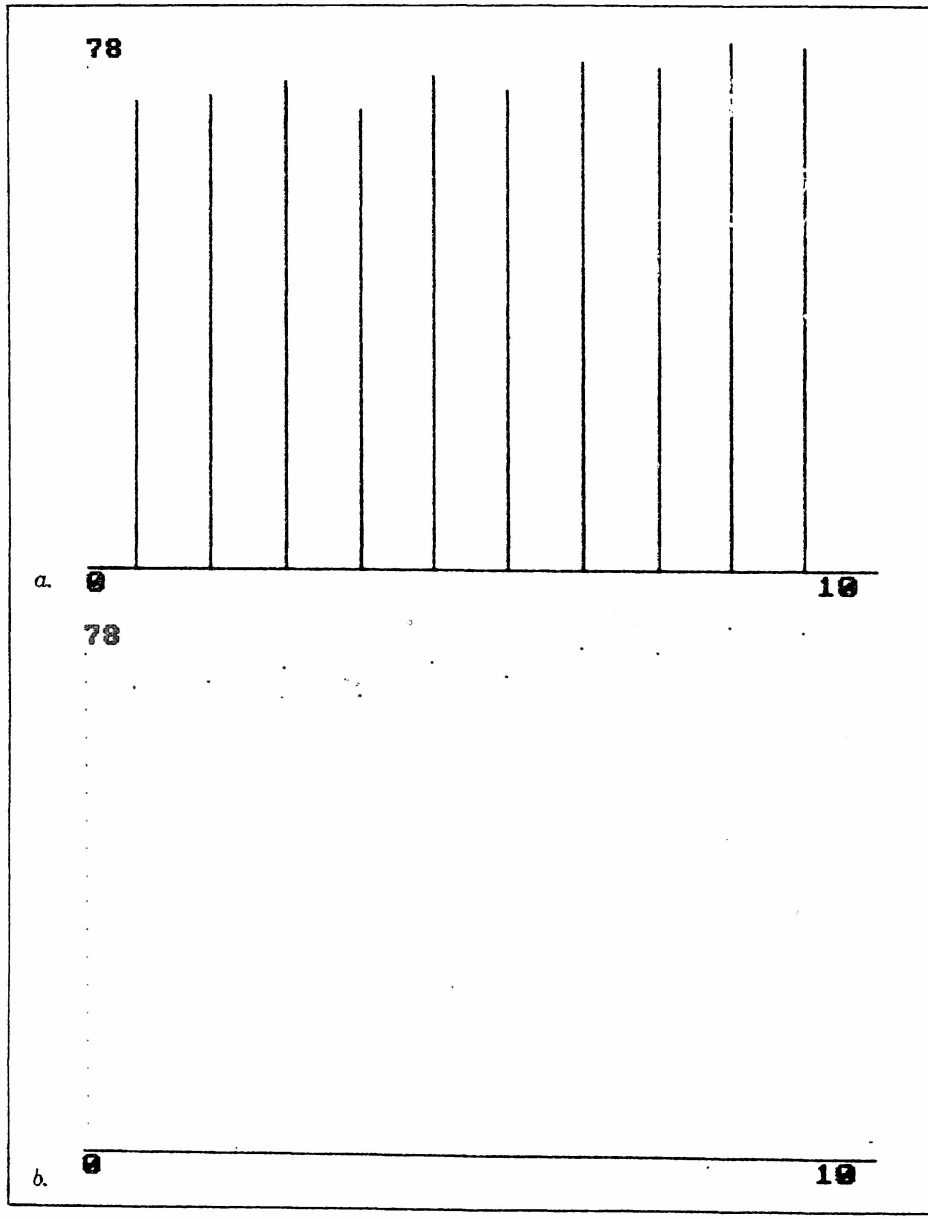


Figura 6-5 Gráfico de barras (*a*) e de pontos (*b*) da expectativa de vida.

$$b = \frac{\sum_{i=1}^{N} (X_i - M_x)(Y_i - M_y)}{\sum_{i=1}^{N} (X_i - M_x)^2}$$

onde M_x é a média da coordenada X e M_y é a média da coordenada Y. A dedução desta fórmula está além do propósito deste livro, mas tendo achado b você poderá usá-lo para computar a, como mostrado aqui:

$$a = M_x - bM_y$$

Depois de ter calculado a e b, você pode fazer com que X assuma qualquer valor e ache o de Y. Por exemplo, se você utilizar os dados de expectativa de vida, achará que a equação de regressão é mais ou menos essa:

$$Y = 67{,}46 + 0{,}95 * X$$

Portanto, para encontrar a expectativa de vida em 1985, que está 15 anos além de 1970, a fórmula será:

$$\begin{aligned} \textit{expectativa de vida} &= 67{,}46 + 0{,}95 * 15 \\ &\cong 82 \end{aligned}$$

Entretanto, mesmo com a reta de melhor ajuste para os dados, você ainda pode querer saber quanto a reta realmente se relaciona bem com os dados. Se a reta e os dados tiverem apenas uma ligeira correlação, então a reta de regressão é de pouca utilidade. Entretanto, se a reta se ajustar bem aos dados, então ela é um indicador muito mais válido. O modo mais comum de determinar e representar a correlação dos dados com a reta de regressão é calcular o coeficiente de correlação, um número entre 0 e 1. O coeficiente de correlação é, essencialmente, uma percentagem relacionada com a distância entre cada ponto e a reta. Se o coeficiente de correlação for 1, então os dados corresponderão perfeitamente à reta; ou seja, cada elemento da amostra também estará sobre a reta de regressão. Um coeficiente 0 significa que não existem pontos na amostra que realmente estejam na reta. A fórmula para encontrar o coeficiente de correlação *cor* está a seguir:

$$cor = \frac{\frac{1}{N}\sum_{i=1}^{N}(X_i - M_x)(Y_i - M_y)}{\sqrt{\frac{1}{N}\sum_{i=1}^{N}(X_i - M_x)^2}\ \sqrt{\frac{1}{N}\sum_{i=1}^{N}(Y_i - M_y)^2}}$$

Aqui M_x é a média da coordenada de X e M_y é a média da coordenada de Y. Geralmente, um valor 0,81 é considerado uma forte correlação. Isso indica que cerca de 66% dos dados estão sobre a reta de regressão. Para converter qualquer coeficiente de correlação em uma percentagem, simplesmente eleve-o ao quadrado.

Aqui está o procedimento **Regressão.** Ela utiliza os métodos recém-descritos para encontrar a equação de regressão e o coeficiente de correlação, e também faz um gráfico de pontos tanto dos dados de amostra quanto da reta.

```
procedure Regressao(dado: Matriz; num: integer);
{calcula a equacao de regressao e o coeficiente de
 correlacao - entao imprime os dados e a linha de regressao.}

var
   a,b,x_med,y_med,temp,temp2,cor: real;
   dado2: Matriz;
   t,min,max: integer;
   car: char;

begin
   {Calcula a media de x e y}
   y_med:=0; x_med:=0;
   for t:=1 to num do
   begin
      y_med:=y_med+dado[t];
      x_med:=x_med+t;  {porque x e' tempo}
   end;
   x_med:=x_med/num;
   y_med:=y_med/num;

   {calcula o fator b da equacao de regressao}
   temp:=0; temp2:=0;
   for t:=1 to num do
   begin
      temp:=temp+(dado[t]-y_med)*(t-x_med);
      temp2:=temp2+(t-x_med)*(t-x_med);
   end;
   b:=temp/temp2;

   {calcula a equacao de regressao}
   a:=y_med-(b*x_med);
```

```
    {calcula o coeficiente de correlacao}
    for t:=1 to num do dado2[t]:=t;  {copia os dados}
    cor:=temp/num;
    cor:=cor/(Dev_Pad(dado,num)*Dev_Pad(dado2,num));
    WriteLn('A equacao de regressao e'' : Y = ',a:15:5,'+',b:15:5,'* X');
    WriteLn('O coeficiente de correlacao e'' : ',cor:15:5);
    Write('Imprimir dados e linha de regressao? (S/N) ');
    Read(car); WriteLn;
    car:=UpCase(car);
    if car <> 'N' then
    begin
       GraphColorMode;
       Palette(0);
       {desenha os graficos}
       for t:=1 to num*2 do dado2[t]:=a+(b*t);  {Matriz de regressao}
       min:=GetMin(dado,num)*2;
       max:=GetMax(dado,num)*2;
       ScatterPlot(dado,num,min,max,num*2);
       ScatterPlot(dado2,num*2,min,max,num*2);
       Read;
       TextMode;
    end;

end;  {Regressao}
```

Um gráfico de pontos tanto dos dados de amostra quanto da reta de regressão é mostrado na Figura 6-6. O ponto importante a recordar, quando do uso de projeções como esta, é que o passado não prediz necessariamente o futuro – do contrário, não haveria graça!

FAZENDO UM PROGRAMA ESTATÍSTICO COMPLETO

Até aqui, este capítulo desenvolveu diversas funções que executam cálculos estatísticos em populações de variável única. Esta seção reúne as funções para formar um programa completo para analisar dados, imprimir gráficos de barras ou de pontos, e fazer projeções. Antes que você possa desenhar um programa completo, deve definir um registro para manter informações de dados das variáveis e algumas rotinas de apoio necessárias.

Primeiro você precisa de uma matriz que detenha as informações das amostras. Você pode usar uma unidimensional de ponto-flutuante chamada **dado**, de tamanho **MAX**. **MAX** é definida de modo que se ajuste à maior amostra que você necessite, que nesse caso será 100. As definições de tipo constante e os dados globais são mostrados aqui:

```
program Estatistica;

const
   MAX = 100;

type
   str80 = string[80];
   Item = real;
   Matriz = array[1..MAX] of Item;
var
    dado:Matriz;
    a,m,md,dpd:real;
    num:integer;
    car:char;
    arq:file of Item;
```

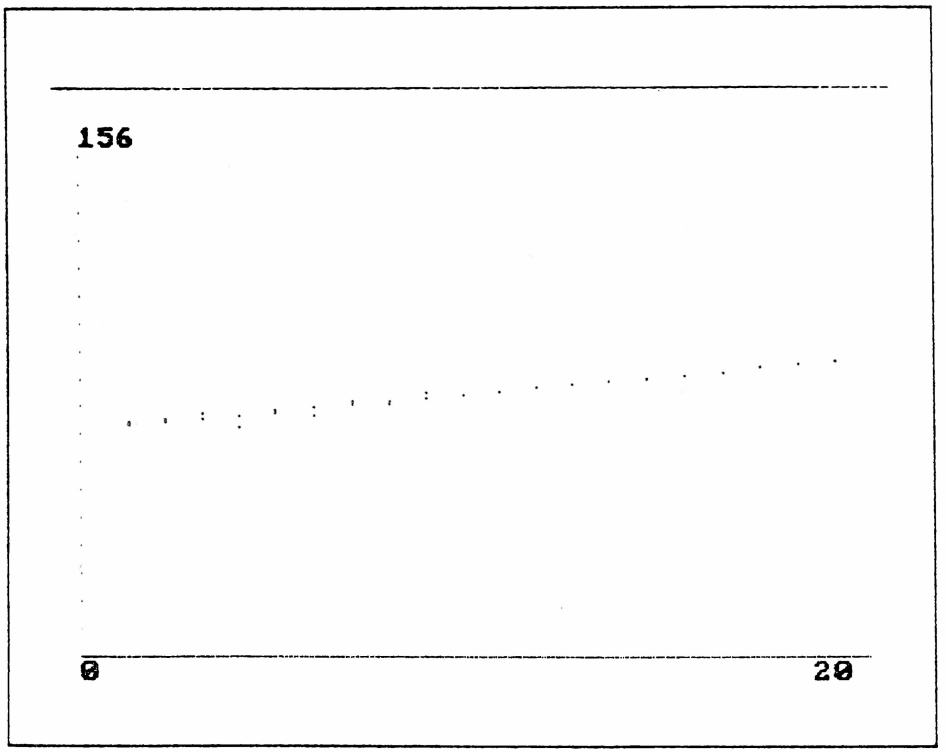


Figura 6-6 Reta de regressão para expectativa de vida.

Além das funções estatísticas já desenvolvidas, você também precisa de rotinas para gravar e carregar dados. A rotina **Grava** também deve armazenar o número de elementos da amostra e a rotina **Le** deve ler o número de volta.

```
procedure Grava(dado: Matriz; num:integer);
var
   t:integer;
   nome_arq:string[80];
   temp:real;

begin
   Write('Nome do arquivo: ');
   Read(nome_arq); WriteLn;
   Assign(arq,nome_arq);
   Rewrite(arq);
   temp:=num;   {muda o tipo}
   Write(arq,temp);
   for t:=1 to num do Write(arq,dado[t]);
   Close(arq);
end;  {Grava}

procedure Le;
var
   t:integer;
   nome_arq:string[80];
   temp:real;

begin
   Write('Nome do arquivo: ');
   Read(nome_arq); WriteLn;
   Assign(arq,nome_arq);
   Reset(arq);
   Read(arq,temp);
   num:=trunc(temp);
   for t:=1 to num do Read(arq,dado[t]);
   Close(arq)
end;  {Le}
```

Para sua conveniência, aqui está o programa de estatística completo:

```
program Estatistica;

const
   MAX = 100;

type
   str80 = string[80];
   Item = real;
   Matriz = array[1..MAX] of Item;
var
    dado:Matriz;
    a,m,md,dpd:real;
    num:integer;
    car:char;
    arq:file of Item;
```

```
procedure QuickSort (var dado: Matriz; conta: integer);
   procedure qs (l,r: integer; var it: Matriz);
      var
      i,j: integer;
      x,y: Item;
      begin
         i:=l; j:=r;
         x:=it[(l+r) div 2];
         repeat
            while it[i] < x do i:=i+1;
            while x < it[j] do j:=j-1;
            if i <= j then
            begin
               y:=it[i];
               it[i]:=it[j];
               it[j]:=y;
               i:=i+1; j:=j-1;
            end;
         until i > j;
         if l < j then qs(l,j,it);
         if l < r then qs(i,r,it);
      end;
begin
     qs(1,conta,dado);
end; {quicksort}

function Pronto(car: char;s:str80): boolean;
var
   t:integer;

begin
   Pronto:=FALSE;
   for t:=1 to length(s) do
      if s[t]=car then Pronto:=TRUE;
end;  {Pronto}

function Menu:char;
var
   car:char;

begin
   WriteLn;
   repeat
      WriteLn('Digitar dados');
      WriteLn('Mostrar dados');
      WriteLn('Estatisticas Basicas');
      WriteLn('Regressao e grafico de pontos');
      WriteLn('Barra');
      WriteLn('Gravar');
      WriteLn('Ler');
      WriteLn('Fim');
      WriteLn;
      Write('Escolha um (D, M, E, R, B, G, L, F): ');
      Read(car); WriteLn;
      car:=UpCase(car);
   until Pronto(car,'DMERBGLF');
   menu:=car;
end; {Menu}

procedure Mostra(dado:Matriz; num:integer);
var
   t:integer;

begin
   for t:=1 to num do WriteLn(t,':', dado[t]:15:5);
   WriteLn;
end;  {Mostra}
```

```
procedure Digitacao(var dado: Matriz);
var
   t:integer;

begin
   Write('Numero de items?: ');
   Read(num); WriteLn;
   for t:=1 to num do begin
      Write('Digite item',t,': ');
      Read(dado[t]); WriteLn;
   end;
end;  {Digitacao}

function Media(dado: Matriz; num: integer): real;
var
   t: integer;
   med: real;

begin
   med:=0;
   for t:=1 to num do med:=med+dado[t];
   Media:=med/num;
end;  {Media}

function Dev_Pad (dado: Matriz; num: integer): real;
var
   t: integer;
   pad,med: real;

begin
   med:=Media(dado,num);
   pad:=0;
   for t:=1 to num do
      pad:=pad+((dado[t]-med)*(dado[t]-med));

   pad:=pad/num;
   Dev_Pad:=Sqrt(pad);

end;  {Dev_Pad}

function Moda(dado: Matriz; num: integer): real;
var
   t,w,conta,conta_velho: integer;
   md,md_velha: real;

begin
   md_velha:=0; conta_velho:=0;
   for t:=1 to num do
   begin
      md:=dado[t];
      conta:=1;
      for w:=t+1 to num do
         if md=dado[w] then conta:=conta+1;
      if conta > conta_velho then
      begin
         md_velha:=md;
         conta_velho:=conta;
      end;
   end;
   Moda:=md_velha;
end;  {Moda}

function Mediana (dado: Matriz; num: integer): real;
var
   dtemp: Matriz;
   t: integer;
```

```
begin
   for t:=1 to num do dtemp[t]:=dado[t];  {copia daods para ordenacao}
   QuickSort(dtemp,num);
   Mediana:=dtemp[num div 2];   {elemento central}
end;  {Mediana}

function Maximo(dado: Matriz; num: integer): integer;
var
   t:integer;
   max:real;

begin
   max:=dado[1];
   for t:=2 to num do
      if dado[t] > max then max:=dado[t];
   Maximo:=trunc(max);
end;  {Maximo}

function Minimo(dado: Matriz; num: integer): integer;
var
   t:integer;
   min:real;

begin
   min:=dado[1];
   for t:=2 to num do
      if dado[t] > min then min:=dado[t];
   Minimo:=trunc(min);
end;  {Minimo}

procedure Graf_Bar(dado: Matriz; num: integer);
{desenha um grafico de barra usando os graficos do IBM PC }
var
   x,y,max,min,t,incr: integer;
   a,norm,dif: real;
   ch: char;

begin
   GraphColorMode;
   Palette(0);
   {acha min e max para normalizar os dados}
   max:=Maximo(dado,num);
   min:=Minimo(dado,num);
   if min > 0 then min:=0;
   dif:=max-min;
   norm:=190/dif;
   GotoXY(1,25); WriteLn(min);
   GotoXY(1,1); WriteLn(max);
   GotoXY(38,25); WriteLn(num);
   for t:=1 to 19 do Plot(0,t*10,1);
   Draw(0,190,320,190,1);
   for t:=1 to num do
   begin
      a:=dado[t]-min;
      a:=a*norm;
      y:=trunc(a);
      incr:=300 div num;
      x:=((t-1)*incr)+20;
      Draw(x,190,x,190-y,2);
   end;
   Read(ch);
   TextMode;

end;  {Graf_Bar}


procedure Graf_Pontos(dado: Matriz; num,ymin,ymax,xmax: integer);
var
   x,y,t,incr: integer;
   a,norm,dif: real;
```

```
begin
   {acha max e min para normalizar os dados}
   if ymin > 0 then ymin:=0;
   dif:=ymax-ymin;
   norm:=190/dif;
   GotoXY(1,25); WriteLn(ymin);
   GotoXY(1,1); WriteLn(ymax);
   GotoXY(38,25); WriteLn(xmax);
   for t:=1 to 19 do Plot(0,t*10,1);
   Draw(0,190,320,190,1);
   for t:=1 to num do
   begin
      a:=dado[t]-ymin;
      a:=a*norm;
      y:=trunc(a);
      incr:=300 div xmax;
      x:=((t-1)*incr)+20;
      Plot(x,190-y,2);
   end;
end;  {Graf_Pontos}

procedure Regressao(dado: Matriz; num: integer);
{calcula a equacao de regressao e o coeficiente de
 correlacao - entao imprime os dados e a linha de regressao.}

var
   a,b,x_med,y_med,temp,temp2,cor: real;
   dado2: Matriz;
   t,min,max: integer;
   car: char;

begin
   {Calcula a media de x e y}
   y_med:=0; x_med:=0;
   for t:=1 to num do
   begin
      y_med:=y_med+dado[t];
      x_med:=x_med+t;  {porque x e' tempo}
   end;
   x_med:=x_med/num;
   y_med:=y_med/num;

   {calcula o fator b da equacao de regressao}
   temp:=0; temp2:=0;
   for t:=1 to num do
   begin
      temp:=temp+(dado[t]-y_med)*(t-x_med);
      temp2:=temp2+(t-x_med)*(t-x_med);
   end;
   b:=temp/temp2;

   {calcula a equacao de regressao}
   a:=y_med-(b*x_med);

   {calcula o coeficiente de correlacao}
   for t:=1 to num do dado2[t]:=t;  {copia os dados}
   cor:=cor/(Dev_Pad(dado,num)*Dev_Pad(dado2,num));
   WriteLn('A equacao de regressao e'' : Y = ',a:15:5,'+',b:15:5,'* X');
   WriteLn('O coeficiente de correlacao e'' : ',cor:15:5);
   Write('Imprimir dados e linha de regressao? (S/N) ');
   Read(car); WriteLn;
   car:=UpCase(car);
   if car <> 'N' then
   begin
      GraphColorMode;
      Palette(0);
      {desenha os graficos}
      for t:=1 to num*2 do dado2[t]:=a+(b*t);  {Matriz de regressao}
```

```
      min:=Minimo(dado,num)*2;
      max:=Maximo(dado,num)*2;
      Graf_Pontos(dado,num,min,max,num*2);
      Graf_Pontos(dado2,num*2,min,max,num*2);
      Read;
      TextMode;
   end;

end;  {Regressao}

procedure Grava(dado: Matriz; num:integer);
var
   t:integer;
   nome_arq:string[80];
   temp:real;

begin
   Write('Nome do arquivo: ');
   Read(nome_arq); WriteLn;
   Assign(arq,nome_arq);
   Rewrite(arq);
   temp:=num;   {muda o tipo}
   Write(arq,temp);
   for t:=1 to num do Write(arq,dado[t]);
   Close(arq);
end;  {Grava}

procedure Le;
var
   t:integer;
   nome_arq:string[80];
   temp:real;

begin
   Write('Nome do arquivo: ');
   Read(nome_arq); WriteLn;
   Assign(arq,nome_arq);
   Reset(arq);
   Read(arq,temp);
   num:=trunc(temp);
   for t:=1 to num do Read(arq,dado[t]);
   Close(arq)
end;  {Le}


begin
   repeat
      car:=UpCase(menu);
      case car of
         'D': Digitacao(dado);
         'E': begin
                 a:=Media(dado,num);
                 m:=Mediana(dado,num);
                 dpd:=Dev_Pad(dado,num);
                 md:=Moda(dado,num);
                 WriteLn('Media: ',a:15:5);
                 WriteLn('Mediana: ',m:15:5);
                 WriteLn('Desvio Padrao: ',dpd:15:5);
                 WriteLn('Moda: ',md:15:5);
                 WriteLn;
              end;
         'M': Mostra(dado,num);
         'B': Graf_Bar(dado,num);
         'R': Regressao(dado,num);
         'G': Grava(dado,num);
         'L': Le;
      end;
   until car='F';
end.
```

USANDO O PROGRAMA DE ESTATÍSTICA

Para dar uma idéia de como você pode usar o programa estatístico, desenvolvido neste capítulo, aqui está uma análise simples do mercado de valores para a Widget, Inc. Como investidor, você estará tentando decidir se vale a pena investir na Widget através da compra de ações; se você deve vender (vendendo ações que você não tem, na esperança de uma queda rápida de preços, de modo a poder comprá-las mais tarde por um preço mais barato); ou se você deve investir em outro lugar.

Mês	Preço das Ações (Cz$)
1	10
2	10
3	11
4	9
5	8
6	8
7	9
8	10
9	10
10	13
11	11
12	11
13	11
14	11
15	12
16	13
17	14
18	16
19	17
20	15
21	15
22	16
23	14
24	16

Você deve, primeiro, descobrir se o preço da ação da Widget registrou uma tendência. Depois de introduzir os números, você encontrará as seguintes estatísticas básicas:

Média: 12,08	Desvio padrão: 2,68
Mediana: 11	Moda: 11

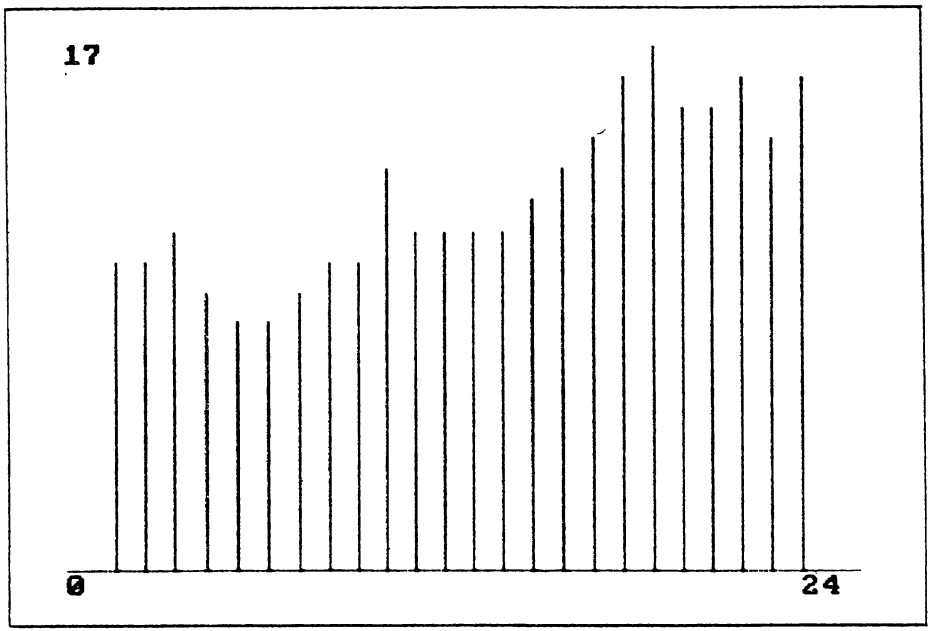


Figura 6-7 Gráficos e informações estatísticas sobre os lucros da Widget nos últimos 24 meses.

Em seguida, você deve traçar um gráfico de barras do preço da ação, como mostrado na Figura 6-7. Pode haver uma tendência, mas você deve executar uma análise de regressão formal. A equação de regressão é

$$Y = 7{,}90 + 0{,}33 * X$$

com um coeficiente de correlação de 0,86, ou de aproximadamente 74%. Isso é bastante bom – de fato, há uma tendência definida. A impressão de um gráfico de pontos, como mostrado na Figura 6-8, faz o crescimento dessa linha imediatamente aparente. Tais resultados poderiam fazer com que um investidor deixasse de lado quaisquer precauções e comprasse 1.000 ações o mais rápido possível.

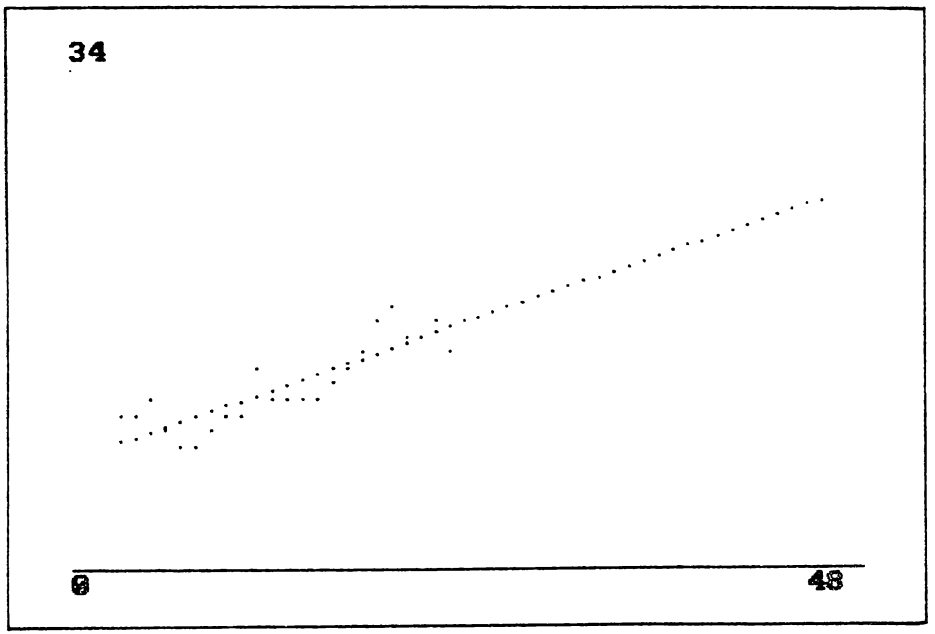


Figura 6-8 Gráfico de pontos mostrando o crescimento da Widget, Inc.

PENSAMENTOS FINAIS

A aplicação correta da análise estatística requer um entendimento geral de como os resultados se originam e o que eles significam. Como no exemplo da Widget, é fácil esquecer que os eventos passados não podem ser culpados por circunstâncias que poderiam afetar radicalmente o produto final. Confiança cega em evidências estatísticas pode causar resultados muito perturbadores. Estatísticas são usadas para refletir a natureza – mas não devemos esperar que a natureza reflita as estatísticas.

CAPÍTULO 7

CRIPTOGRAFIA E COMPRESSÃO DE DADOS

As pessoas que gostam de computadores e de programação, freqüentemente apreciam brincar com códigos e cifras. Talvez a razão para isso seja que todos os códigos envolvem algoritmos, assim como os programas. Ou talvez essas pessoas simplesmente tenham uma afinidade com coisas crípticas que a maioria das pessoas não possam entender. Todos os programadores parecem encontrar grande satisfação quando um não-programador olha para a listagem de um programa e diz alguma coisa como, "Meu Deus, isso realmente parece complicado!" Afinal de contas, o próprio ato de escrever um programa é chamado de "codificação".

Proximamente associado ao tópico de criptografia está o de *compressão de dados*. Compressão de dados significa compactar informação em um espaço menor do que o normalmente usado. Devido ao fato da compressão de dados poder representar um papel na criptografia e utilizar muitos dos mesmos princípios que esta, ela foi incluída neste capítulo.

A criptografia baseada em computadores é importante por duas razões principais. A mais óbvia é a necessidade de manter dados secretos seguros em sistemas compartilhados. Embora a proteção por senhas seja adequada para muitas situações, arquivos importantes e confidenciais são comumente codificados para fornecer um nível mais alto de proteção. A segunda utilidade de códigos baseados em computador é na transmissão de dados. Os códigos não são usados apenas para coisas como informações secretas governamentais, mas também por redes transmissoras para proteger transmissões céu-terra. Por serem tão complexos, esses procedimentos de codificação são rotineiramente feitos por computador.

Compressão de dados é comumente usada para aumentar a capacidade de armazenamento de dados de vários dispositivos armazenadores. Embora o custo dos dispositivos armazenadores de dados tenha caído muito nos últimos anos, sempre haverá necessidade de se colocar mais informações em áreas menores.

UMA CURTA HISTÓRIA DA CRIPTOGRAFIA

Embora ninguém saiba quando começou a escrita secreta, uma tabuinha cuneiforme feita por volta de 1500 a.C. contém um dos mais antigos exemplos conhecidos. Ela contém uma fórmula codificada para elaborar verniz para cerâmica. Os gregos já usavam códigos em 475 a.C., e a classe alta de Roma, durante o reinado de Júlio César freqüentemente usou cifras simples. Durante a Idade Média o interesse na criptografia (assim como em muitas outras investigações) decresceu, exceto entre os monges, que ocasionalmente a usavam. Com o advento da Renascença Italiana, a arte da criptografia floresceu novamente. No tempo de Luis XIV, na França, um código baseado em 587 chaves, selecionadas aleatoriamente, foi usado para mensagens governamentais.

No século XIX, dois fatores determinaram o progresso da criptografia. O primeiro foi as estórias de Edgar Allan Poe, como por exemplo *"The Gold Bug"*, onde o uso de mensagens codificadas excitava a imaginação de muitos leitores. O segundo foi a invenção do telefone e do código Morse. O código Morse foi a primeira representação binária (pontos e traços) do alfabeto a ter grande uso. Durante a I Guerra Mundial, várias nações construíram "máquinas codificadoras" mecânicas, que permitiam codificação e decodificação fácil de textos através do uso de cifras sofisticadas e complexas. Neste ponto, a história da criptografia muda, ligeiramente, para a história da quebra de códigos.

Antes de dispositivos passarem a ser usados para codificar e decodificar mensagens, as cifras complexas não eram usadas com freqüência, devido ao esforço que exigiam tanto para codificação quanto para decodificação. Daí, a maioria dos códigos podia ser quebrada em um período de tempo relativamente pequeno. Entretanto, a arte de decifração de códigos tornou-se muito mais difícil depois que as máquinas de código começaram a ser usadas. Embora os computadores modernos sejam capazes de decifrar aqueles códigos rapidamente, isto não diminui o talento de Herbert Yardley, ainda considerado o maior mestre decifrador de códigos de todos os tempos. Ele não só decifrou o código diplomático americano em 1915, nas suas horas de folga, como decifrou também o

código diplomático japonês em 1922 – mesmo sem saber japonês! Ele conseguiu essa façanha usando tabelas de freqüência da língua japonesa.

Na II Guerra Mundial, o melhor método para decifrar códigos era roubar a máquina codificadora do inimigo, evitando com isso o tedioso (mesmo que intelectualmente gratificante) processo de decifração de códigos. De fato, a posse, pelos aliados, de uma máquina codificadora contribuiu grandemente para o término da guerra.

Com o advento dos computadores – especialmente computadores multiusuários –, a necessidade de códigos seguros e indecifráveis tornou-se ainda mais importante. Não apenas os arquivos de computadores precisam permanecer secretos, como o próprio acesso ao computador tem de ser gerenciado e regulado. Numerosos métodos de criptografar arquivos de dados foram desenvolvidos, e o algoritmo *DES (Data Encryption Standard)*, aceito pelo *National Bureau of Standards* é, em geral, tido como seguro contra esforços de decifração de código. Entretanto, o DES é de aplicação muito difícil e pode não ser adequado a todas as situações.

TIPOS DE CIFRAS

Dos métodos de codificação mais tradicionais, há dois tipos básicos: *substituição* e *transposição*. Uma cifra de substituição troca um caractere por outro, mas mantém a mensagem na ordem adequada. Uma cifra de transposição mistura os caracteres de uma mensagem, de acordo com alguma regra. Esses métodos podem ser usados em qualquer nível de complexidade desejada, e podem até mesmo ser usados em conjunto. O computador digital adiciona uma terceira técnica de criptografia, chamada de *manipulação de bit*, que através de um algoritmo altera a representação computadorizada dos dados.

Todos os três métodos podem utilizar uma chave. Uma chave é uma seqüência de caracteres (*string*) necessária para decodificar uma mensagem. Não confunda a chave com o método, pois para decodificar a mensagem não basta conhecer a chave – o algoritmo de ciframento também tem de ser conhecido. A chave "personaliza" uma mensagem codificada, de modo que apenas aquelas pessoas que conhecem a chave podem decodificá-la, mesmo que o método usado para a codificação esteja acessível.

Dois termos que devem se tornar familiares para você são *original* e *cifra*. O original de uma mensagem é o texto que você pode ler; a cifra é a versão codificada.

Este capítulo apresenta métodos computadorizados que usam cada um dos três métodos básicos na codificação de arquivos-textos. Você verá diversos programas gerais que codificam e decodificam arquivos-textos. Com uma exceção, esses programas possuem uma função **Cifra** e outra **Decifra**. A função **Decifra** sempre reverte o processo **Cifra** usado para criar o texto cifrado. Por simplicidade, todas as letras em uma mensagem, mostradas em exemplos deste capítulo, são codificadas e decodificadas em letra maiúscula.

CIFRAS DE SUBSTITUIÇÃO

Uma das mais simples cifras de substituição desloca o alfabeto em uma determinada medida. Por exemplo, se cada letra fosse deslocada três posições, então

abcdefghijklmnopqrstuvwxyz

se tornaria

defghijklmnopqrstuvwxyzabc

Note que as letras a, b e c foram tiradas da frente e colocadas no fim. Para codificar uma mensagem utilizando este método, simplesmente substitua o alfabeto mudado pelo verdadeiro. Por exemplo, a mensagem

encontre me ao por do sol

fica

hqfrqxuh ph dr sru gr vro

O programa mostrado aqui permite que você codifique qualquer mensagem de texto, usando qualquer equivalência que você escolher.

```
program subs1;  {A simple substitution cipher }
type
   str80 = string[80];

var
   inf,outf:str80;
   start:integer;
   ch:char;
```

```
procedure Code(inf,outf:str80;start:integer);
var
   infile,outfile:file of char;
   ch:char;
   t:integer;

begin
   Assign(infile, inf);
   Reset(infile);
   Assign(outfile,outf);
   Rewrite(outfile);

   while not EOF(infile) do
   begin
      read(infile,ch);
      ch:=UpCase(ch);
      if (ch>='A') and (ch<='Z') then
      begin
         t:=Ord(ch)+start;
         if t>Ord('Z') then t:=t-26; {wrap around}
         ch:=Chr(t);
      end;
      Write(outfile,ch);
   end;
   WriteLn('file coded');
   Close(infile); Close(outfile);
end;  {Code}

procedure Decode(inf,outf:str80;start:integer);
var
   infile,outfile:file of char;
   ch:char;
   t:integer;
begin
   Assign(infile, inf);
   Reset(infile);
   Assign(outfile,outf);
   ReWrite(outfile);

   while not EOF(infile) do
   begin
      read(infile,ch);
      ch:=UpCase(ch);
      if (ch>='A') and (ch<='Z') then
      begin
         t:=Ord(ch)-start;
         if t<Ord('A') then t:=t+26; {wrap around}
         ch:=Chr(t);
      end;
      Write(outfile,ch);
   end;
   WriteLn('file decoded');
   Close(infile); Close(outfile);
end;  {Decode}

begin
   Write('enter input file: ');
   ReadLn(inf);
```

```
    Write('enter output file: ');
    ReadLn(outf);
    Write('starting position (1-26): ');
    ReadLn(start);
    Write('Code or Decode (C or D): ');
    ReadLn(ch);
    if UpCase(ch)='C' then Code(inf,outf,start)
    else if UpCase(ch)='D' then Decode(inf,outf,start)
end.
```

Embora uma cifra de substituição geralmente engane colegiais, não é adequada para a maioria das aplicações, pois pode ser facilmente quebrada. Além disso há apenas 26 substituições possíveis, e é fácil tentar todas em um curto período de tempo. Um melhoramento na cifra de substituição é utilizar um alfabeto aleatório em vez de uma substituição simples.

Um deslize da cifra de substituição simples é que ela mantém os espaços entre as palavras, o que a torna mais fácil de ser decifrada por quebradores de códigos. Um outro aperfeiçoamento seria codificar espaços. (Na verdade, toda a pontuação deveria ser codificada, mas para simplificar os exemplos não farão isso.) Por exemplo, você poderia mapear esta seqüência de caracteres randômica, contendo todas as letras do alfabeto e um espaço

abcdefghijklmnopqrstuvwxyz<espaço>

nesta outra seqüência de caracteres:

qazwsxedcrfvtgbyhnujm ikolp

Você pode se perguntar se o uso de um alfabeto aleatório traz alguma vantagem substancial na segurança de uma mensagem codificada, comparado a uma versão de substituição simples. A resposta é sim, pois há 26 fatorial (26!) maneiras de se arranjar um alfabeto, com o espaço, este número se torna 27 fatorial (27!). O fatorial de um número é aquele número vezes todos os números inteiros menores que ele, até 1. Por exemplo, 6! é 6*5*4*3*2*1, que é igual a 720. Portanto, 26! é um número muito grande.

O programa mostrado aqui é uma cifra de substituição melhorada, que utiliza o alfabeto aleatório mostrado acima. Se você codificar a mensagem

meet me at sunset

usando o programa de cifra de substituição melhorada, ele ficará assim:

tssjptspqjpumgusj

que definitivamente é um código mais difícil de ser decifrado.

```
program subs2;  {um cifrador de substituicao melhorado}
type
   str80 = string[80];

var
   arq_ent,arq_sai: str80;
   alfabeto,alfa_sub:str80;
   inicio: integer;
   car: char;

function Acha(alfabeto: str80; car: char): integer;
{esta funcao retorna um indice para o alfabeto}
var
   t: integer;

begin
   Acha:=-1;  {codigo de erro}
   for t:=1 to 27 do if car=alfabeto[t] then Acha:=t;
end;  {Acha}

function Letra(car: char): boolean;
{retorna TRUE se car for uma letra}
begin
   Letra:= (UpCase(car) >= 'A') and (UpCase(car) <= 'Z');
end;  {Letra}

procedure Cifra(arq_ent,arq_sai: str80);
var
   entrada,saida: file of char;
   car: char;

begin
   Assign(entrada,arq_ent);
   Reset(entrada);
   Assign(saida,arq_sai);
   Rewrite(saida);

   while not EOF(entrada) do
   begin
      read(entrada,car);
      car:=UpCase(car);
      if Letra(car) or (car=' ') then
      begin
         car:=alfa_sub[Acha(alfabeto,car)];  {acha substituicao}
      end;
      Write(saida,car);
   end;
   WriteLn('Arquivo cifrado.');
   Close(entrada); Close(saida);
end;  {Cifra}
```

```
procedure Decifra(arq_ent,arq_sai: str80);
var
   entrada,saida: file of char;
   car: char;

begin
   Assign(entrada,arq_ent);
   Reset(entrada);
   Assign(saida,arq_sai);
   ReWrite(saida);

   while not EOF(entrada) do
   begin
      read(entrada,car);
      car:=UpCase(car);
      if Letra(car) or (car=' ') then
      begin
         car:=alfabeto[Acha(alfa_sub,car)];  {troca pelo alfabeto real}
      end;
      Write(saida,car);
   end;
   WriteLn('Arquivo decifrado.');
   Close(entrada); Close(saida);
end;  {Decifra}

begin
   alfabeto:='ABCDEFGHIJKLMNOPQRSTUVWXYZ ';
   alfa_sub:='QWERTYU IOPASDFGHJKLZXCVBNM'; {alfabeto de substituicao}
   Write('Digite o nome do arquivo de entrada: ');
   ReadLn(arq_ent);
   Write('Digite o nome do arquivo de saida: ');
   ReadLn(arq_sai);
   Write('Cifra or Decifra (C or D): ');
   ReadLn(car);
   if UpCase(car)='C' then Cifra(arq_ent,arq_sai)
   else if UpCase(car)='D' then Decifra(arq_ent,arq_sai);
end.
```

Embora a decifração de código seja examinada mais tarde neste capítulo, você deve saber que mesmo esse código de substituição ainda pode ser facilmente decifrado através do uso de *tabela de freqüência* da língua inglesa, na qual a informação estatística do uso de cada letra está registrada. (Esse tipo de substituição é usado nos "criptogramas" próximos às palavras cruzadas, fornecidos por muitos jornais.) Como você pode facilmente ver, quando olha a mensagem codificada, "s" quase certamente tem de ser "e", a letra mais comum da língua inglesa, e "p" deve ser o espaço. O restante da mensagem pode ser decodificado com mais algum tempo e esforço.

Quanto maior for uma mensagem codificada, tanto mais fácil será decifrá-la com uma tabela de freqüência. Para impedir o sucesso de um decifrador que está freqüentemente aplicando tabelas a uma mensagem codificada, você pode usar uma *cifra de substituição múltipla*. A mesma letra na mensagem original não corresponderá

necessariamente à mesma letra na forma codificada. Uma cifra de substituição múltipla é conseguida adicionando-se um segundo alfabeto embaralhado e trocando-se um alfabeto pelo outro cada vez que uma letra se repete. Para a segunda seqüência de caracteres, use

poi uytrewqasdfghjklmnbvcxz

O programa mostrado aqui trocará de alfabeto depois que uma letra for repetida duas vezes. Se você o usar para codificar a mensagem

meet me at sunset

essa forma codificada será:

tsslzsspplpumguuj

Para ver como isso funciona, coloque o alfabeto ordenado e os dois alfabetos aleatórios (chamados **sub** e **sub 2**) um sobre o outro, como mostrado aqui.

```
alfabeto:  abcdefghijklmnopqrstuvwxyz<espaço>
sub:       qazwsxedcrfvtgbyhnujm ikolp
sub 2:     poi uytrewqasdfghjklmnbvcxz
```

Quando o programa começar, o primeiro alfabeto aleatório será usado. A primeira letra na mensagem será "m", que corresponde a "t" em **sub** e a variável "m" na matriz **conta** será aumentada para 1. A próxima letra será "e", que se tornará "s"; e a variável "e" na matriz **conta** será aumentada para 1. Então o segundo "e" de "meet" será encontrado; desta vez **sub** ainda será usado para traduzi-lo, então ele também se tornará "s" e a variável será aumentada para 2. Isso fará com que o programa mude para o alfabeto **sub 2** e zere a variável "e" na matriz **conta**. Então o "t" de "meet" tornar-se-á "1" e o espaço se tornará "z". Então será encontrado o "m" de "me". Depois de ele ser traduzido para "s", voltará o alfabeto **sub**, pois foi encontrada uma letra repetida. Esse sistema de troca de alfabetos continuará até o fim da mensagem.

Aqui está um programa que criará uma cifra de substituição múltipla:

```
program subs3;  {cifrador de substituicao multipla}
type
   str80 = string[80];

var
   arq_ent,arq_sai: str80;
   alfabeto,alfa_sub,alfa_sub2: str80;
   inicio,t: integer;
   car: char;
   conta: array[1..27] of integer;

function Acha(alfabeto: str80; car: char): integer;
{esta funcao retorna um indice para o alfabeto}
var
   t: integer;

begin
   Acha:=-1;  {codigo de erro}
   for t:=1 to 27 do if car=alfabeto[t] then Acha:=t;
end;  {Acha}

function Letra(car: char): boolean;
{retorna TRUE se car for uma letra}
begin
   Letra:= (UpCase(car) >= 'A') and (UpCase(car) <= 'Z');
end;  {Letra}

function Indice(car:char): integer;
begin
   if Letra(car) then Indice:=Ord(UpCase(car))-Ord('A')+1
   else Indice:=27;  {entao e' o espaco do fim}
end;  {Indice}

procedure Cifra(arq_ent,arq_sai: str80);
var
   entrada,saida: file of char;
   car: char;
   trocou: boolean;

begin
   Assign(entrada,arq_ent);
   Reset(entrada);
   Assign(saida,arq_sai);
   Rewrite(saida);

   trocou:=TRUE;
   while not EOF(entrada) do
   begin
      read(entrada,car);
      car:=UpCase(car);
      if Letra(car) or (car=' ') then
      begin
         if trocou then
         begin
            car:=alfa_sub[Acha(alfabeto,car)];  {acha substituicao}
            conta[Indice(car)]:=conta[Indice(car)]+1;
         end
```

```
         else
         begin
            car:=alfa_sub2[Acha(alfabeto,car)];  {segunda substituicao}
            conta[Indice(car)]:=conta[Indice(car)]+1;
         end
      end;
      Write(saida,car);
      if conta[Indice(car)]=2 then
      begin
         trocou:=not trocou;
         for t:=1 to 27 do conta[t]:=0; {zera matriz de contagem}
      end;
   end;
   WriteLn('Arquivo cifrado.');
   Close(entrada); Close(saida);
end;  {Cifra}

procedure Decifra(arq_ent,arq_sai: str80);
var
   entrada,saida: file of char;
   car: char;
   trocou: boolean;
begin
   Assign(entrada,arq_ent);
   Reset(entrada);
   Assign(saida,arq_sai);
   ReWrite(saida);

   trocou:=TRUE;
   while not EOF(entrada) do
   begin
      read(entrada,car);
      car:=UpCase(car);
      if Letra(car) or (car=' ') then
      begin
         if trocou then
         begin
            car:=alfabeto[Acha(alfa_sub,car)];  {acha substituicao}
            conta[Indice(car)]:=conta[Indice(car)]+1;
         end
         else
         begin
            car:=alfabeto[Acha(alfa_sub2,car)];  {segunda substituicao}
            conta[Indice(car)]:=conta[Indice(car)]+1;
         end
      end;
      Write(saida,car);
      if conta[Indice(car)]=2 then
      begin
         trocou:=not trocou;
         for t:=1 to 27 do conta[t]:=0; {zera matriz de contagem}
      end;
   end;
   WriteLn('Arquivo decifrado.');
   Close(entrada); Close(saida);
end;  {Decifra}
```

```
begin
   alfabeto :='ABCDEFGHIJKLMNOPQRSTUVWXYZ ';
   alfa_sub :='QWERTYU IOPASDFGHJKLZXCVBNM'; {alfabeto de substituicao 1}
   alfa_sub2:=' MNBVCXZASDFGHJKLPOIUYTREWQ'; {alfabeto de substituicao 2}
   for t:=1 to 27 do conta[t]:=0; {inicializa matriz de contagem}
   Write('Digite o nome do arquivo de entrada: ');
   ReadLn(arq_ent);
   Write('Digite o nome do arquivo de saida: ');
   ReadLn(arq_sai);
   Write('Cifrar ou Decifrar (C ou D): ');
   ReadLn(car);
   if UpCase(car)='C' then Cifra(arq_ent,arq_sai)
   else if UpCase(car)='D' then Decifra(arq_ent,arq_sai);
end.
```

O uso de cifras de substituição múltipla torna a quebra de um código muito mais difícil pelo uso de tabelas de freqüência, pois em tempos diferentes letras diferentes correspondem à mesma coisa. Se você pensar nisso, será possível utilizar vários alfabetos aleatórios diferentes e uma rotina de alteração mais complexa para fazer com que todas as letras do texto codificado ocorressem com a mesma freqüência. Neste caso, uma tabela de freqüência seria inútil para quebrar o código.

CIFRAS DE TRANSPOSIÇÃO

Um dos mais antigos usos de códigos de transposição conhecidos foi elaborado pelos espartanos por volta de 475 a.C. Ele usava um dispositivo chamado *skytale*, que era basicamente uma faixa através da qual uma mensagem era escrita e era enrolada em um cilindro. Teoricamente, é impossível ler a faixa sem o cilindro, pois as letras estão fora de ordem. Na prática, entretanto, esse método deixa algo a desejar pois podem ser tentados cilindros de diferentes tamanhos até a mensagem começar a fazer sentido.

Você pode criar uma versão computadorizada de um *skytale*, colocando a mensagem original em uma matriz de um certo modo e escrevendo-a de outro modo. Para fazer isso, uma seqüência de caracteres unidimensional é usada para guardar a matriz a ser codificada, mas a mensagem está escrita no arquivo em disco como uma matriz bidimensional. Para esta versão, o original é uma matriz unidimensional de 100 bytes de comprimento, mas esta é escrita em disco como uma matriz bidimensional de 5 x 20. No entanto, você pode usar as dimensões que quiser. Devido ao fato de uma matriz de tamanho fixo armazenar a mensagem, é provável que nem todos os elementos da matriz sejam usados. Isto torna necessário inicializar a matriz antes de introduzir o original. Na

prática, é melhor inicializar a matriz utilizando caracteres aleatórios; entretanto, para simplificar, o símbolo # é usado (qualquer outro caractere serviria).

Se você colocasse a mensagem

meet me at sunset

na matriz **skytale** e a visse como uma matriz bidimensional, ela ficaria assim:

m	e	e	t	
m	e		a	t
	s	u	n	s
e	t	#	#	#
#	#	#	#	#

Daí, se você escrevesse a matriz por colunas, a mensagem ficaria assim:

mm e...eest...e u...tan... ts

onde os pontos indicam o número certo de sinais #. Para decodificar a mensagem, as colunas são introduzidas no *skytale*. Então, a matriz *skytale* pode ser exibida em ordem normal. O programa *skytale* utiliza esse método para codificar e decodificar mensagens.

```
program skytale;  {Cifrador Skytale}
{Codifica mensagens de ate 100 caracteres}
type
   str100 = string[100];
   str80  = string[80];

var
   arq_ent,arq_sai: str80;
   skytale: str100;
   t: integer;
   car: char;

function Letra(car: char): boolean;
{retorna TRUE se car for uma letra}
begin
   Letra:= (UpCase(car) >= 'A') and (UpCase(car) <= 'Z');
end;  {Letra}
```

```
procedure Cifra(arq_ent,arq_sai: str80);
{le um arquivo de texto e o transforma em uma matriz bidimensional}
var
   entrada,saida: file of char;
   car: char;
   t,t2: integer;

begin
   Assign(entrada,arq_ent);
   Reset(entrada);
   Assign(saida,arq_sai);
   Rewrite(saida);

   t:=1;
   while not EOF(entrada) do
   begin
      read(entrada,skytale[t]);
      t:=t+1;
   end;

   {cria a matriz bidimensional de 5 X 20}
   for t:=1 to 5 do
      for t2:=0 to 19 do
         Write(saida,skytale[t+(t2*5)]);
   WriteLn('Arquivo cifrado');
   Close(entrada); Close(saida);
end;  {Cifra}

procedure Decifra(arq_ent,arq_sai: str80);
var
   entrada,saida: file of char;
   car: char;
   t,t2:integer;

begin
   Assign(entrada,arq_ent);
   Reset(entrada);
   Assign(saida,arq_sai);
   ReWrite(saida);

   for t:=1 to 5 do
      for t2:=0 to 19 do
         Read(entrada,skytale[t+(t2*5)]);

   {escreve normalmente}
   for t:=1 to 100 do Write(saida,skytale[t]);

   WriteLn('Arquivo decifrado');
   Close(entrada); Close(saida);
end;  {Decifra}

begin
   for t:=1 to 100 do skytale[t]:=#;
   Write('Nome do arquivo de entrada: ');
   ReadLn(arq_ent);
   Write('Nome do arquivo de saida: ');
   ReadLn(arq_sai);
```

```
   Write('Cifra ou Decifra (C ou D): ');
   ReadLn(car);
   if UpCase(car)='C' then Cifra(arq_ent,arq_sai)
   else if UpCase(car)='D' then Decifra(arq_ent,arq_sai);
end.
```

Há outros métodos para se obter mensagens transpostas. Um método especialmente conveniente para computadores utiliza letras trocadas dentro da mensagem definida por algum algoritmo. Por exemplo, um programa que transpõe letras é transcrito a seguir.

```
program transpoe;  {Um cifrador de tranposicao}
{Codifica mensagens de ate 100 caracteres}
type
   str100 = string[100];
   str80  = string[80];

var
   arq_ent,arq_sai: str80;
   mensagem: str100;
   t: integer;
   car: char;

procedure Cifra(arq_ent,arq_sai: str80);
var
   entrada,saida: file of char;
   temp: char;
   t,t2: integer;

begin
   Assign(entrada,arq_ent);
   Reset(entrada);
   Assign(saida,arq_sai);
   Rewrite(saida);

   t:=1;
   while not EOF(entrada) and (t <= 100) do
   begin
      read(entrada,mensagem[t]);
      t:=t+1;
   end;
   mensagem[t-1]:='#';  {remove EOF}

   {transpoe os caracteres}
   for t2:=0 to 4 do
      for t:=1 to 10 do
      begin
         temp:=mensagem[t+t2*20];
         mensagem[t+t2*20]:=mensagem[t+10+t2*20];
         mensagem[t+10+t2*20]:=temp;
      end;

   {escreve}
   for t:=1 to 100 do Write(saida,mensagem[t]);
```

```
   WriteLn('Arquivo cifrado');
   Close(entrada); Close(saida);
end;  {Cifra}

procedure Decifra(arq_ent,arq_sai: str80);
var
   entrada,saida: file of char;
   temp: char;
   t,t2:integer;

begin
   Assign(entrada,arq_ent);
   Reset(entrada);
   Assign(saida,arq_sai);
   ReWrite(saida);

   t:=1;
   while (not EOF(entrada)) and (t <= 100) do
   begin
      read(entrada,mensagem[t]);
      t:=t+1;
   end;
   mensagem[t-1]:='#';  {remove EOF}

   {transpoe os caracteres}
   for t2:=0 to 4 do
      for t:=1 to 10 do
      begin
         temp:=mensagem[t+t2*20];
         mensagem[t+t2*20]:=mensagem[t+10+t2*20];
         mensagem[t+10+t2*20]:=temp;
      end;

   {escreve}
   for t:=1 to 100 do Write(saida,mensagem[t]);

   WriteLn('Arquivo decifrado');
   Close(entrada); Close(saida);
end;  {Decifra}

begin
   for t:=1 to 100 do mensagem[t]:=#;
   Write('Nome do arquivo de entrada: ');
   ReadLn(arq_ent);
   Write('Nome do arquivo de saida: ');
   ReadLn(arq_sai);
   Write('Cifra ou Decifra (C ou D): ');
   ReadLn(car);
   if UpCase(car)='C' then Cifra(arq_ent,arq_sai)
   else if UpCase(car)='D' then Decifra(arq_ent,arq_sai);
end.
```

Embora códigos de transposição possam ser eficientes, os algoritmos tornar-se-ão muito complexos se um alto grau de segurança for necessário.

CIFRAS MANIPULADORAS DE BIT

O computador digital deu origem a um novo método de codificação através da manipulação dos bits que compõem os caracteres reais do original. Embora um verdadeiro purista pudesse afirmar que a *manipulação de bit* (ou alteração, como às vezes é chamada) na verdade é apenas uma variação da cifra de substituição, os conceitos, métodos e opções diferem tão significativamente que ela tem de ser considerada um legítimo método de cifragem.

Cifras de manipulação de bit são bem convenientes para computadores pois eles empregam operações facilmente executáveis pelo sistema. Além disso, o texto cifrado tende a parecer completamente ininteligível, o que aumenta a segurança, fazendo o arquivo parecer inútil ou danificado, e confundindo qualquer um que tente ganhar acesso à mensagem.

Geralmente as cifras de manipulação de bits só são aplicáveis a arquivos baseados em computador e não podem ser usados para produzir mensagens impressas, pois a manipulação de bits tende a produzir caracteres não imprimíveis. Por essa razão, você deve pressupor que o arquivo deverá permanecer num arquivo de computador.

Cifras de manipulação de bit convertem o original em cifra, através da alteração do padrão real de bits de cada caractere, usando um ou mais dos seguintes operadores lógicos:

AND
OR
NOT
XOR

O Turbo Pascal é uma das melhores linguagens para criar cifras de manipulação de bit, pois admite esses operadores para uso com dados do tipo **byte**. Quando esses operadores são aplicados a variáveis **byte**, as operações ocorrem bit-a-bit, tornando mais fácil a alteração do estado dos bits dentro de um byte.

A cifra de manipulação mais simples e menos segura usa apenas o operador **NOT**, o operador do complemento. (Lembre-se de que o operador **NOT** provoca a inversão de cada bit dentro de um byte: um 1 torna-se 0 e 0 torna-se 1.) Portanto, um byte complementado duas vezes é igual ao original. O seguinte programa, chamado Complemento, codifica qualquer arquivo-texto pela inversão de bits dentro de cada

caractere. Devido à forte checagem de tipo do Turbo Pascal, o programa deve usar variáveis **byte** em vez de variáveis **char**, para que os operadores de manipulação de bit possam ser usados.

```
program complemento;  {Cifrador de complementacao}
type
   str80  = string[80];

var
   arq_ent,arq_sai: str80;
   t: integer;
   car: char;

procedure Cifra(arq_ent,arq_sai: str80);
var
   entrada,saida: file of byte;
   car:byte;

begin
   Assign(entrada,arq_ent);
   Reset(entrada);
   Assign(saida,arq_sai);
   Rewrite(saida);

   while not EOF(entrada) do
   begin
      read(entrada,car);
      car:=not car;
      Write(saida,car);
   end;

   WriteLn('Arquivo cifrado');
   Close(entrada); Close(saida);
end;  {Cifra}

procedure Decifra(arq_ent,arq_sai: str80);
var
   entrada,saida: file of byte;
   car: byte;

begin
   Assign(entrada,arq_ent);
   Reset(entrada);
   Assign(saida,arq_sai);
   ReWrite(saida);

   while not EOF(entrada) do
   begin
      read(entrada,car);
      car:=not car;
      Write(saida,car);
   end;

   WriteLn('Arquivo decifrado');
   Close(entrada); Close(saida);
end;  {Decifra}
```

```
begin
   Write('Nome do arquivo de entrada: ');
   ReadLn(arq_ent);
   Write('Nome do arquivo de saida: ');
   ReadLn(arq_sai);
   Write('Cifra ou Decifra (C ou D): ');
   ReadLn(car);
   if UpCase(car)='C' then Cifra(arq_ent,arq_sai)
   else if UpCase(car)='D' then Decifra(arq_ent,arq_sai);
end.
```

É difícil mostrar qual seria a aparência do texto cifrado, pois a manipulação de tipo usada aqui geralmente cria caracteres não imprimíveis. Tente-o no seu computador e examine o arquivo – ele parecerá bastante críptico.

Há dois problemas com esse esquema de codificação simples. Primeiro, o programa de criptização não utiliza uma chave para codificar, então, qualquer um com acesso ao programa pode decodificar e codificar arquivos. Segundo, e talvez mais importante, esse método seria facilmente percebido por um experiente programador de computadores.

Um método melhorado de manipulação de bit utiliza o operador **XOR.**

O operador **XOR** tem a seguinte tabela-verdade:

XOR	0	1
0	0	1
1	0	0

O produto da operação **XOR** é **TRUE** se e apenas se um operando for **TRUE** e o outro **FALSE.** Isso dá ao **XOR** uma propriedade única: se você usar **XOR** em um byte com um outro byte chamado chave, e então tomar o resultado da operação e usar **XOR** novamente com a chave, o resultado será o byte original, como está mostrado aqui:

```
        1101 1001
XOR     0101 0011   (chave)
        1000 1010

        1000 1010
XOR     0101 0011   (chave)
        1101 1001
```

Quando usado para a codificação de um arquivo, esse processo resolve os dois problemas inerentes ao método do complemento. Antes de mais nada, devido ao fato de ele utilizar uma chave, o programa de criptização, sozinho, não pode decodificar um arquivo; segundo, como o uso da chave faz cada arquivo único, o que foi feito com o arquivo não é óbvio para alguém com conhecimentos de ciência da computação.

A chave não precisa ter apenas um byte de comprimento. Por exemplo, você poderia usar uma chave de diversos caracteres e alternar os caracteres através do arquivo. Entretanto, aqui é usada uma chave de apenas um caractere, para manter o programa simples:

```
program Xor_Com_Chave;  {xor com uma chave, para aumentar a seguranca}
type
   str80  = string[80];

var
   arq_ent,arq_sai: str80;
   t: integer;
   car: char;
   chave: byte;

procedure Cifra(arq_ent,arq_sai: str80; chave:byte);
var
   entrada,saida: file of byte;
   car:byte;

begin
   Assign(entrada,arq_ent);
   Reset(entrada);
   Assign(saida,arq_sai);
   Rewrite(saida);

   while not EOF(entrada) do
   begin
      read(entrada,car);
      car:=chave xor car;
      Write(saida,car);
   end;

   WriteLn('Arquivo cifrado');
   Close(entrada); Close(saida);
end;  {Cifra}

procedure Decifra(arq_ent,arq_sai: str80; chave: byte);
var
   entrada,saida: file of byte;
   car: byte;

begin
   Assign(entrada,arq_ent);
   Reset(entrada);
   Assign(saida,arq_sai);
   ReWrite(saida);
```

```
    while not EOF(entrada) do
    begin
       read(entrada,car);
       car:=chave xor car;
       Write(saida,car);
    end;

    WriteLn('Arquivo decifrado');
    Close(entrada); Close(saida);
end;  {Decifra}

begin
    Write('Nome do arquivo de entrada: ');
    ReadLn(arq_ent);
    Write('Nome do arquivo de saida: ');
    ReadLn(arq_sai);
    Write('Chave(um caractere): ');
    ReadLn(car);
    chave:=Ord(car);
    Write('Cifra ou Decifra (C ou D): ');
    ReadLn(car);
    if UpCase(car)='C' then Cifra(arq_ent,arq_sai,chave)
    else if UpCase(car)='D' then Decifra(arq_ent,arq_sai,chave);
end.
```

COMPRESSÃO DE DADOS

Técnicas de compressão de dados, essencialmente, espremem uma determinada quantidade de informação em uma área menor. Elas são usadas em sistemas computacionais para aumentar a capacidade de armazenamento do sistema (por redução das necessidades de armazenamento do usuário do computador), para reduzir o tempo de transferência (especialmente através de linhas telefônicas), e estabelecer um nível de segurança. Embora haja muitos esquemas de compressão de dados disponíveis, nós examinaremos apenas dois deles. O primeiro é por *compressão de bits*, onde mais de um caractere é armazenado em um mesmo byte, e o segundo é por *deleção de caracteres*, no qual caracteres de um arquivo são apagados.

DE OITO PARA SETE

A maioria dos computadores utiliza tamanhos de bytes que são potências de 2 por causa da representação binária de dados na máquina. As letras maiúsculas e minúsculas e a pontuação requerem apenas cerca de 63 códigos diferentes, precisando de apenas 6 bits

para representar um byte. (Um byte de 6 bits poderia ter valores de 0 a 63.) Entretanto, a maioria dos computadores utiliza um byte de oito bits; portanto, em qualquer arquivo de texto, 25% do armazenamento do byte é desperdiçado. Você poderia, portanto, compactar 4 caracteres em três bytes se pudesse usar os dois últimos bits de cada byte. O único problema é que existem mais de 63 códigos ASCII, organizados de modo que as letras maiúsculas e minúsculas caem mais ou menos no meio da escala. Isso quer dizer que alguns caracteres necessários requerem pelo menos 7 bits. É possível usar uma representação não-ASCII (o que é feito em raras ocasiões), mas não é geralmente recomendável. Uma opção mais fácil é compactar 8 caracteres em 7 bytes, explorando o fato de que nenhuma letra ou marca de pontuação normal utiliza o oitavo bit de um byte. Portanto, você pode usar o oitavo bit de cada um dos sete bytes para armazenar o oitavo caractere. Esse método economiza 12,5%.

Entretanto, muitos computadores, incluindo o IBM PC, de fato utilizam caracteres de oito bits para representar alguns caracteres especiais ou gráficos. Além disso, alguns processadores de texto também usam o oitavo bit para indicar instruções de processamento de texto. Portanto, usar esse tipo de compactação de dados só funciona em arquivos ASCII "estritos", que não usam o oitavo bit.

Para visualizar como isso funciona, considere os 8 caracteres seguintes representados como bytes de oito bits:

```
byte 1  0 1 1 1  0 1 0 1
byte 2  0 1 1 1  1 1 0 1
byte 3  0 0 1 0  0 0 1 1
byte 4  0 1 0 1  0 1 1 0
byte 5  0 0 0 1  0 0 0 0
byte 6  0 1 1 0  1 1 0 1
byte 7  0 0 1 0  1 0 1 0
byte 8  0 1 1 1  1 0 0 1
```

Como você pode ver, o oitavo bit é sempre 0. Esse é sempre o caso, a menos que o oitavo bit seja usado para checagem de paridade. A maneira mais fácil de comprimir 8 caracteres em 7 é distribuir os 7 bits significativos do byte 1 nas 7 posições não usadas de oitavo bit das posições de 2 a 8. Os 7 bytes remanescentes então aparecem assim:

byte 1 – na vertical

byte 2 1 1 1 1 1 1 0 1
byte 3 1 0 1 0 0 0 1 1
byte 4 1 1 0 1 0 0 1 1
byte 5 0 0 0 1 0 0 0 0
byte 6 1 1 1 0 1 1 0 1
byte 7 0 0 1 0 1 0 1 0
byte 8 1 1 1 1 1 0 0 1

Para reconstruir o byte 1, você precisa apenas juntá-lo de novo, tomando o oitavo bit de cada um dos outros sete bits.

Esta técnica de compressão comprime qualquer texto de 1/8, ou 12,5%. Essa economia é bastante substancial. Por exemplo, se você estivesse transmitindo o código-fonte de seu programa favorito para um amigo, por uma linha telefônica de longa distância, estaria economizando 12,5% das despesas de transmissão. (Lembre-se: o código-objeto, ou função executável do programa, precisa dos 8 bits completos.) O seguinte programa comprime um arquivo-texto, usando o método descrito:

```
program compressor;  {coloca caracteres em palavras de 7 bits}
type
   str80  = string[80];

var
   arq_ent,arq_sai: str80;
   t: integer;
   car: char;

procedure Comprime(arq_ent,arq_sai:str80);
var
   entrada,saida: file of byte;
   car,car2:byte;
   feito:boolean;

begin
   Assign(entrada,arq_ent);
   Reset(entrada);
   Assign(saida,arq_sai);
   Rewrite(saida);

   feito:=FALSE;
   repeat
      read(entrada,car);
      if EOF(entrada) then
         feito:=TRUE
      else
      begin
         car:=car shl 1;  {retira o bit nao usado}
```

```
         for t:=0 to 6 do
         begin
            if EOF(entrada) then
            begin
               car2:=0;
               feito:=TRUE;
            end else read(entrada,car2);
            car2:=car2 and 127;  {vira o bit superior}
            car2:=car2 or ((car shl t) and 128);  {empacota os bits}
            Write(saida,car2);
         end;
      end;  {else}
   until feito;

   WriteLn('Arquivo comprimido');
   Close(entrada); Close(saida);
end;  {Comprime}


procedure Descomprime(arq_ent,arq_sai:str80);
var
   entrada,saida: file of byte;
   car,car2:byte;
   s: array[1..7] of byte;
   feito:boolean;

begin
   Assign(entrada,arq_ent);
   Reset(entrada);
   Assign(saida,arq_sai);
   Rewrite(saida);

   feito:=FALSE;
   repeat
      car:=0;
      for t:=1 to 7 do
      begin
         if EOF(entrada) then
            feito:=TRUE
         else
         begin
            read(entrada,car2);
            s[t]:=car2 and 127;  {vira o bit superior}
            car2:=car2 and 128;  {apaga os bits inferiores}
            car2:=car2 shr t;  {desempacota}
            car:=car or car2;  {reconstroi o oitavo byte}
         end;
      end;
      Write(saida,car);
      for t:=1 to 7 do Write(saida,s[t]);
   until feito;

   WriteLn('Arquivo descomprimido');
   Close(entrada); Close(saida);
end;  {Descomprime}
```

```
begin
   Write('Nome do arquivo de entrada: ');
   ReadLn(arq_ent);
   Write('Nome do arquivo de saida: ');
   ReadLn(arq_sai);
   Write('Comprime ou Descomprime (C ou D): ');
   ReadLn(car);
   if UpCase(car)='C' then Comprime(arq_ent,arq_sai)
   else if UpCase(car)='D' then Descomprime(arq_ent,arq_sai);
end.
```

O código deste programa é bastante complexo porque vários bits devem ser mudados. Se você se lembrar do que foi feito do primeiro byte de cada oito, o código ficará mais fácil de ser seguido.

A LINGUAGEM DE 16 CARACTERES

Embora inadequado para a maioria das situações, um método interessante de compressão de dados deleta letras desnecessárias de palavras – em essência mudando muitas palavras por abreviações. A compressão de dados é alcançada porque os caracteres não usados não são armazenados. Economizar espaço usando abreviações é muito comum – é por isso que "Sr." é usado em vez de "Senhor". Em vez de usar realmente abreviações, o método apresentado nesta seção remove automaticamente certas letras de uma mensagem. Para fazer isso é necessário um *alfabeto mínimo*. Um alfabeto mínimo é aquele no qual diversas letras raramente usadas são removidas, deixando apenas aquelas necessárias para formar a maioria das palavras ou para evitar ambigüidade. Portanto, qualquer letra que não conste do alfabeto mínimo será extraída de qualquer palavra em que ela apareça. Exatamente quantos caracteres existem em um alfabeto mínimo é uma questão de escolha. Entretanto, neste capítulo utilizamos 14 letras mais comuns, mais espaço e caractere *return*.

Automatizar o processo de abreviação requer que você saiba quais letras do alfabeto são usadas mais freqüentemente, de modo que você possa criar um alfabeto mínimo. Em teoria, você poderia contar as letras de cada palavra em um dicionário. Entretanto, diferentes autores utilizam freqüências de misturas diferentes, portanto uma tabela de freqüência, baseada apenas nas palavras disponíveis em inglês, pode não refletir a verdadeira freqüência de uso das letras. (Levaria muito tempo para contar as letras!) Como alternativa, você poderia contar a freqüência de letras neste capítulo e usá-la como base de seu alfabeto mínimo. Para fazer isso você poderia usar o seguinte programa simples. Esse programa salta toda pontuação, exceto pontos, vírgulas e espaços.

```
program Conta;
{conta o numero de ocorrencias de cada tipo
de caractere num arquivo}

type
   str80 = string[80];

var
  arq_ent: str80;
  t: integer;
  letras: array[0..25] of integer;
  espaco,ponto,virgula: integer;

function Letra(car:char): boolean;
{TRUE se car for uma letra do alfabeto}
begin
   Letra:=(UpCase(car) >= 'A') and (UpCase(car) <= 'Z');
end;  {Letra}

procedure Conta(arq_ent: str80);
var
   entrada: file of char;
   car: char;

begin
   Assign(entrada,arq_ent);
   Reset(entrada);

   while not EOF(entrada) do
   begin
      Read(entrada,car);
      car:=UpCase(car);
      if Letra(car) then
         letras[Ord(car)-Ord('A')]:=letras[Ord(car)-Ord('A')]+1
      else case car of
         ' ': espaco:=espaco+1;
         '.': ponto:=ponto+1;
         ',': virgula:=virgula+1;
      end;
   end;
   Close(entrada);
end;  {Conta}

begin
   Write('Nome do arquivo de entrada: ');
   ReadLn(arq_ent);
   for t:=0 to 25 do letras[t]:=0;
   espaco:=0; virgula:=0; ponto:=0;
   Conta(arq_ent);
   for t:=0 to 25 do
      WriteLn(Chr(t+Ord('A')),':',letras[t]);
   WriteLn('espaco: ',espaco);
   WriteLn('ponto: ',ponto);
   WriteLn('virgula: ',virgula);
end.
```

Para conseguir compressão de dados significativa, você deve cortar o alfabeto substancialmente, removendo as letras usadas menos freqüentemente. Embora haja muitas opiniões sobre o que seja exatamente um alfabeto mínimo, as 14 letras principais e o espaço somam em torno de 85% de todos os caracteres neste capítulo. Devido ao caractere *return* também ser necessário para evitar a quebra de palavras, ele também será incluído. Portanto, este capítulo usa um alfabeto mínimo, consistindo em 14 caracteres, no espaço e no retorno de carro:

A B D E H I L M N O R S T U <espaço> <RC>

Aqui está um programa que remove todos os caracteres, exceto os 16 selecionados. O programa realmente começa uma nova linha, se um caractere *return* estiver presente. Isso torna a saída legível – não é necessário armazenar o caractere *linefeed*, pois ele pode ser reconstruído mais tarde.

```
program Compressor2;  {linguagem de 16 caracteres}
type
   str80  = string[80];

var
   arq_ent,arq_sai: str80;
   t: integer;
   car: char;

procedure Comp2(arq_ent,arq_sai:str80);
var
   entrada,saida: file of char;
   car: char;
   feito:boolean;

begin
   Assign(entrada,arq_ent);
   Reset(entrada);
   Assign(saida,arq_sai);
   Rewrite(saida);

   feito:=FALSE;
   repeat
      if not EOF(entrada) then
      begin
         Read(entrada,car);
         car:=UpCase(car);
         if Pos(car,'ABCDEJILMNORSTU ') <> 0 then Write(saida,car);
         if Ord(car)=13 then Write(saida,car);  {cr}
         if Ord(car)=10 then Write(saida,car);  {lf}
      end
      else feito:=TRUE;
   until feito;
```

```
    WriteLn('Arquivo comprimido');
    Close(entrada); Close(saida);
end;  {Comp2}

begin
    Write('Nome do arquivo de entrada: ');
    ReadLn(arq_ent);
    Write('Nome do arquivo de saida: ');
    ReadLn(arq_sai);
    Comp2(arq_ent,arq_sai);
end.
```

O programa utiliza a função **Pos**, que verifica se cada caractere lido está no alfabeto mínimo. **Pos** retorna 0 se não houver correspondência, ou a posição da primeira correspondência encontrada.

Se você usar esse programa na mensagem.

Atenção alto comando:

> Ataque bem-sucedido. Por favor mandem suprimentos adicionais e novos soldados. Isso é essencial para manter nossa cabeça-de-ponte.
> General Frashier

a mensagem comprimida ficaria assim:

Atenao alto omando

> ataue bem suedido. Or aor mandem surimentos adiionais e noos soldados.
> Isso é essenial ara manter nossa abea de onte.
> ... eneral rashier

DECIFRAÇÃO DE CÓDIGOS

Nenhum capítulo de cifragem de dados é completo sem uma olhada rápida em decifração de códigos. A arte de decifração de códigos é essencialmente de tentativa e erro. Com o uso de computadores digitais, cifras relativamente simples podem ser facilmente quebradas através de teste exaustivo. Entretanto, os códigos mais complexos ou não podem ser quebrados ou requerem técnicas e recursos não comumente disponíveis. Por simplicidade, esta seção dedica-se a decifrar os códigos mais simples.

Se você quiser decifrar uma mensagem que foi cifrada, usando um método de substituição simples apenas com o alfabeto trocado, então tudo que tem a fazer é tentar todas as 26 substituições possíveis para ver qual serve. Um programa para fazer isso é mostrado aqui:

```
program Decifrador;  {Programa decodificador de cifradores
                      de substituicao simples. Menssagens podem
                      ter ate' 1000 caracteres.}

type
   str80 = string[80];

var
   arq_ent: str80;
   mensagem: array[1..1000] of char;  {contem a mensagem}
   car: char;

function Letra(car: char): boolean;
{returns TRUE if car is a letter of the alphabet}
begin
   Letra:= (UpCase(car) >= 'A') and (UpCase(car) <= 'Z');
end;  {Letra}

procedure Decifra(arq_ent: str80);
var
   entrada: file of char;
   car: char;
   feito: boolean;
   sub,t,t2,l: integer;


begin
   Assign(entrada,arq_ent);
   Reset(entrada);

   feito:=FALSE;
   l:=1;
   repeat
      Read(entrada,mensagem[l]);
      mensagem[l]:=UpCase(mensagem[l]);
      l:=l+1;
   until EOF(entrada);
   l:=l-1;  {apaga o caractere de EOF}

   t:=0; sub:=-1;   {nao decifrada}
   repeat
      for t2:=1 to l do
      begin
         car:=mensagem[t2];
         if Letra(car) then
         begin
            car:=Chr(Ord(car)+t);
            if car > 'Z' then car:=Chr(Ord(car)-26);
         end;
         Write(car);
```

```
      end;
      WriteLn;
      WriteLn('Decifrada? (S/N): ');
      Read(car);
      WriteLn;
      if UpCase(car)='S' then sub:=t;
      t:=t+1;
   until (t=26) or (UpCase(car)='S');
   if sub <> -1 then Write('O resultado e'' ',sub,'.');
   Close(entrada);
end;  {Decifra}

begin
   Write('Nome do arquivo de entrada: ');
   ReadLn(arq_ent);
   Decifra(arq_ent);
end.
```

Com apenas uma ligeira variação, você poderia usar o mesmo programa para quebrar cifras que utilizam um alfabeto aleatório. Nesse caso, substitua manualmente alfabetos introduzidos como mostrado no programa:

```
program Decifrador2;  {Decodificador de cifradores de substituicao
                       aleatoria. Mensagens podem ter ate' 1000
                       caracteres.}
type
   str80 = string[80];

var
   arq_ent: str80;
   sub: array[0..25] of char;
   mensagem: array[1..1000] of char;  {holds input mensagem}
   car: char;

function Letra(car: char): boolean;
{TRUE se car uma letra do alfabeto}
begin
   Letra:= (UpCase(car) >= 'A') and (UpCase(car) <= 'Z');
end;  {Letra}

procedure Decifra2(arq_ent: str80);
var
   entrada: file of char;
   car: char;
   feito: boolean;
   t,l: integer;


begin
   Assign(entrada,arq_ent);
   Reset(entrada);

   feito:=FALSE;
   l:=1;
   repeat
```

```
      Read(entrada,mensagem[l]);
      mensagem[l]:=UpCase(mensagem[l]);
      l:=l+1;
   until EOF(entrada);
   l:=l-1;  {clear EOF char}

   repeat
      Write('Digite alfabeto de substituicao: ');
      ReadLn(sub);
      for t:=1 to l do
      begin
         car:=mensagem[t];
         if Letra(car) then begin
            car:=sub[Ord(car)-Ord('A')];
         end;
         Write(car);
      end;
      WriteLn;
      WriteLn('Decifrada? (S/N): ');
      ReadLn(car);
      if UpCase(car)='S' then feito:=TRUE;
   until feito;
   WriteLn('O alfabeto de substituicao e'': ',sub);
   Close(entrada);
end;  {Decifra2}

begin
   Write('Nome do arquivo de entrada: ');
   ReadLn(arq_ent);
   Decifra2(arq_ent);
end.
```

Ao contrário das cifras de substituição, as cifras de transposição e cifras de manipulação de bit são mais difíceis de quebrar pelos métodos de tentativa e erro mostrados aqui. Se você tiver de decifrar códigos tão complexos, boa sorte!

CAPÍTULO 8

GERADORES DE NÚMEROS ALEATÓRIOS E SIMULAÇÕES

Seqüências de números aleatórios (ou randômicos) são usadas em uma série de situações de programação, variando de simulações (que são as mais comuns) a jogos e programas educacionais. O Turbo Pascal contém uma função interna chamada **Random**, que gera números aleatórios. Como você verá neste capítulo, **Random** é um excelente gerador de números aleatórios, mas talvez em algumas aplicações você precise de dois ou mais geradores diferentes que forneçam séries diferentes de números aleatórios para tarefas diferentes. Além disso, algumas simulações requerem um gerador de números aleatórios *assimétrico* ou desequilibrado, o qual produza uma seqüência que sofra uma inclinação maior para um lado ou para outro. A primeira parte deste capítulo é devotada à construção de geradores de números aleatórios e ao teste de sua qualidade.

A segunda parte desse capítulo mostra como você pode utilizar números aleatórios em simulações do mundo real. A primeira é a simulação de uma fila de caixas de um supermercado, e a segunda é o gerenciamento de carteiras de ações pelo método de "random walk". Ambos ilustram os fundamentos dos programas de simulação.

GERADORES DE NÚMEROS ALEATÓRIOS

Tecnicamente, o termo *gerador de números aleatórios* é absurdo; números, por si próprios, não são aleatórios. Por exemplo, 100 é um número aleatório? E 25? O que realmente se entende por *gerador de números aleatórios* é alguma coisa que cria uma

seqüência de números a qual parece estar em ordem aleatória. Isso levanta uma questão mais complexa: o que é uma seqüência aleatória de números? A única resposta correta é que uma amostra aleatória de números é uma seqüência na qual não há qualquer relação entre os elementos. Essa definição leva ao paradoxo de que qualquer seqüência pode ser tanto aleatória quanto não-aleatória, dependendo do modo como a seqüência foi obtida. Por exemplo, essa lista de números

```
1 2 3 4 5 6 7 8 9
```

foi obtida batendo-se as teclas da fileira de cima, em ordem, no teclado, de modo que a seqüência não pode ser interpretada como de geração aleatória. Mas e se acontecesse de você tirar a mesma seqüência de um barril de bolas de tênis numeradas? Então essa seria uma seqüência gerada aleatoriamente. Esta discussão mostra que a aleatoriedade de uma seqüência depende de como ela foi gerada e não de qual seja ela.

Tenha em mente que seqüências de números gerados por um computador são *determinísticas*. Cada número, exceto o primeiro, depende do número que o precede. Tecnicamente, isso significa que um computador pode criar apenas seqüências pseudo-aleatórias. Entretanto, isso é suficiente para a maioria dos problemas e, para os propósitos deste livro, as seqüências serão chamadas simplesmente de aleatórias.

Geralmente, é melhor que os números de uma seqüência aleatória estejam distribuídos uniformemente (não confunda isso com distribuição normal ou a curva senoidal.) Em uma distribuição uniforme, todos os eventos são igualmente prováveis, de modo que o gráfico de uma distribuição uniforme tende a ser uma linha reta em vez de uma curva.

Antes da difusão do uso dos computadores, sempre que se precisava de números aleatórios eles eram produzidos atirando-se dados ou tirando-se bolas com números de uma jarra. Em 1955, a RAND Corporation publicou uma tabela com 1 milhão de dígitos aleatórios obtidos com a ajuda de uma máquina similar a um computador. Nos primeiros dias da ciência da computação, embora tenham sido inventados muitos métodos para gerar números aleatórios, a maioria foi descartada.

Um método particularmente interessante que quase funcionou foi o desenvolvido por John von Neumann, o pai do computador moderno. Freqüentemente citado como *método do meio do quadrado (middle-square method)*, ele eleva o número aleatório anterior ao quadrado e extrai os dígitos do meio. Por exemplo, se você estivesse criando números de três dígitos e o valor anterior fosse 121, você o elevaria ao quadrado obtendo 14.641. A extração dos dígitos do meio produziria 464 como número seguinte. O problema

com esse método é que ele tende a dar um padrão de repetição muito curto chamado de ciclo, principalmente depois que um zero entrou no padrão. Por essa razão o método não é mais usado.

Atualmente, o modo mais comum de se gerar números aleatórios é utilizando a equação

$$R_{n+1} = (aR_n + c) \bmod m$$

onde as seguintes condições devem ser observadas:

$R >= 0$
$a >= 0$
$c >= 0$
$m > R_0$, a e c

Note que R_n é o número anterior, e R_{n+1} é o número seguinte. Esse método é citado algumas vezes como *método linear congruencial*. A fórmula é tão simples que você poderia pensar que a geração de números aleatórios é fácil. Entretanto há uma armadilha: a eficiência dessa equação depende muito dos valores de a, c e m. A escolha desses valores às vezes tem mais de arte do que de ciência. Há regras complexas que podem auxiliá-lo a escolher os valores; entretanto, essa discussão cobrirá apenas algumas regras e experimentos simples.

O módulo (m) deve ser bastante grande, pois ele determina o alcance dos números aleatórios. A operação do módulo produz o resto de uma divisão que utiliza os mesmos operadores. Daí, 10 **mod** 3 é 1, porque 3 cabe em 10 três vezes com resto 1. Portanto, se o módulo for 12, a equação pode produzir números de 0 à 11, ao passo que se o módulo for 21.425, os números produzidos podem variar de 0 a 21.424. Lembre-se de que um módulo pequeno não afeta efetivamente a aleatoriedade – afeta apenas a extensão do intervalo. A escolha do multiplicador a e do incremento c é muito mais difícil. Normalmente, o multiplicador pode ser bastante grande e o incremento bastante pequeno. São necessários muitos testes para se confirmar que foi criado um bom gerador.

Como um primeiro exemplo, aqui está um dos geradores de números aleatórios mais comuns. A equação mostrada em **Ran1** foi usada como base para o gerador de números aleatórios em várias linguagens populares de programação.

```
var
   a1: integer;  {deve ter valor 1 antes da primeira chamada}

function Ran1: real;
var
   t: real;
begin
   t:=(a1*32749+3) mod 32749;
   a1:=Trunc(t);
   Ran1:=Abs(t/32749);
end;  {Ran1}
```

Essa função possui três características importantes. Primeiro, o número aleatório é, na verdade, um número inteiro, embora a função retorne um real. Os inteiros são necessários para o método congruencial linear, mas espera-se que geradores de números aleatórios, por convenção, retornem um número entre 0 e 1, isto é, um valor real.

Segundo, a *semente*, ou valor inicial, é fixada através da variável global **a1**. Antes da primeira chamada de **Ran 1** a variável **a1** deve ser inicializada em 1.

Terceiro, em **Ran 1**, o número aleatório é dividido pelo módulo antes que a função retorne para gerar um número entre 0 e 1. Se você estudar isso, verá que o valor de **a1**, antes da linha de retorno, tem de estar entre 0 e 32.478. Portanto, quando **a1** for dividido por 32.749, será obtido um número igual ou maior que 0 mas menor que 1.

Muitos geradores de números aleatórios não são úteis porque produzem distribuições não uniformes ou porque produzem seqüências curtas e repetitivas. Mesmo que esses problemas pareçam bastante insignificantes, se o gerador for usado constantemente ele pode provocar resultados tendenciosos. A solução é criar vários geradores diferentes e usá-los individual ou conjuntamente para obter números mais aleatórios. O uso de vários geradores ajuda a suavizar a distribuição das seqüências, reduzindo as pequenas inclinações de cada gerador. Portanto, aqui está uma outra função geradora de números aleatórios, chamada **Ran2**, que produz uma boa distribuição.

```
var
   a2: integer;  {deve ter valor 203 antes da primeira chamada}

function Ran2: real;
var
   t: real;
begin
   t:=(a2*10001+3) mod 17417;
   a2:=Trunc(t);
   Ran2:=Abs(t/17417);
end;  {Ran2}
```

A variável global **a2** deve ser inicializada em 203 antes da primeira chamada de **Ran 2.**

Cada um desses geradores de números aleatórios produz uma boa seqüência de números aleatórios. Mesmo assim, a questão permanece: qual a aleatoriedade das seqüências? Qual a qualidade desses geradores?

DETERMINANDO A QUALIDADE DE UM GERADOR

Você pode utilizar diversos testes para determinar a aleatoriedade de uma seqüência de números. Nenhum desses testes dirá se a seqüência é aleatória; pelo contrário, eles dirão se ela não é. Os testes podem identificar uma seqüência não-aleatória, mas, mesmo que um teste específico não encontre um problema, isso não significa que dada seqüência seja realmente aleatória. Entretanto, o teste de fato aumenta nossa confiança no gerador de números aleatórios que produziu a seqüência. Para o propósito deste livro, a maioria dos testes é complicada ou demorada demais na sua forma mais rigorosa. Portanto, você verá agora, brevemente, alguns modos simples com os quais pode-se testar uma seqüência.

Para começar, aqui está um modo de descobrir quanto os números de uma seqüência ajustam-se ao que se poderia esperar que fosse uma distribuição aleatória. Por exemplo, digamos que você estivesse tentando gerar seqüências aleatórias com os dígitos de 0 a 9. A probabilidade de ocorrência de cada dígito é de 1/10, pois existem 10 possibilidades para cada número na seqüência, todas igualmente possíveis. Suponha que a seqüência

9 1 8 2 4 6 3 7 5 8 2 9 0 4 2 4 7 8 6 2

tenha sido gerada. Se você contar o número de vezes que cada dígito ocorre, o resultado é

Dígito	**Ocorrências**
0	1
1	1
2	4
3	1
4	3
5	1
6	2
7	2
8	3
9	2

Você deveria se perguntar em seguida se essa distribuição é suficientemente similar à esperada.

Lembre-se: se um gerador de números aleatórios for bom, ele gerará seqüências aleatoriamente. Em um estado realmente aleatório, todas as seqüências são possíveis. Isso parece implicar que qualquer seqüência gerada deva ser qualificada como uma seqüência aleatória válida. Então, como você poderá dizer se a seqüência dada agora é aleatória? De fato, como qualquer seqüência de dez dígitos pode ser não-aleatória, se qualquer seqüência é possível? A resposta é que algumas seqüências têm maior probabilidade de serem aleatórias que outras. Você pode determinar a *probabilidade* de uma dada seqüência ser aleatória, usando o *teste de qui-quadrado*.

Basicamente, o teste de qui-quadrado subtrai o número esperado do número observado de ocorrências para todos os números gerados. Esse resultado é chamado de V. Você pode usar V para encontrar uma percentagem em uma tabela de valores de qui-quadrado. Essa percentagem representa a probabilidade de que tenha sido produzida uma seqüência aleatória. Uma pequena tabela de qui-quadrado é dada na Figura 8-1; você pode encontrar tabelas completas na maior parte dos livros de estatística.

	$p=99\%$	$p=95\%$	$p=75\%$	$p=50\%$	$p=25\%$	$p=5\%$
$n=5$	0,5543	1,1455	2,675	4,351	6,626	11,07
$n=10$	2,558	3,940	6,737	9,342	12,55	18,31
$n=15$	5,229	7,261	11,04	14,34	18,25	25,00
$n=20$	8,260	10,85	15,45	19,34	23,83	31,41
$n=30$	14,95	18,49	24,48	29,34	34,80	43,77

Figura 8-1 Valores escolhidos de qui-quadrado.

A fórmula para obter V é

$$V=\sum_{i=1}^{N}\frac{(O_i-E_i)^2}{E_i}$$

onde O_i é o número de ocorrências observadas, E_i é o número de ocorrências esperadas, e N é o número de elementos discretos. O valor de E_i é obtido multiplicando-se a probabilidade da ocorrência de cada elemento pelo número de observações. Nesse caso, por esperarmos que cada dígito ocorra um décimo das vezes e que sejam tomadas 20

amostras, o valor para E é 2 para todos os dígitos. N é 10 pois existem 10 elementos possíveis – os dígitos de 0 a 9. Portanto,

$$V = \frac{(1-2)^2}{2} + \frac{(1-2)^2}{2} + \frac{(4-2)^2}{2} + \frac{(1-2)^2}{2} + \frac{(3-2)}{2} +$$

$$\frac{(1-2)^2}{2} + \frac{(2-2)^2}{2} + \frac{(2-2)^2}{2} + \frac{(3-2)^2}{2} + \frac{(2-2)^2}{2}$$

$$= \frac{1}{2} + \frac{1}{2} + \frac{4}{2} + \frac{1}{2} + \frac{1}{2} + \frac{1}{2} + 0 + 0 + \frac{1}{2} + 0$$

$$= 5$$

Para determinar a probabilidade de que uma seqüência não seja aleatória, ache a linha na tabela mostrada na Figura 8-1 que iguale o número de observações; neste caso é 20. Então, siga até encontrar um número que seja maior que V. Neste caso, é a coluna 1. Isso quer dizer que existe uma probabilidade de 99% de que uma amostra de 20 elementos tenha um V maior que 8,260. Por outro lado, isso significa que existe apenas 1% de probabilidade de que a seqüência testada tenha sido gerada aleatoriamente. Para "passar" no teste do qui-quadrado, a probabilidade para V deve cair entre 25% e 75%. (Essa faixa é obtida através do uso de matemática que vai além do propósito deste livro.)

Entretanto, você pode confrontar essa conclusão com a seguinte questão: desde que todas as seqüências são possíveis, como pode essa seqüência ter apenas 1% de chance de ser legítima? A resposta é que trata-se apenas de uma probabilidade – o teste de qui-quadrado, na verdade, não é nenhum teste, é apenas medida de segurança adicional. De fato, para evitar a rejeição de um bom gerador de números aleatórios, se você utilizar o teste de qui-quadrado, deverá obter várias seqüências diferentes e tomar a média dos resultados. Qualquer seqüência simples poderia ser rejeitada, mas várias seqüências em média conjunta devem fornecer um bom teste.

Por outro lado, uma seqüência pode passar no teste de qui-quadrado e mesmo assim não ser aleatória. Por exemplo,

1 3 5 7 9 1 3 5 7 9

passa no teste de qui-quadrado mas não parece ser muito aleatória. Nesse caso, foi gerada uma *escala*. Uma *escala* é simplesmente uma seqüência de números estritamente crescente ou decrescente em intervalos igualmente espaçados. Neste caso, cada grupo de cinco dígitos

está em ordem estritamente crescente, e, como tal (supondo-se que ele continue), não seria uma seqüência aleatória. Escalas também podem ser separadas por dígitos de ruído: os dígitos que compõem a escala podem estar dispersos por uma seqüência que, de outro modo, seria aleatória. É possível elaborar testes que detectem essas situações, mas eles vão além do propósito deste livro.

Uma outra característica a ser testada é o comprimento do *período*; ou seja, quantos números podem ser gerados antes que a seqüência comece a se repetir – ou pior, que degenere num ciclo curto. Todos os geradores de números aleatórios, baseados em computador, eventualmente repetem uma seqüência. Quanto mais longo o período, melhor o gerador. Mesmo que a freqüência dos números dentro do período seja uniformemente distribuída, isto não constitui uma série aleatória, pois uma série realmente aleatória não se repete de modo consistente. Geralmente, um período de diversos milhares de números é suficiente para a maioria das aplicações. (Novamente, pode se executar um teste para isso.)

Diversos outros testes podem ser aplicados para determinar a qualidade de um gerador de números aleatórios. De fato, têm havido provavelmente mais algoritmos escritos para testar geradores de números aleatórios do que para construí-los. Aqui está um outro teste que permite, através de um gráfico que mostra como uma seqüência de números aleatórios distribui seus valores, testar visualmente um gerador de números aleatórios. O ideal seria que o gráfico se baseasse na freqüência de cada número. Entretanto, isso seria impraticável, já que um gerador de números aleatórios pode produzir milhares de números diferentes. Em vez disso, você criará um gráfico agrupado pelo primeiro dígito de cada número; por exemplo, desde que todos os números aleatórios produzidos estejam entre 0 e 1, o número 0,9365783 será agrupado sob o 9, e o número 0,34523445 será agrupado sob o 3. Isso quer dizer que o gráfico de saída do programa de monitoração de números aleatórios tem 10 colunas, cada uma delas representando quantas vezes ocorreu um determinado número de um grupo. O programa também imprime a média de cada seqüência, que pode ser usada para detectar uma tendência dos números. Como os demais programas gráficos deste capítulo, este programa só roda num IBM PC que possua uma placa CGA. (*Nota do Revisor Técnico:* Este é o caso da maioria dos PCs nacionais.) Ambas as funções anteriormente desenvolvidas, **Ran1** e **Ran2**, assim como a função **Random**, interna do Turbo Pascal, são mostradas lado a lado para facilitar a comparação.

```
program Gerador_Ran;  {compara tres rotinas de geracao de numeros
                       randomicos}
const
   CONTA = 1000;

var
  freq1,freq2,freq3: array[1..9] of integer;
  a2,a1: integer;
```

```
  f,f2,f3: real;
  r,r2,r3: real;
  x,y: integer;

procedure Mostra;
var
   t: integer;

begin
   for t:=0 to 9 do
   begin
      Draw(t*10,180,t*10,180-freq1[t],2);
      Draw(t*10+110,180,t*10+110,180-freq2[t],2);
      Draw(t*10+220,180,t*10+220,180-freq3[t],2);
   end;
end;  {Mostra}

function Ran1: real;
var
   t: real;
begin
   t:=(a1*32749+3) mod 32749;
   a1:=Trunc(t);
   Ran1:=Abs(t/32749);
end;  {Ran1}

function Ran2: real;
var
   t: real;
begin
   t:=(a2*10001+3) mod 17417;
   a2:=Trunc(t);
   Ran2:=Abs(t/17417);
end;  {Ran2}

begin
   GraphColorMode;
   Palette(0);   {escolhe as cores do grafico}
   Gotoxy(1,1);
   Write('COMPARACAO ENTRE OS GERADORES');
   Gotoxy(1,2);
   Write('DE NUMEROS ALEATORIOS');
   Draw(0,180,90,180,3);
   Draw(110,180,200,180,3);
   Draw(220,180,310,180,3);
   Gotoxy(5,25);
   Write('  RANDOM          RAN1          RAN2');
   a1:=1; a2:=203;
   f:=0; f2:=0; f3:=0;

   for x:=0 to 9 do
   begin   {inicializa as matrizes de frequencia}
      freq1[x]:=0;
      freq2[x]:=0;
      freq3[x]:=0;
   end;
   for x:=1 to CONTA do
```

```
  begin
     r:=Random;    {produz um numero aleatorio}
     f:=f+r;       {soma para o calculo da media}
     y:=Trunc(r*10);  {converte r num inteiro entre 0 e 9}
     freq1[y]:=freq1[y]+1; {contagem da frequencia}

     r2:=Ran1;     {produz um numero aleatorio}
     f2:=f2+r2;    {soma para o calculo da media}
     y:=Trunc(r2*10);  {converte r num inteiro entre 0 e 9}
     freq2[y]:=freq2[y]+1; {contagem da frequencia}

     r3:=Ran2;     {produz um numero aleatorio}
     f3:=f3+r3;    {soma para o calculo da media}
     y:=Trunc(r3*10);  {converte r num inteiro entre 0 e 9}
     freq3[y]:=freq3[y]+1; {contagem da frequencia}

     Mostra;  {grafico das frequencias}
     Gotoxy(1,25);
     Write(x);
  end;
  ReadLn;
  GraphColorMode;

  WriteLn('A media de Random e'': ',f/CONTA);
  WriteLn('A media de Ran1 e'': ',f2/CONTA);
  WriteLn('A media de Ran2 e'': ',f3/CONTA);
  ReadLn;
  TextMode;
end.
```

Neste programa, cada função gera 1.000 números, e a função de freqüência apropriada é atualizada com base no primeiro dígito. Depois que cada número aleatório é gerado, o procedimento **Mostra** traça a matriz de todas as três freqüências na tela, de modo que você pode acompanhar o crescimento de cada uma. A Figura 8-2 mostra a saída de cada gerador de números aleatórios ao final dos 1.000 números. A média foi 0,489932 para **Ran1**, 0,4858311 para **Ran2** e 0,500279 para **Random.** Estes números são aceitáveis.

Para utilizar o programa eficientemente, você deve prestar atenção tanto na forma quanto no modo como o gráfico cresce para detectar os ciclos curtos. Por exemplo **Ran2** gera significativamente menos números entre 0,9 e 0,999999 (a barra da extrema direita) do que fazem tanto **Random** quanto **Ran1.**

É claro que esse teste não é conclusivo, mas ele lança uma luz no modo como o gerador produz seus números, e pode acelerar o processo de teste, permitindo que funções obviamente pobres sejam rejeitadas rapidamente. (Ele também é um belo programa para quando alguém pede que você lhe mostre o seu computador!)

USANDO GERADORES MÚLTIPLOS

Uma técnica simples que melhora a aleatoriedade das seqüências produzidas pelos três geradores é combiná-los sob o controle de uma função mestra. Essa função escolhe entre dois deles com base no resultado do terceiro. Com essa técnica você pode obter períodos muito longos e diminuir o efeito de qualquer ciclo ou tendência. A função chamada de **CombRandom**, mostrada aqui, combina **Ran1**, **Ran2** e **Random**:

```
function CombRandom: real;
{selecao aleatoria de geradores}
var
   f: real;

begin
   f:=Ran2;
   if f > 0.5 then CombRandom:=Random
   else CombRandom:=Ran1;  {selecao aleatoria}
end;  {CombRandom}
```

O resultado de **Ran2** é usado para decidir se **Ran1** ou **Random** determinará o valor da função mestra **CombRandom.** Com esse método, o período de **CombRandom** é igual ou maior que a soma do período de **Random** e **Ran1.** Portanto, esse método torna possível produzir seqüências com períodos muito longos. Sinta-se livre para alterar a mistura entre **Random** e **Ran1** através da mudança no **if,** obtendo a distribuição exata que você deseja entre **Random** e **Ran1.** Você pode, também, acrescentar geradores adicionais e selecionar entre eles, para obter períodos ainda mais longos.

Aqui está um programa para mostrar o gráfico de **CombRandom** e sua média. A Figura 8-3 mostra o gráfico final depois que 1.000 números aleatórios foram computados. A média de **CombRandom** foi 0,496833.

```
program Multi_Random;  {combina tres geradores de numeros randomicos
                        num so'}

const
   CONTR = 1000;

var
  freq: array[1..9] of integer;
  a2,a1: integer;
  f,r: real;
  x,y: integer;

procedure Mostra;
var
   t: integer;
```

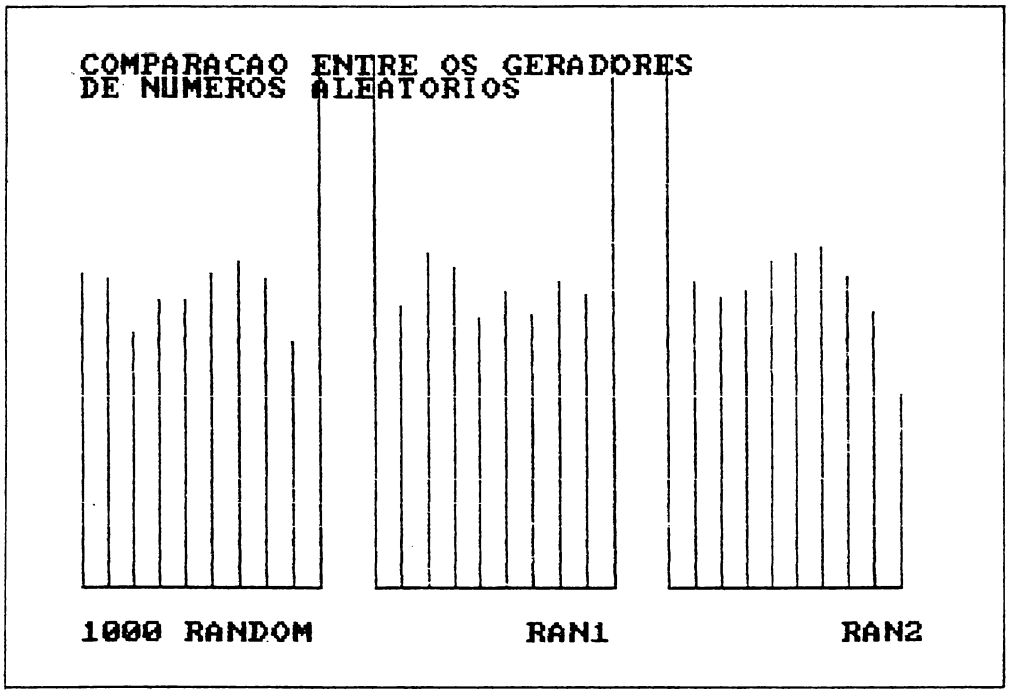


Figura 8-2 Saída do gerador de números aleatórios.

```
begin
   for t:=0 to 9 do
   begin
      Draw(t*10+110,180,t*10+110,180-freq[t],2);
   end;
end;  {Mostra}

function Ran1: real;
var
   t: real;
begin
   t:=(a1*32749+3)  mod 32749;
   a1:=Trunc(t);
   Ran1:=Abs(t/32749);
end;  {Ran1}

function Ran2: real;
var
   t: real;
begin
   t:=(a2*10001+3)  mod 17417;
   a2:=Trunc(t);
   Ran2:=Abs(t/17417);
end;  {Ran2}
```

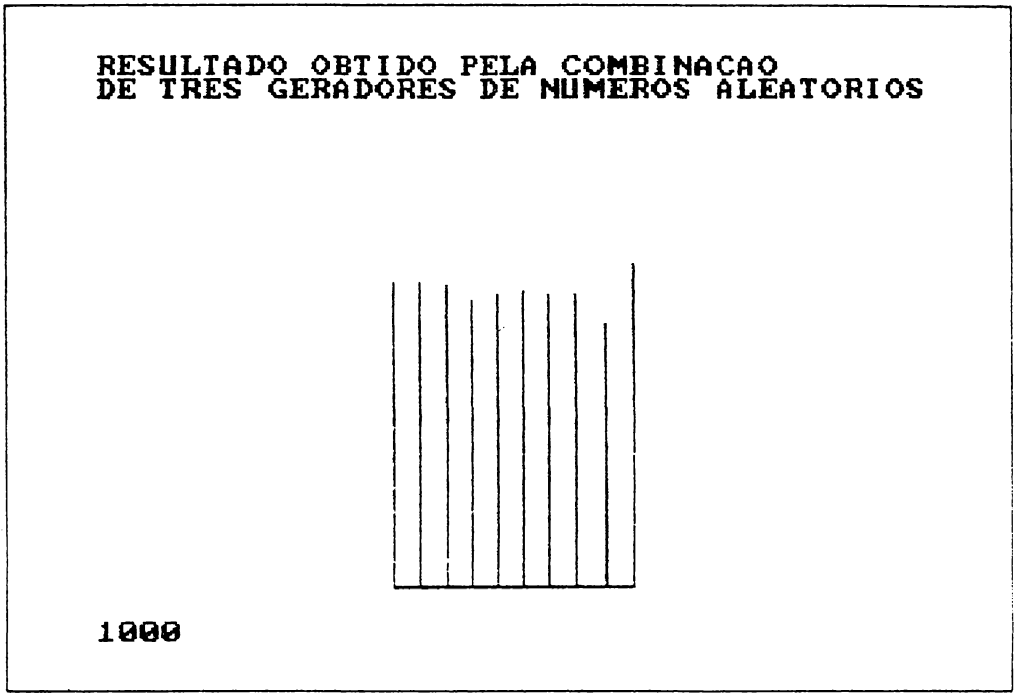


Figura 8-3 Gráfico final de CombRandom.

```
function CombRandom: real;
{selecao aleatoria de geradores}
var
   f: real;

begin
   f:=Ran2;
   if f > 0.5 then CombRandom:=Random
   else CombRandom:=Ran1;  {selecao aleatoria}
end;  {CombRandom}


begin
   GraphColorMode;
   Palette(0);   {escolhe as cores do grafico}

   Gotoxy(1,1);
   Write('RESULTADO OBTIDO PELA COMBINACAO');
   Gotoxy(1,2);
   Write('DE TRES GERADORES DE NUMEROS ALEATORIOS');
   Draw(110,180,200,180,3);
   Gotoxy(5,25);
```

```
   a1:=1; a2:=203;  {inicializa as variaveis dos geradores}
   f:=0;

   for x:=0 to 9 do freq[x]:=0;   {inicializa as matrizes de frequencia}

   for x:=1 to CONTR do
   begin
      r:=CombRandom;   {produz um numero aleatorio}
      f:=f+r;     {soma para o calculo da media}
      y:=Trunc(r*10);  {converte r num inteiro entre 0 e 9}
      freq[y]:=freq[y]+1; {contagem da frequencia}

      Mostra;  {grafico das frequencias}
      Gotoxy(1,25);
      Write(x);
   end;
   ReadLn;
   GraphColorMode;

   WriteLn('A media de CombRandom e'': ');
   WriteLn(f/CONTR);
   ReadLn;
   TextMode;
end.
```

SIMULAÇÕES

O restante deste capítulo examina a aplicação de geradores de números aleatórios às *simulações*. Uma simulação é um modelo computadorizado de uma situação do mundo real. Qualquer coisa pode ser simulada, mas o sucesso de uma simulação depende do conhecimento que o programador tem sobre o evento a ser simulado. Como muitas situações do mundo real têm milhares de variáveis, muitas delas são difíceis de se simular eficazmente. Entretanto, há diversas situações que se prestam para a simulação.

As simulações são importantes por duas razões. Primeiro, elas lhe permitem alterar os parâmetros de uma situação e observar os efeitos, quando, na realidade, tais experiências seriam muito caras ou perigosas. Por exemplo, a simulação de uma usina nuclear permite testar o efeito de certas falhas técnicas sem nenhum risco. Segundo, uma simulação permite a criação de situações impossíveis no mundo real. Por exemplo, um psicólogo poderia querer estudar os efeitos do aumento gradual da inteligência de um rato para ver em que nível ele percorre um labirinto mais rapidamente. Apesar disto ser impossível, na vida real, uma simulação pode lançar alguma luz na comparação entre inteligência e instinto. Segue o primeiro dos dois exemplos de simulação deste capítulo.

SIMULAÇÃO DE FILAS NUM SUPERMERCADO

O primeiro exemplo simula as filas que se formam nas caixas de um supermercado. Suponha que a loja fique aberta dez horas por dia, com horários de pico das 12 às 13 horas e das 17 às 18 horas. O intervalo das 12 horas é duas vezes, e o horário das 17 é três vezes mais movimentado que o normal. Durante a simulação, um gerador de números aleatórios "cria" fregueses, um segundo gerador determina o tempo de um freguês no caixa, e um terceiro gerador decide em que fila os fregueses que vão chegando ficarão. O objetivo da simulação é ajudar a gerência a determinar o número ótimo de caixas abertos durante um dia típico, de modo que nunca haja mais de dez pessoas por fila, e de modo que nenhum caixa aberto fique sem serviço.

O segredo deste tipo de simulação está na criação de processos paralelos. Apesar de o Turbo Pascal não permitir a simultaneidade de processos, é possível simular o multiprocessamento através da intercalação de diversas funções no fluxo de execução, como ocorre nos sistemas de *time-sharing*. Por exemplo, a função que simula a passagem de fregueses pelos caixas só processa uma parte dos fregueses de cada vez. Desta forma, os processos avançam mais ou menos ao mesmo tempo. O programa principal, sem os procedimentos e funções de apoio, está listado a seguir:

```
program fila_do_caixa;  {Simulacao de filas em caixas de supermercado}
{$I GRAPH.P} {incluir rotinas graficas}
var
   filas,conta: array[0..9] of integer;
   aberta: array[0..9] of boolean;
   fregues,tempo: integer;
   a1,a2: integer;
   y,x: integer;
   muda: boolean;

begin
   GraphColorMode;
   Palette(0);  {prepara modo grafico}

   a1:=0; a2:=203;  {inicializa variaveis do gerador aleatorio}

   muda:=FALSE;
   fregues:=0;
   tempo:=0;

   for x:=0 to 9 do
   begin
      filas[x]:=0;  {inicializa filas}
      aberta[x]:=FALSE;  {nenhum fregues ou caixa aberto no inicio}
      conta[x]:=0;  {conta filas}
   end;
   GotoXY(20,24); Write('1           10');
   GotoXY(1,24); Write('F I L A S');
```

```
   {comeca o dia abrindo o primeiro caixa}
   aberta[0]:=TRUE;

   repeat
      Addfregues;
      Addfila;
      Mostra;
      Passa;
      Mostra;
      if (tempo > 30) and (tempo < 50) then Addfregues;
      if (tempo > 70) and (tempo < 80) then
      begin
         Addfregues;
         Addfregues;
      end;
      tempo:=tempo+1;
   until tempo > 100;  {fim do dia}
   ReadLn;
   TextMode;
end.
```

O arquivo **GRAPH.P** foi incluído de modo que o programa possa utilizar a função gráfica ampliada **Circle.**

O *loop* principal dirige toda a simulação:

```
repeat
   AddCust;
   AddQueue;
   Display;
   CheckOut;
   Display;
   if (time > 30) and (time < 50) then AddCust;
   if (time > 70) and (time < 80) then
   begin
      AddCust;
      AddCust;
   end;
   time:=time+1;
until time > 100;  {end of day}
```

A função **MaisFregueses** utiliza **Ran1** ou **Random** para gerar o número de fregueses que chega às caixas a cada requisição. **FilaCresce** é usada para, de acordo com os resultados de **Ran2,** colocar fregueses na fila de um caixa aberto, além de abrir uma fila nova, caso as já abertas estejam cheias. **Desenha** mostra um gráfico da simulação. **Caixa** usa **Ran2** para atribuir uma lista de compras a cada freguês; cada chamada reduz uma unidade da lista. Quando a lista de um freguês for 0 ele deixará o caixa.

A variável **tempo** altera a taxa com que são gerados os fregueses, de modo a corresponder às horas de pico da loja. Cada passo através do *loop* representa um décimo de hora.

As Figuras 8-4, 8-5 e 8-6 mostram o estado de cada caixa quando **tempo**=28, **tempo**=60 e **tempo**=88, correspondendo ao tempo normal, ao fim do primeiro pico, e ao fim do segundo pico, respectivamente. Note que no final do segundo pico, é necessário um máximo de 5 caixas. Se a simulação foi programada adequadamente, isso significa que o supermercado não precisa operar os outros cinco caixas.

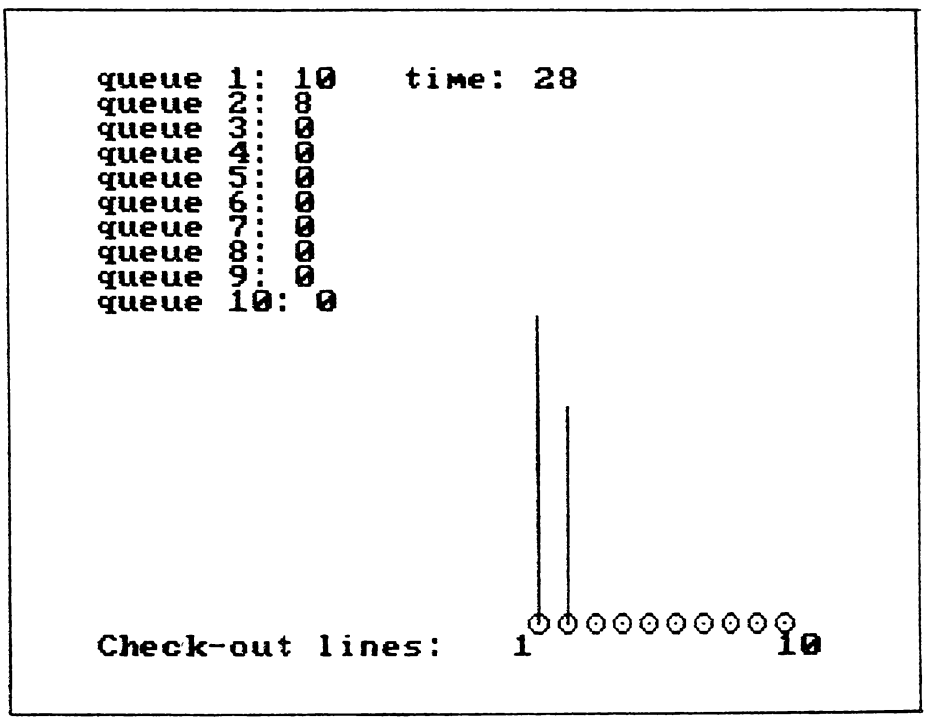


Figura 8-4 Estado dos caixas quando **tempo**=28.

Você pode controlar, diretamente, diversas variáveis do programa. Primeiro, você pode alterar o modo como os fregueses chegam e o número deles. Você também pode alterar **MaisFregueses** para que à medida que as horas de pico se vão, retornem gradualmente mais ou menos fregueses. O programa supõe ainda que os fregueses escolherão aleatoriamente a fila na qual se colocarão. Embora isso possa ser verdade para alguns fregueses, outros obviamente escolherão as filas mais curtas. Você pode computar

isso alterando a função **FilaCresce** para colocar, algumas vezes, fregueses na fila mais curta e outras vezes aleatoriamente. A simulação não leva em conta acidentes ocasionais – como uma garrafa de *ketchup* quebrada – ou um freguês recalcitrante no caixa, que fariam com que a fila "empacasse" temporariamente.

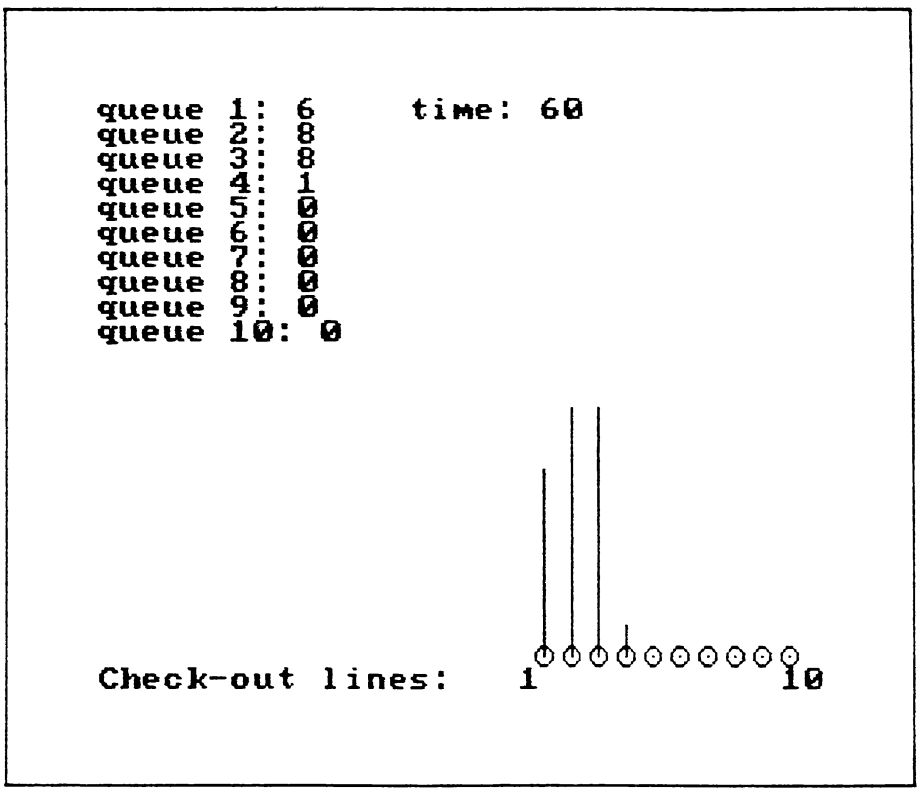


Figura 8-5 Estado dos caixas quando **tempo** = 60.

Mostramos aqui o programa completo:

```
program fila_do_caixa;  {Simulacao de filas em caixas de supermercado}
{$I GRAPH.P} {incluir rotinas graficas}
var
   filas,conta: array[0..9] of integer;
   aberto: array[0..9] of boolean;
   fregues,tempo: integer;
   a1,a2: integer;
   y,x: integer;
   muda: boolean;
```

```
function Ran1: real;
var
   t: real;
begin
   t:=(a1*32749+3)  mod 32749;
   a1:=Trunc(t);
   Ran1:=Abs(t/32749);
end;  {Ran1}

function Ran2: real;
var
   t: real;
begin
   t:=(a2*10001+3)  mod 17417;
   a2:=Trunc(t);
   Ran2:=Abs(t/17417);
end;  {Ran2}

function CombRandom: real;
{selecao aleatoria de geradores}
var
   f: real;

begin
   f:=Ran2;
   if f > 0.5 then CombRandom:=Random
   else CombRandom:=Ran1;  {selecao aleatoria}
end;  {CombRandom}

procedure Maisfregues;
var
   f,r: real;
begin
   if muda then f:=Random  {alterna entre}
   else f:=Ran2;             {os geradores}

   if f > 0.5 then
      if f < 0.6 then fregues:=fregues+1       {Mais um fregues}
      else if f < 0.7 then fregues:=fregues+2  {Mais dois fregueses}
      else if f < 0.8 then fregues:=fregues+3  {Mais tres fregueses}
      else fregues:=fregues+4;

end;  {Maisfregues}

procedure Passa;
var
   t: integer;

begin
   for t:=0 to 9 do
   begin
      if filas[t] <> 0 then
      begin
         {gera tempo necessario para passar pelo caixa}
         while conta[t]=0  conta[t]:=Trunc(Ran1*5);
         {outro fregues}
         conta[t]:=conta[t]-1;
         if conta[t]=0 then filas[t]:=filas[t]-1;
```

```
         {remove o fregues}
      end;
      if filas[t]=0 then aberto[t]:=FALSE;  {fecha o caixa t}
   end;
end;  {Passa}

function Caixas_Lotados: boolean;
{verifica se caixas estao lotados}
var
   t: integer;

begin
   Caixas_Lotados:=TRUE;
   for t:=0 to 9 do
      if (filas[t] < 10) and aberto[t] then Caixas_Lotados:=FALSE;
end;  {Caixas_Lotados}

procedure Maisfila;
{aberto check-out fila}
var
   t,fila: integer;
   feito: boolean;

begin
   feito:= FALSE;
   while fregues <> 0 do
   begin
      if Caixas_Lotados then
      begin
         t:=0;
         repeat
            if not aberto[t] then
            begin
               aberto[t]:=TRUE;
               feito:=TRUE;
            end;
            t:=t+1;
         until feito or (t=9);
      end
      else
      begin
         fila:=Trunc(Ran2*10);
         if aberto[fila] and (filas[fila] < 10) then
         begin
            filas[fila]:=filas[fila]+1;
            fregues:=fregues-1;
         end;
      end;
      if Caixas_Lotados and aberto[9] then fregues:=0;  {todos lotados}
   end;
end;  {Maisfila}

procedure Mostra;
var
   t: integer;

begin
   GotoXY(15,1);
   Write('Tempo: ',tempo);
```

```
    for t:=0 to 9 do
    begin
       Draw((t*10)+160,180,(t*10)+160,80,0);
       Circle((t*10)+160,180,3,3);
       Draw((t*10)+160,180,(t*10)+160,180-filas[t]*10,2);
       GotoXY(1,1+t);
       Write('Fila ',t+1,': ', filas[t],' ');
    end;
end;  {Mostra}

begin
    GraphColorMode;
    Palette(0);  {prepara modo grafico}

    a1:=0; a2:=203;  {inicializa variaveis do gerador aleatorio}

    muda:=FALSE;
    fregues:=0;
    tempo:=0;

    for x:=0 to 9 do
    begin
       filas[x]:=0;  {inicializa filas}
       aberto[x]:=FALSE;  {nenhum fregues ou caixa aberto no inicio}
       conta[x]:=0;  {conta filas}
    end;
    GotoXY(20,24); Write('1           10');
    GotoXY(10,24); Write('Filas -->');

    {comeca o dia abrindo o primeiro caixa}
    aberto[0]:=TRUE;

    repeat
       Maisfregues;
       Maisfila;
       Mostra;
       Passa;
       Mostra;
       if (tempo > 30) and (tempo < 50) then Maisfregues;
       if (tempo > 70) and (tempo < 80) then
       begin
          Maisfregues;
          Maisfregues;
       end;
       tempo:=tempo+1;
    until tempo > 1000;  {fim do dia}
    ReadLn;
    TextMode;
end.
```

MONITORAÇÃO DE CARTEIRAS DE AÇÕES

A arte do gerenciamento de carteiras de ações é geralmente baseada em várias teorias e suposições sobre muitos fatores, alguns dos quais não podem ser facilmente conhecidos, a

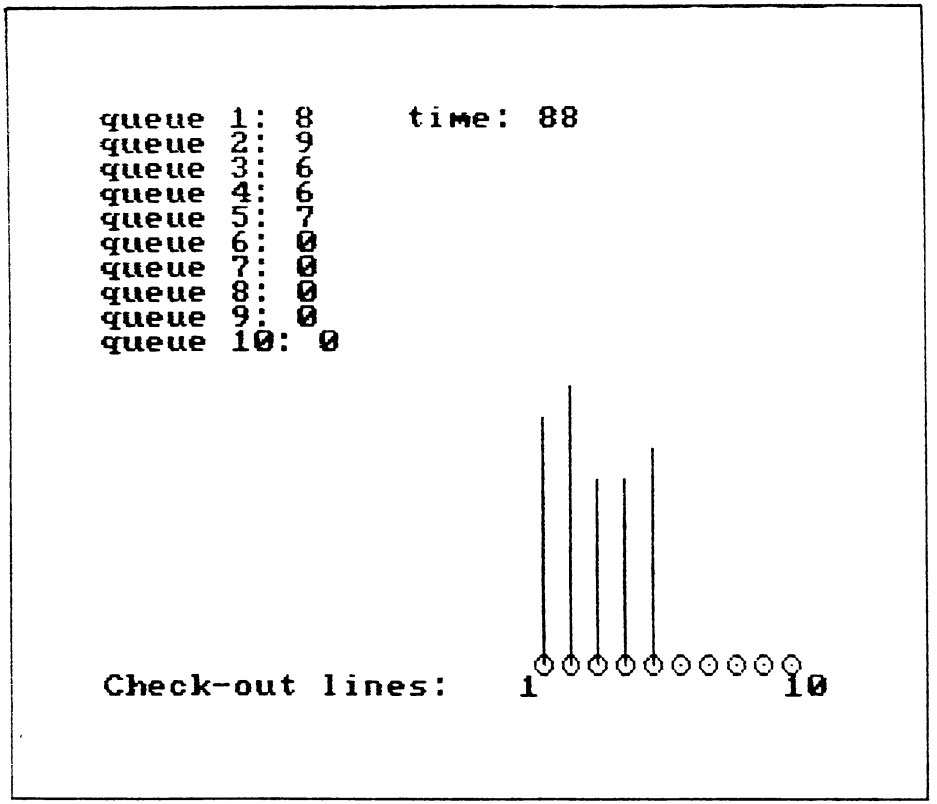


Figura 8-6 Estado da fila de caixas quando **tempo** = 88.

menos que você seja do meio. Existem estratégias de compra e venda, baseadas em análises estatísticas de preços de ações, de preço do ouro, de PNB, e até mesmo de ciclos da lua. A vingança do cientista da computação é usar o computador para simular o mercado livre – a bolsa de ações – sem todos os aborrecimentos da vida real.

Você pode pensar que o mercado de ações é simplesmente difícil demais para ser simulado; que ele possui variáveis demais, muitas das quais desconhecidas, e que algumas vezes muda bruscamente enquanto em outras navega placidamente. Entretanto, o próprio problema é a solução: devido ao fato de o mercado ser tão complexo, pode ser pensado como sendo composto de *eventos de ocorrência aleatória*. Isto significa que você pode simular o mercado de ações como uma série de ocorrências de eventos desligados e aleatórios.

Essa simulação é carinhosamente chamada de método cambaleante de administração de carteiras. O termo origina-se do clássico experimento que envolve um bêbado perambulando na rua, vagando de um poste a outro. Com a teoria do *random-walk* você deixa que a sorte seja seu guia, já que ela é tão boa quanto qualquer outra.

Antes de ir adiante, esteja avisado: o método *random-walk* é desacreditado por corretores profissionais. Ele é apresentado aqui apenas para seu divertimento, não para ajudá-lo a realizar decisões de investimentos.

Para aplicar o método *random-walk*, selecione dez companhias do *Wall Street Journal* por algum métdo casual – por exemplo, jogando dardos no jornal e usando as companhias cujo nome você acertar. Depois de ter escolhido dez companhias, alimente o programa de simulação de andar aleatório com elas.

O programa lhe dirá sempre para fazer uma destas cinco transações:

- Vender
- Comprar
- Especular
- Comprar na margem
- Segurar (não fazer nada)

A operação de vender, comprar e segurar ações são auto-explicativas. Aqui, estamos chamando de *especular* à operação de vender ações das quais não se tenha esperança de comprá-las mais barato e entregá-las rapidamente ao comprador. Esta é uma maneira de ganhar dinheiro quando o mercado está caindo. Quando você compra na margem, usa (por uma pequena taxa) o dinheiro de corretagem da casa para financiar parte do custo das ações que adquiriu. A idéia que está por trás de comprar na margem é que a ação suba o suficiente, e você ganhe mais dinheiro do que ganharia se tivesse comprado uma quantidade menor de ações à vista. Isso só dá dinheiro em um mercado com tendência a subir muito.

O programa *random-walk* é mostrado aqui. A função interna **KeyPressed** checa o estado do teclado e espera pela pressão de uma tecla. Isso permite que você use a seqüência produzida pelo gerador de números aleatórios em um ponto aleatório – basicamente, criando um valor somente aleatório, que evita que o programa dê sempre o mesmo conselho.

```
program Acoes; {Programa de investimentos por 'random walk'}
type
   str80 = string[80];
   fazer = (comprar,vender,esperar,especular,emprestar);
var
   t: integer;
   acoes: array[1..10] of str80;
   ch: char;
   transacao: fazer;
   f: real;

procedure Entrada;  {Digitacao dos nomes das acoes}
var
   t: integer;

begin
   for t:=1 to 10 do
   begin
      Write('Digite o nome da acao: ');
      ReadLn(acoes[t]);
   end;
end;  {Entrada}

function Fazer_a_seguir: fazer;
var
   f: real;

begin
   Fazer_a_seguir:=esperar;
   case Trunc(Random*10) of
      1: Fazer_a_seguir:=vender;
      2: Fazer_a_seguir:=comprar;
      3: Fazer_a_seguir:=especular;
      4: Fazer_a_seguir:=emprestar;
   end;
end;  {Fazer_a_seguir}

begin
   Write('Espere um pouco e pressione uma tecla...');
   repeat
      f:=Random;  {inicializa gerador}
   until KeyPressed;

   WriteLn;
   WriteLn('Digitar novas acoes? (S/N) ');
   Read(kbd,ch);
   if UpCase(ch)='S' then Entrada;

   repeat
      for t:=1 to 10 do
      begin
         transacao:=Fazer_a_seguir;
         if Length(acoes[t]) > 0 then
         begin
            Write(acoes[t],':');
            case transacao of
               comprar: WriteLn('comprar');
```

```
                vender: WriteLn('vender');
                especular: WriteLn('vender especular');
                emprestar: WriteLn('comprar emprestando');
                esperar: WriteLn('esperar');
              end;
          end;
      end;
      WriteLn('Mais uma vez? (S/N) ');
      Read(kbd,ch);
   until UpCase(ch)='N';
end.
```

O programa requer que você interprete suas instruções da seguinte maneira:

Instrução	**Interpretação**
compre	Compre tantas ações quantas puder, sem pedir dinheiro emprestado.
venda	Caso você não possua alguma ação, venda todas. Então selecione, aleatoriamente, uma outra companhia para reinvestir seu dinheiro.
especule	Venda 100 ações da companhia especificada, mesmo que você não as possua, na esperança de comprar mais barato no futuro.
compre na margem	Tome dinheiro emprestado para comprar ações da companhia especificada.
segure	Não faça nada.

Por exemplo, se você fosse rodar este programa usando os nomes fictícios de companhias de *Com 1* a *Com 10*, os conselhos do primeiro dia poderiam ser mais ou menos assim:

Com 1: venda
Com 2: compre
Com 3: compre na margem
Com 4: especule
Com 5: segure
Com 6: segure
Com 7: segure
Com 8: venda
Com 9: segure
Com 10: especule

Os conselhos do segundo dia poderiam ser:

Com 1: segure
Com 2: segure
Com 3: venda

Com 4: especule
Com 5: segure
Com 6: segure
Com 7: compre
Com 8: compre na margem
Com 9: segure
Com 10: venda

Lembre-se: devido ao fato do programa esperar que você pressione uma tecla, sua saída diferirá daquela mostrada aqui. Você pode preferir rodar o programa semanalmente ou mensalmente em vez de diariamente.

Esteja livre para alterar o programa de qualquer maneira. Por exemplo, o programa poderá dar números de ações para comprar, dependendo do dinheiro que você disponha para investimento. Lembre-se, novamente, de que este programa é apenas para diversão e que não é um modo recomendado de efetuar realmente investimentos no mercado.

De qualquer modo, é interessante criar uma carteira de papéis e seguir o seu desempenho.

CAPÍTULO **9**

ANÁLISE E RESOLUÇÃO DE EXPRESSÕES

Escrever um programa que analise e resolva uma expressão numérica qualquer, 10–5*3 por exemplo, é um problema que a imensa maioria dos programadores prefere ignorar, deixando a solução a cargo de alguns iluminados, sumo-sacerdotes aptos a conhecer os segredos mais profundos da máquina. Quase todos os usuários ficam impressionados pelo modo como compiladores e interpretadores de linguagens de alto nível convertem expressões tão complexas quanto 10*3–(4+Const)/12 em instruções executáveis por um computador. Este processo, chamado *análise de expressões (expression parsing)*, é a espinha dorsal de todos os compiladores e interpretadores, e poucos são os programadores capazes de escrever um analisador de expressões.

Não deveria ser assim. A análise de expressões é, na verdade, um processo muito direto e um problema muito similar a várias outras tarefas de programação. Sob certos aspectos, um analisador de expressões é muito mais simples do que parece, pois ele trabalha apenas com as regras da álgebra. Neste capítulo será apresentado um *analisador recursivo decrescente (recursive descent parser)*, bem como todas as rotinas de apoio que permitirão a resolução de expressões numéricas complexas. Todas estas rotinas serão colocadas num só arquivo, a ser usado quando necessário. Após dominar o uso de tal arquivo, você poderá modificá-lo e expandi-lo, amoldando-o às suas necessidades, tornando-se, você também, um "iluminado".

EXPRESSÕES

Apesar de expressões poderam ser formadas com qualquer tipo de informação, este capítulo se concentrará num só tipo: as *expressões numéricas*. Assumiremos que expressões numéricas podem conter os seguintes dados:

- Números
- Os operadores +, –, /, *,^ e =
- Parênteses
- Variáveis

O símbolo "^" indica exponenciação, como no BASIC, e o símbolo "=" representa o operador de atribuição. São seguidas as regras normais de álgebra, com as quais você está familiarizado. Alguns exemplos de expressões são:

```
10–8
(100–5)*14/6
a+b+c
10^5
a=10–b
```

Os operadores seguem a ordem de precedência algébrica normal, ou seja:

```
(mais alta)     ^
                * /
                + –
(mais baixa)     =
```

Operadores de igual precedência são resolvidos da esquerda para a direita.

Nos exemplos deste capítulo foram adotadas algumas convenções. Todos os nomes de variável têm apenas uma letra, resultando, portanto, em 26 variáveis disponíveis. Todos os números são do tipo **integer.** Finalmente, pouca verificação de erros foi incluída nas rotinas, isto para tornar a lógica usada mais clara e ordenada. Naturalmente, qualquer dos aspectos citados acima pode ser facilmente modificado.

Resolva a expressão

```
10–2*3
```

O resultado correto é 4. É fácil escrever um programa que resolva esta expressão específica, mas você deve estar imaginando um meio de criar um programa que resolva qualquer expressão dada. Num primeiro momento, pode ter lhe ocorrido algo como a rotina abaixo:

```
a:=pega o primeiro operando
while (operandos presentes) do
begin
    op:=pega operador
    b:=pega segundo operando
    a:=a op b
end;
```

Esta rotina pega o primeiro e o segundo operandos, o operador entre eles e realiza a operação, repetindo tal processo até que não existam mais operandos presentes. Entretanto, a resposta desta rotina para a expressão 10–2*3 é 24 (8*3) e não 4 (a resposta correta), pois o procedimento ignora a precedência dos operadores. Não é possível resolver uma expressão automaticamente, da esquerda para a direita, pois uma multiplicação, por exemplo, deve ser resolvida antes de uma subtração. Tal programa pode parecer, à primeira vista, facilmente superável, mas ele só se torna mais complexo com a inclusão de parênteses, exponenciações, variáveis, chamadas de função etc.

Existem vários métodos para resolver expressões deste tipo, mas nós só estudaremos um, o mais simples e mais comum (alguns métodos de construir analisadores utilizam tabelas que requerem quase outro programa para serem geradas. São os chamados *analisadores controlados por tabela (table driven parsers)*. O método que examinaremos é chamado analisador recursivo descendente, e ao longo deste capítulo você entenderá como ele obteve este nome.

DESMEMBRANDO UMA EXPRESSÃO

Antes de desenvolver uma rotina para resolver uma expressão, você deve ser capaz de separar facilmente partes da expressão. Por exemplo, dada a expressão

$$A*B-(W + 10)$$

você deve obter os operandos, *A*, *B*, *W* e 10, os parênteses e os operadores *, – e +. É necessária, portanto, uma rotina que tome cada item da expressão individualmente. Tal

rotina deve também ignorar os espaços em branco e saber quando o final da expréssão foi atingido.

Cada item da expressão é um símbolo. Assim, a função que obtém os símbolos da expressão é chamada **Pega_Simb**. Uma seqüência de caracteres global é usada para conter a expressão. Em **Pega_Simb**, tal seqüência se chama **prog**. A variável **prog** é global, pois deve manter seu conteúdo inalterado entre cada chamada de **Pega_Simb**, além de permitir que outras funções a usem. O inteiro global **t** é usado como índice de **prog**, permitindo que **Pega_Simb** avance pela expressão, um símbolo de cada vez. **Pega_Simb** assume que **prog** termina com um cifrão ($). O "$" indicará o fim da expressão.

É preciso, também, determinar que tipo de símbolo foi obtido. O analisador desenvolvido neste capítulo só utiliza três tipos de símbolo: **VARIAVEL, NUMERO** e **DELIMITADOR**, sendo **DELIMITADOR** usado tanto para operadores quanto para parênteses. Eis aqui, então, o procedimento **Pega_Simb**, com todas as suas variáveis globais, declarações e funções de apoio:

```
type
  str80=string[80];
  T_El=(DELIMITADOR,VARIAVEL,NUMERO);

var
  Elemento,Prog:str80;
  Tipo_El:T_El;
  PosErro,T:integer;
  Resultado:real;

function Letra(ch:char):boolean;
{ devolve TRUE se for uma letra }
begin
  Letra:=(UpCase(ch)>='A') and (UpCase(ch)<='Z');
end; { Letra }

function Branco(ch:char):boolean;
{ devolve TRUE se for espaco, TAB ou CR }
begin
  Branco:=(ch=' ') or (ch=chr(9)) or (ch=chr(13));
end; { Branco }

function Delim(ch:char):boolean;
{ devolve TRUE se for um dos delimitadores
begin
  if pos(ch,' +-/*%^=()$')<>0 then Delim:=true
                              else Delim:=false;
end; { Delim }

function Digito(ch:char):boolean;
{ devolve TRUE se for um digito de 0 a 9 }
begin
  Digito:=(ch>='0') and (ch<='9');
end; { Digito }
```

```
procedure Pega Simb;
var
  temp:str80;

begin
  Elemento:='';  { string vazio }
  while(Branco(Prog[I])) do I:=I+1;  { pula os proximos espacos }
  if Prog[T]='$' then Elemento:='$';
  if pos(Prog[T],'+-*/%^=()')<>0 then
  begin
    Tipo_El:=DELIMITADOR;
    Elemento:=Prog[T];     { e' um operador }
    T:=T+1;
  end else if Letra(Prog[T]) then
  begin
    while (not Delim(Prog[T])) do
    begin
      Elemento:=concat(Elemento,Prog[T]);   { constroi a variavel }
      T:=T+1
    end;
    Tipo_El:=VARIAVEL;
  end else if Digito(Prog[T]) then
  begin
    while (not Delim(Prog[T])) do
    begin
      Elemento:=concat(Elemento,Prog[T]);   { constroi o numero }
      T:=T+1
    end;
    Tipo_El:=NUMERO;
  end;
end; { Pega Simb }
```

Antes que este procedimento seja usado é preciso fixar a variável global **t** em 1. Esta variável é usada como índice da seqüência **prog**, a qual contém a expressão a ser desmembrada. O primeiro passo do procedimento é verificar a presença do indicador $. Em seguida, os espaços em branco que porventura existam na frente da expressão são eliminados. Tais espaços podem ser usados para tornar a expressão mais legível, mas serão ignorados pelo procedimento.

Uma vez descartados os espaços em branco, **prog[t]** deverá conter um número, uma variável, um operador ou um $ (caso não haja nenhuma expressão presente). Caso este primeiro caractere seja um operador, ele será passado como uma seqüência de caracteres para a variável global **elemento**, e o tipo **DELIMITADOR** será colocado em **Tipo_el**. Se o caractere for uma letra, a rotina assumirá que ele é uma das variáveis, passando-o para **elemento; Tipo_el** terá como valor **VARIAVEL**. Se o caractere for um inteiro ele será passado para **elemento** com o tipo **NUMERO**. Finalmente, se o caractere não for nenhum dos acima citados, pode-se presumir que o final da expressão foi alcançado, e **elemento** será $.

Para manter esta rotina legível, muito da checagem de erros foi omitido, e algumas suposições foram feitas. Por exemplo, qualquer caractere não reconhecido será ignorado. Além disto, as variáveis podem ter nomes de qualquer tamanho, mas apenas a primeira letra é significativa. Estes e outros detalhes podem, porém, ser facilmente incluídos se necessário. **Pega_Simb** pode ser modificado ou expandido para trabalhar com seqüências de caracteres, números racionais, ou qualquer outra coisa.

Para compreender melhor o funcionamento de **Pega_Simb** examine a lista de símbolos e tipos de símbolo obtidos com a seguinte expressão: $A + 100 - (B*C)/2$.

Símbolo	Tipo de símbolo
A	**VARIAVEL**
+	**DELIMITADOR**
100	**NUMERO**
-	**DELIMITADOR**
(	**DELIMITADOR**
B	**VARIAVEL**
*	**DELIMITADOR**
C	**VARIAVEL**
)	**DELIMITADOR**
/	**DELIMITADOR**
2	**NUMERO**
$	**Fim da Expressão**

ANÁLISE DE EXPRESSÕES

Existem muitas maneiras de analisar e resolver uma expressão. No contexto deste capítulo, expressões são estruturas recursivas de dados, definidas em termos de si mesmas. Se restringirmos as expressões ao uso dos operadores +, –, *, / e dos parênteses, qualquer expressão poderá ser definida pelas seguintes regras:

expressão = > *termo*[+*termo*][–*termo*]
termo = > *fator*[**fator*][/*fator*]
fator = > *variável*, *número* ou (*expressão*)

onde qualquer parte pode ser nula. Os colchetes significam "opcionais" e as setas, "produz". Estas são as chamadas "regras de produção" da expressão. Por exemplo, a

segunda regra é lida "termo produz fator multiplicado por fator, ou fator dividido por fator". A ordem de precedência dos operadores está implícita no próprio modo de definição da expressão.

A expressão

$$10+5*B$$

tem dois termos: 10 e $5*B$. Ela tem, entretanto, três fatores, 10, 5 e B. Estes fatores consistem em dois números e uma variável. Já a expressão

$$14*(7-C)$$

consiste em dois termos, 14 e $(7-C)$ – um número e uma expressão entre parênteses. A expressão entre parênteses possui um número e uma variável.

Este processo forma o núcleo de um analisador recursivo descendente, que é, basicamente, um grupo de rotinas mutuamente recursivas trabalhando encadeadas. A cada passo, o analisador pode realizar as operações especificadas na seqüência algébrica correta. Examine melhor como isto funciona, seguindo a análise e resolução da expressão 9/3–(100+56):

1. Tome o primeiro termo: 9/3.
2. Tome cada fator do primeiro termo e divida os inteiros. O valor obtido é 3.
3. Tome o segundo termo: (100+56). Aqui, a segunda expressão deve ser analisada recursivamente.
4. Some os fatores. O resultado é 156.
5. Saindo da chamada recursiva, subtraia 156 de 3, chegando ao resultado final, –153.

Neste ponto, você pode estar um pouco confuso, mas não se preocupe. Este é um conceito bastante complexo, que requer algum tempo para ser entendido. Existem dois aspectos importantes na visão recursiva de expressões aqui apresentadas. Primeiro, a ordem de precedência das operações está *implícita* nas próprias regras de produção. Segundo, este método de analisar e resolver uma expressão é muito similar ao método usado para fazer o mesmo sem um computador.

UM ANALISADOR SIMPLES

Dois analisadores serão desenvolvidos neste capítulo. O primeiro analisa e resolve apenas expressões que não contenham variáveis, sendo esta a forma mais simples de um analisador. O segundo analisador inclui 26 variáveis possíveis (de A a Z), permitindo também que sejam atribuídos valores a cada variável usada.

Abaixo, a listagem completa da versão mais simples do analisador recursivo decrescente, para expressões formadas apenas por números inteiros. Ele utiliza os mesmos dados globais do procedimento **Pega_Simb.**

```
{ ****************** Analisador de Expressoes ****************** }

procedure Nivel2(var Resultado:real); forward;
procedure Nivel3(var Resultado:real); forward;
procedure Nivel4(var Resultado:real); forward;
procedure Nivel5(var Resultado:real); forward;
procedure Nivel6(var Resultado:real); forward;
procedure Primitiva(var Resultado:real); forward;

procedure Pega_Exp(var Resultado:real);
begin
  Pega_Simb;
  if length(Elemento)<>0 then
    Nivel2(Resultado)
  else
    Erro(3);
end; { Pega_Exp }

procedure Nivel2;
var
  Op:char;
  Guarda:real;

begin
  Nivel3(Resultado);
  Op:=Elemento[1];
  while((Op='+') or (Op='-')) do
  begin
    Pega_Simb;
    Nivel3(Guarda);
    Arit(Op,Resultado,Guarda);
    Op:=Elemento[1];
  end;
end; { Nivel2 }

procedure Nivel3;
var
  Op:char;
  Guarda:real;
```

```
begin
  Nivel4(Resultado);
  Op:=Elemento[1];
  while((Op='*') or (Op='/')) do
  begin
    Pega_Simb;
    Nivel4(Guarda);
    Arit(Op,Resultado,Guarda);
    Op:=Elemento[1];
  end;
end; { Nivel3 }

procedure Nivel4;
var
  Guarda:real;

begin
  Nivel5(Resultado);
  if Elemento[1]='^' then
  begin
    Pega_Simb;
    Nivel4(Guarda);
    Arit('^',Resultado,Guarda);
  end;
end;  { Nivel4 }

procedure Nivel5;
var
  Op:char;

begin
  Op:=' ';
  if((Tipo_El=DELIMITADOR) and ((Elemento[1]='+') or (Elemento[1]='-')))
  then
  begin
    Op:=Elemento[1];
    Pega_Simb;
  end;
  Nivel6(Resultado);
  if Op='-' then Resultado:=-Resultado;
end; { Nivel5 }

procedure Nivel6;
begin
  if (Elemento[1]='(') and (Tipo_El=DELIMITADOR) then
  begin
    Pega_Simb;
    Nivel2(Resultado);
    if Elemento[1]<>')' then Erro(2); { falta parenteses }
    Pega_Simb;
  end
  else Primitiva(Resultado);
end; { Nivel6 }

procedure Primitiva;
begin
  if Tipo_El=NUMERO then val(Elemento,Resultado,PosErro)
                    else Erro(1);
  Pega_Simb;
end; { Primitiva }
```

Este analisador aceita os operadores +, –, *, /, a exponenciação (^), inteiros negativos e parênteses. Ele possui seis níveis, além da função **Primitiva,** que produz o valor de um número inteiro. O comando **forward** é necessário porque algumas destas rotinas são mutuamente recursivas, sendo, portanto, impossível definir todos os procedimentos antes de chamá-las.

Ao programa anterior devem ser acrescentadas algumas rotinas especiais: **Erro,** que imprime mensagens de erro, e **Pot** e **Arit,** que realizam várias operações aritméticas. Estes subprogramas são os seguintes:

```
Procedure Serror(i:integer); {print error messages}
begin
   case i of
     1: WriteLn('Syntax Error');
     2: WriteLn('Unbalanced Parentheses');
     3: Writeln('No Expression Present');
   end;
end; {Serror}

function Pwr(a,b:real):real;
{raise a to b power}
var
   t: integer;
   temp:real;
begin
   if a=0 then Pwr:=1 else
   begin
      temp:=a;
      for t:=trunc(b) downto 2 do a:=a*temp;
      Pwr:=a;
   end;
end; {Pwr}

procedure Arith(op:char; var result,operand:real);
{perform arithmetic functions}
begin
   case op of
      '+':result:=result+operand;
      '-':result:=result-operand;
      '*':result:=result*operand;
      '/':result:=result/operand;
      '^':result:=Pwr(result,operand);
   end;
end; {Arith}
```

Como já foi visto, o procedimento global **Pega_Simb** busca, na expressão, o próximo símbolo e seu tipo. A seqüência de caracteres **prog** contém a expressão. A seguir está transcrito todo o analisador, suas rotinas de apoio, e um programa simples de demonstração.

```
program analisador; {Analisador para numeros reais e operadores,
                     sem variaveis}

type
    str80=string[80];
    Nome_Simb=(DELIMITADOR, VARIAVEL, NUMERO);
var
   simbolo,exp:str80;
   Tipo_Simb:Nome_Simb;
   codigo,t:integer;
   resultado:real;

function Letra(car:char):boolean;
{TRUE se car e' uma letra do alfabeto}

begin
     Letra:=(UpCase(car)>='A') and (UpCase(car)<='Z');
end;   {Letra}

function Branco(car:char):boolean;
{TRUE se return,espaco ou tab}

begin
     Branco:=(car=' ') or (car=chr(9)) or (car=chr(13));
end;   {Branco}

function Delim(car:char):boolean;
{TRUE se for um DELIMITADOR}

begin
     if pos(car,' +-/*%^=()$')<>0 then Delim:=TRUE
     else Delim:=FALSE;
end;  {Delim}

function Digito(car:char) :boolean;
{TRUE se for um digito entre 0 e 9}

begin
     Digito:=(car>='0') and (car<='9');
end;   {Digito}

procedure Analisa_Simb;
var
   temp:str80;

begin
     simbolo:='';  {sequencia de caracteres nula}
     while(Branco(exp[t])) do t:=t+1; {retira os espacos antes da expressao}
     if exp[t]='$' then simbolo:='$';
     if pos(exp[t],'+-*/^=()')<>0 then
     begin
          Tipo_Simb:=DELIMITADOR;
          simbolo:=exp[t]; {e' um operador}
          t:=t+1;
     end else if Letra(exp[t]) then
     begin
          while(not Delim(exp[t])) do
          begin
```

```
                simbolo:=concat(simbolo,exp[t]); {monta o simbolo}
                t:=t+1;
           end;
           Tipo_Simb:=VARIAVEL;
     end
     else if Digito(exp[t]) then
     begin
           while(not Delim(exp[t])) do
           begin
                simbolo:=concat(simbolo,exp[t]); {monta o numero}
                t:=t+1;
                Tipo_Simb:=NUMERO;
           end;
     end;
end; {Analisa_Simb}

procedure Erro(i:integer); {imprime mensagens de erro}
begin
     case i of
           1: WriteLn('Erro de Sintaxe');
           2: WriteLn('Erro na Colocacao de Parenteses');
           3: WriteLn('Expressao Nao Encontrada');
     end;
end; {Erro}

function Pot(a,b:real):real;
{eleva 'a' a 'b'}
var
   t: integer;
   temp:real;
begin
     if b=0 then Pot:=1 else
     begin
           temp:=a;
           for t:=trunc(b) downto 2 do a:=a*temp;
           Pot:=a;
     end;
end; {Pot}

procedure Arit(op:char; var resultado,operando:real);
{realiza operacoes aritmeticas}
begin
     case op of
           '+':resultado:=resultado+operando;
           '-':resultado:=resultado-operando;
           '*':resultado:=resultado*operando;
           '/':resultado:=resultado/operando;
           '^':resultado:=Pot(resultado,operando);
     end;
end; {Arit}

{Analisador de Expressoes,com variaveis e atribuicao}
procedure Nivel2(var resultado:real);forward;
procedure Nivel3(var resultado:real);forward;
procedure Nivel4(var resultado:real);forward;
procedure Nivel5(var resultado:real);forward;
procedure Nivel6(var resultado:real);forward;
procedure Primitiva(var resultado:real);forward;
```

```
procedure Verifica_Simb(var resultado:real);
begin
     Analisa_Simb;
     if length(simbolo) <> 1 then
        Nivel2(resultado)
     else
        Erro(3);
end; {Verifica_Simb}

procedure Nivel2;
var
   op:char;
   mantem:real;
begin
     Nivel3(resultado);
     op:=simbolo[1];
     while((op='+') or (op='-')) do
     begin
          Analisa_Simb;
          Nivel3(mantem);
          arit(op,resultado,mantem);
          op:=simbolo[1]
     end;
end; {Nivel2}

procedure Nivel3;
var
   op:char;
   mantem:real;
begin
     Nivel4(resultado);
     op:=simbolo[1];
     while ((op='*') or (op='/')) do
     begin
          Analisa_Simb;
          Nivel4(mantem);
          arit(op,resultado,mantem);
          op:=simbolo[1];
     end;
end; {Nivel3}

procedure Nivel4;
var
   mantem:real;
begin
     Nivel5(resultado);
     if simbolo[1]='^' then
     begin
          Analisa_Simb;
          Nivel4(mantem);
          arit('^',resultado,mantem);{exponenciacao}
     end;
end; {Nivel4}

procedure Nivel5;
var
   op:char;
```

```
begin
     op:=' ';
     if ((Tipo_Simb=DELIMITADOR) and ((simbolo[1]='+') or (simbolo[1]='-')))
     then
     begin {sinal negativo ou positivo}
           op:=simbolo[1];
           Analisa_Simb;
     end;
     Nivel6(resultado);
     if op='-' then resultado:=-resultado;
end; {Nivel5}

procedure Nivel6;
begin
     if (simbolo[1]='(') and (Tipo_Simb=DELIMITADOR) then
     begin {subexpressao entre parenteses}
           Analisa_Simb;
           Nivel2(resultado);
           if simbolo[1]<>')' then Erro(2); {erro na colocacao de parenteses}
           Analisa_Simb;
     end
     else Primitiva(resultado);
end; {Nivel6}

procedure Primitiva;
begin
     if Tipo_Simb=NUMERO then
        val(simbolo,resultado,codigo)
     else
         Erro(1);
     Analisa_Simb;
end; {Primitiva}

begin {principal}
      repeat
           t:=1;
           Write('Escreva a expressao: ');
           ReadLn(exp);
           exp:=concat(exp,'$');
           Verifica_Simb(resultado);
           WriteLn(resultado);
      until exp='fim$';
end.
```

Para entender exatamente como o analisador trabalha, usaremos a seguinte expressão (supondo-a contida em **prog**):

10–3*2

Quando **Pega_Exp** (rotina de entrada do analisador) é chamada, ela lê o primeiro símbolo da expressão e, se não houver símbolo algum, a mensagem **falta expressao** é impressa. Se um símbolo estiver presente, então a rotina **Nivel2** é chamada (**Nivel1** será acrescentada posteriormente, quando o operador de atribuição for necessário).

Elemento agora contém o número 10. **Nivel2** chama **Nivel3**, que chama **Nivel4**, que por sua vez chama **Nivel5**. **Nivel5** checa se o símbolo é um "+" ou um "–", o que não é o caso, e chama **Nivel6**. **Nivel6** pode tanto chamar recursivamente **Nivel2** (no caso de uma expressão entre parênteses), como chamar **Primitiva**, para achar o valor do inteiro. Quando **Primitiva** é, finalmente, executada, o valor 10 é colocado em **resultado**, e **Pega_Simb** lê o próximo símbolo. As funções começam a voltar cadeia acima. Como o símbolo agora é um "–", o processo vai até **Nivel2**.

O próximo passo é muito importante. Como o símbolo é um "–", ele é guardado e **Pega_Simb** lê um novo símbolo, 3; a descida da corrente recomeça. **Primitiva** é executada, o inteiro 3 é colocado em **resultado** e um novo símbolo, *, é lido. Isto causa um retorno ao **Nivel3**, onde o último símbolo, 2, é lido. Neste momento, a primeira operação é realizada, com a multiplicação de 2 por 3. O resultado é enviado ao **Nivel2**, onde ocorre a subtração, produzindo 4, o resultado final.

Apesar da aparente complicação, o processo funcionará corretamente com qualquer outra expressão.

Este analisador pode ser usado como uma calculadora simples, num banco de dados ou numa planilha. Mas antes de poder ser usado numa linguagem ou numa calculadora mais sofisticada, o analisador deve ser capaz de manipular variáveis, assunto que veremos a seguir.

A MANIPULAÇÃO DE VARIÁVEIS

Todas as linguagens, além de muitas calculadoras e planilhas eletrônicas usam variáveis para guardar valores que serão necessários em algum momento posterior. Antes que o analisador visto na seção anterior possa ser usado para tais propósitos, ele deve então ser expandido para incluir variáveis. Como o analisador está restrito a números inteiros, as variáveis devem ter valores inteiros. O analisador reconhecerá como variáveis apenas as letras de A a Z (mas isto pode ser facilmente modificado). Cada variável usará um endereço numa matriz de 26 elementos. Assim, a seguinte declaração deve ser acrescentada ao programa:

```
vars : array [0..25] of real;  { 26 variaveis }
```

Antes de serem usadas, estas variáveis devem ser inicializadas em 0.

É necessária, também, uma rotina que verifique o valor de cada variável. Como os nomes das variáveis são as letras de A a Z, a matriz **vars** pode ser indexada com base nestes nomes. Aqui está a função **Acha_Var.**

```
function Acha_Var(S:Str80):real;
var
  t:integer;
begin
  Acha_Var:=vars[Ord(upcase(s[1]))-ord('A')];
end; { Acha_Var }
```

Do modo como está escrita, esta função aceita nomes de variáveis mais longos que apenas uma letra. Porém, só a primeira letra será significativa. Isto pode ser facilmente modificado se necessário.

O procedimento **Primitiva** também deve ser modificado, para admitir tanto números quanto variáveis:

```
procedure Primitiva;
begin
  case Tipo_El of
    NUMERO   : val(Elemento,Resultado,PosErro);
    VARIAVEL : Resultado:=Acha_Var(Elemento);
    else       Erro(1);
    end;
  Pega_Simb;
end; { Primitiva }
```

Neste ponto, o analisador já estaria pronto para lidar corretamente com variáveis. Não há, porém, modo de atribuir valores às variáveis. É sempre possível fazer isto fora do analisador; mas como o "=" pode ser também um operador de atribuição, pode-se tornar a atribuição de valores às variáveis uma parte de analisador. Uma das maneiras de fazer isto é adicionar ao analisador um **Nivel1:**

```
procedure Nivel1;
var
  Guarda:real;
  Temp:T_El;
  Lugar:integer;
  ElemTemp:str80;

begin
  if Tipo_El=VARIAVEL then
  begin
    ElemTemp:=Elemento;
    temp:=Tipo_El;
    Lugar:=Ord(Upcase(Elemento[1]))-ord('A');
    Pega_Simb;
    if Elemento[1]<>'=' then
```

```
    begin
      Devolve;
      Elemento:=ElemTemp;
      Tipo_El:=temp;
      Nivel2(Resultado);
    end else
    begin
      Pega_Simb;
      Nivel2(Resultado);
      vars[Lugar]:=Resultado;
    end;
  end
  else Nivel2(Resultado);
end; { Nivel1 }
```

Quando uma variável for o primeiro símbolo de uma expressão, ela tanto pode ser o alvo de uma atribuição (como em *A*=*B**10) como simplesmente parte da expressão (como em *A*–123.23). Para que **Nivel 1** possa dar seu próximo passo, ele deve dar uma *olhada à frente*. Olhar à frente significa guardar o símbolo atual e obter o próximo para checar sua natureza. Assim, se o símbolo for "=", fica-se sabendo que uma atribuição está em curso, e as rotinas apropriadas são executadas. Se o símbolo não for um "=", ele deve, então, ser devolvido à expressão, e o símbolo anterior deve ser recuperado. Isto é feito pelo procedimento **Devolve**, cuja única função é subtrair 1 do índice **t**. Olhar à frente é um processo que consome tempo, só devendo ser realizado se for absolutamente necessário.

Aqui está, então, o analisador completo, com todas as rotinas de apoio:

```
program analisador2; {Analisador com variaveis}
type
    str80=string[80];
    Nome_Simb=(DELIMITADOR, VARIAVEL, NUMERO);
var
   simbolo,exp:str80;
   Tipo_Simb:Nome_Simb;
   codigo,t:integer;
   resultado:real;
   vars:array[0..25] of real;  {26 variaveis}

function Letra(car:char):boolean;
{TRUE se car e' uma letra do alfabeto}

begin
     Letra:=(UpCase(car)>='A') and (UpCase(car)<='Z');
end;   {Letra}

function Branco(car:char):boolean;
{TRUE se return,espaco ou tab}

begin
     Branco:=(car=' ') or (car=chr(9)) or (car=chr(13));
end;   {Branco}
```

```
function Delim(car:char):boolean;
{TRUE se for um DELIMITADOR}

begin
     if pos(car, '+-/*%^=()$')<>0 then Delim:=TRUE
     else Delim:=FALSE;
end; {Delim}

function Digito(car:char) :boolean;
{TRUE se for um digito entre 0 e 9}

begin
     Digito:=(car>='0') and (car<='9');
end;   {Digito}

procedure Analisa_Simb;
var
   temp:str30;

begin
     simbolo:=' '; {sequencia de caracteres nula}
     while(Branco(exp[t])) do t:=t+1; {retira os espacos antes da expressao}
     if exp[t]='$' then simbolo:='$';
     if pos(exp[t],'+-*/^=()')<>0 then
     begin
          Tipo_Simb:=DELIMITADOR;
          simbolo:=exp[t]; {e' um operador}
          t:=t+1;
     end else if Letra(exp[t]) then
     begin
          while(not Delim(exp[t])) do
          begin
               simbolo:=concat(simbolo,exp[t]); {monta o simbolo}
               t:=t+1;
          end;
          Tipo_Simb:=VARIAVEL;
     end
     else if Digito(exp[t]) then
     begin
          while(not Delim(exp[t])) do
          begin
               simbolo:=concat(simbolo,exp[t]); {monta o numero}
               t:=t+1;
               Tipo_Simb:=NUMERO;
          end;
     end;
end; {Analisa_Simb}


procedure Devolve; {devolve os simbolos nao usados}
begin
     t:=t-length(simbolo);
end; {Devolve}

procedure Erro(i:integer); {imprime mensagens de erro}
begin
     case i of
          1: WriteLn('Erro de Sintaxe');
          2: WriteLn('Erro na Colocacao de Parenteses');
```

```
          3: WriteLn('Expressao Nao Encontrada');
     end;
end; {Erro}

function Pot(a,b:real):real;
{eleva 'a' a 'b'}
var
   t: integer;
   temp:real;
begin
     if b=0 then Pot:=1 else
     begin
          temp:=a;
          for t:=trunc(b) downto 2 do a:=a*temp;
          Pot:=a;
     end;
end; {Pot}

function AchaVar(s:str90):real;
var
   t:integer;
begin
     AchaVar:=vars[Ord(Upcase(s[1]))-Ord('A')];
end;  {AchaVar}

procedure Arit(op:char; var resultado,operando:real);
{realiza operacoes aritmeticas}
begin
     case op of
          '+':resultado:=resultado+operando;
          '-':resultado:=resultado-operando;
          '*':resultado:=resultado*operando;
          '/':resultado:=resultado/operando;
          '^':resultado:=Pot(resultado,operando);
     end;
end; {Arit}

{Analisador de Expressoes,com variaveis e atribuicao}
procedure Nivel1(var resultado:real);forward;
procedure Nivel2(var resultado:real);forward;
procedure Nivel3(var resultado:real);forward;
procedure Nivel4(var resultado:real);forward;
procedure Nivel5(var resultado:real);forward;
procedure Nivel6(var resultado:real);forward;
procedure Primitiva(var resultado:real);forward;


procedure Verifica_Simb(var resultado:real);
begin
     Analisa_Simb;
     if length(simbolo) <> 1 then
        Nivel1(resultado)
     else
        Erro(3);
end; {Verifica_Simb}

procedure Nivel1;
var
```

```
   mantem:real;
   temp:Nome_Simb;
   posicao:integer;
   Simb_Temp:str80;

begin
     if Tipo_Simb=VARIAVEL then
     begin
          {guarda o simbolo anterior}
          Simb_Temp:=simbolo;
          temp:=Tipo_Simb;
          posicao:=Ord(UpCase(simbolo[1]))-Ord('A');
          Analisa_Simb; {verifica se ha = para atribuicao}
          if simbolo[1]<>'=' then
          begin
               Devolve; {devolve o simbolo}
               {repoe simbolo anterior}
               simbolo:=Simb_Temp;
               Tipo_Simb:=temp;
               Nivel2(resultado);
          end else
          begin {ha atribuicao}
                Analisa_Simb;
                Nivel2(resultado);
                vars[posicao]:=resultado;
          end;
     end {if}
     else Nivel2(resultado);
end; {Nivel1}

procedure Nivel2;
var
   op:char;
   mantem:real;
begin
     Nivel3(resultado);
     op:=simbolo[1];
     while((op='+') or (op='-')) do
     begin
          Analisa_Simb;
          Nivel3(mantem);
          arit(op,resultado,mantem);
          op:=simbolo[1]
     end;
end; {Nivel2}

procedure Nivel3;
var
   op:char;
   mantem:real;
begin
     Nivel4(resultado);
     op:=simbolo[1];
     while ((op='*') or (op='/')) do
     begin
          Analisa_Simb;
          Nivel4(mantem);
          arit(op,resultado,mantem);
          op:=simbolo[1];
```

```
     end;
end; {Nivel3}

procedure Nivel4;
var
   mantem:real;
begin
     Nivel5(resultado);
     if simbolo[1]='^' then
     begin
          Analisa_Simb;
          Nivel4(mantem);
          arit('^',resultado,mantem);{exponenciacao}
     end;
end; {Nivel4}

procedure Nivel5;
var
   op:char;
begin
     op:=' ';
     if ((Tipo_Simb=DELIMITADOR) and ((simbolo[1]='+') or (simbolo[1]='-')))
     then
     begin {sinal negativo ou positivo}
           op:=simbolo[1];
           Analisa_Simb;
     end;
     Nivel6(resultado);
     if op='-' then resultado:=-resultado;
end; {Nivel5}

procedure Nivel6;
begin
     if (simbolo[1]='(') and (Tipo_Simb=DELIMITADOR) then
     begin {subexpressao entre parenteses}
           Analisa_Simb;
           Nivel2(resultado);
           if simbolo[1]<>')' then Erro(2); {erro na colocacao de parenteses}
           Analisa_Simb;
     end
     else Primitiva(resultado);
end; {Nivel6}

procedure Primitiva;
begin
     if Tipo_Simb=NUMERO then
        val(simbolo,resultado,codigo)
     else if Tipo_Simb=VARIAVEL then
          resultado:=AchaVar(simbolo)
     else
          Erro(1);
     Analisa_Simb;
end; {Primitiva}

begin {principal}
      for t:=0 to 25 do vars[t]:=0; {inicializa variaveis}
      repeat
            t:=1;
            Write('Escreva a expressao: ');
```

```
            ReadLn(exp);
            exp:=concat(exp,'$');
            Verifica_Simb(resultado);
            WriteLn(resultado);
        until exp='fim$';
end.
```

Como analisador completo, agora você pode introduzir expressões como

$A=10/4$
$A-B$
$C=A*(F-21)$

e elas serão avaliadas apropriadamente.

A SINTAXE NUM ANALISADOR RECURSIVO DECRESCENTE

Na análise de expressões, um erro de sintaxe é uma situação em que a expressão inicializada não se adapta às regras do analisador. Muitas vezes, o erro de sintaxe é um erro humano, geralmente um erro de digitação. Por exemplo, as seguintes expressões não serão analisadas corretamente pelos analisadores vistos neste capítulo:

```
10**8
(10–5)*9)
/8
```

A primeira expressão tem dois operadores em seqüência, a segunda um parênteses a mais e a terceira um sinal de divisão no início da expressão. Como erros de sintaxe podem confundir o analisador e gerar resultados errôneos, é importante e necessário resguardar-se contra eles.

No analisador visto, o procedimento **Erro** é chamado em certas situações. Ao contrário do que ocorre em vários outros analisadores, o método recursivo decrescente torna a verificação de erros de sintaxe muito fácil, pois eles em geral ocorrem em **Primitiva, Acha_Var** ou **Nivel6,** onde os parênteses são checados. A verificação de erros, do modo como está, tem apenas um inconveniente: o analisador não pára ao encontrar um erro. Isto pode causar a impressão de múltiplas mensagens de erro na tela.

Para incluir uma recuperação completa do erro, você deve acrescentar ao programa uma variável global que seja checada a cada nível. A variável seria inicialmente **FALSE**, tornando-se **TRUE** quando **Erro** fosse chamado, forçando o analisador a abandonar uma função por vez.

O único problema do programa ser mantido como está seria a geração de múltiplas mensagens de erro. Isto pode até ser útil, pois todos os erros de uma expressão serão captados ao mesmo tempo. Mas você, certamente, deve expandir a verificação de erros antes de usar o analisador em programas comerciais.

CAPÍTULO **10**

O TURBO PASCAL DATABASE TOOLBOX

Uma poderosa extensão, disponível para ser usada junto com o Turbo Pascal é o Turbo Pascal Database Toolbox. Este pacote contém rotinas que permitem operações com bancos de dados em *B-tree*, ordenação de arquivos de dados e instalação de programas no terminal do usuário final. Estas rotinas são o **Turbo Access**, o **TurboSort** e o **GINST**, respectivamente. Este capítulo examina cada uma delas, dando especial ênfase às rotinas de banco de dados.

O TURBO ACCESS

As rotinas do **Turbo Acess** criam uma estrutura de arquivos em *B-tree*. A *B-tree*, inventada por R. Bayer, difere da árvore binária comum por permitir que cada nó-raiz possua mais de dois ramos, como mostra a Figura 10-1. A organização da *B-tree* possibilita um rápido acesso a arquivos de disco. A implementação de *B-trees* é uma tarefa bastante árdua, mas você não precisa saber exatamente como elas funcionam para utilizar as rotinas do Toolbox – a Borland já fez todo o trabalho pesado para você.

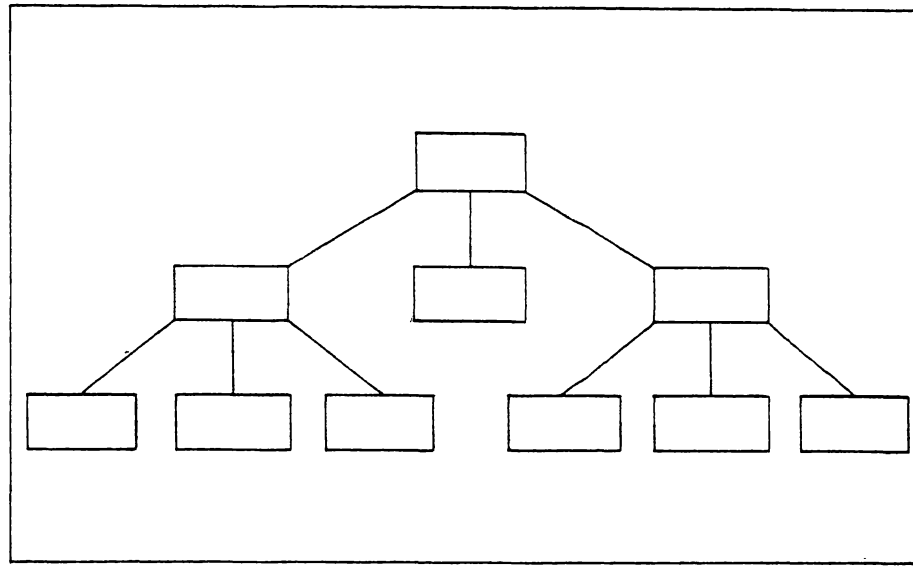


Figura 10-1 Um exemplo de *B-tree*.

ARQUIVOS NECESSÁRIOS PARA AS ROTINAS DO TURBO ACCESS

Para que seja possível usar as rotinas do Turbo Access, os seguintes arquivos devem estar presentes:

Arquivo	Função
ACCESS.BOX	Inicialização e manutenção básica de arquivos de dados e índices.
ADDKEY.BOX	Acrescenta chaves e arquivos de índice.
DELKEY.BOX	Apaga chaves de arquivos de índice.
GETKEY.BOX	Encontra chaves.

Note que o Database Toolbox vem com duas versões das rotinas básicas de inicialização e manutenção de banco de dados, uma para a versão 2 do Turbo Pascal e outra para a versão 3. Elas se chamam **ACCESS2.BOX** e **ACCESS3.BOX**, respectivamente. Você deve criar, com o arquivo apropriado, um terceiro arquivo chamado **ACCESS.BOX**, de modo a possuir, neste arquivo, as rotinas corretas para sua versão do Turbo Pascal. Daqui em diante, este capítulo pressupõe que as rotinas básicas de banco de dados são encontradas no arquivo **ACCESS.BOX.**

Além destes arquivos, você precisará do programa **SETCONST.PAS**, também fornecido pelo Toolbox, para calcular algumas constantes necessárias às rotinas de *B-tree*.

ARQUIVOS DE DADOS E ARQUIVOS-ÍNDICES

No sistema Turbo Access, um arquivo de dados (*data file*) é onde são armazenados os registros que contêm a informação que você deseja manipular. Cada arquivo criado via Turbo Access pode conter até 65.536 registros. Entretanto, o primeiro registro é reservado para uso do Turbo Access, reduzindo o número máximo de registros disponíveis a 65.535. Cada registro pode conter até 65.535 bytes, mas não se recomenda o uso de registros tão grandes. O tamanho mínimo de um registro é 8 bytes. As variáveis que representam os arquivos de dados no programa devem ser declaradas do tipo **DataFile**, definido por **ACCESS.BOX.**

Um arquivo-índice (*index file*) serve para armazenar uma chave (*key*) que identifica um registro de um arquivo de dados, ao lado do número deste registro no arquivo de dados. Cada arquivo-índice pode conter até 65.355 chaves. O arquivo-índice permite uma rápida localização de qualquer registro no arquivo de dados associado. É importante saber que no Turbo Access tôdas as chaves têm de ser seqüências de caracteres (*strings*). As variáveis que representam os arquivos-índices no programa devem ser declaradas do tipo **IndexFile.**

AS CONSTANTES DO MÉTODO B-TREE

Seis constantes especiais devem ser declaradas em qualquer programa que use o sistema Turbo Access. Estas constantes especificam parâmetros como altura da árvore e número de ramos ligados a cada nó. Estas constantes são:

- **MaxDataRecSize**
- **MaxKeyLen**
- **PageSize**
- **Order**
- **PageStackSize**
- **MaxHeight**

Os valores atribuídos a estas constantes são calculados idealmente pelo programa **SETCONST.PAS**. Para usá-lo, você deve fornecer o tamanho do registro que

estará usando, bem como o tamanho da chave. Usaremos o **SETCONST.PAS** mais adiante neste capítulo.

NOMES DE VARIÁVEIS RESERVADOS E CÓDIGOS DE ERRO

Turbo Access usa variáveis globais que começam com as letras "TA". Você pode declarar variáveis cujos nomes começam com "TA", mas evite fazê-lo; isto pode ocasionar erros de compilação por duplicação de definições. Se um erro de E/S acontecer quando você estiver usando rotinas do **Turbo Access**, um dos códigos de erro da Tabela 10-1 deve aparecer.

Tabela 10-1 Códigos de erro de E/S do Turbo Access

Código	Significado
1	Registro de comprimento não previsto.
2	Arquivo inexistente.
3	Diretório lotado.
4	Arquivo não está aberto.
5	Arquivo não está aberto para entrada (*input*).
6	Arquivo não está aberto para saída (*output*).
7	EOF prematuro.
8	Gravação mal sucedida.
9	Formato numérico errado.
10	EOF prematuro.
11	Tamanho do arquivo excede a 65.535 registros.
12	EOF prematuro.
13	Gravação mal sucedida.
14	Tentativa de busca (*seek*) além do EOF.
15	Arquivo não foi aberto.
16	Operação ilegal.
17	Ilegal em modo direto.
18	Uso ilegal de *assign* com arquivo predefinido.
144	Registro de comprimento não previsto.
145	Tentativa de busca (*seek*) além do EOF.
153	EOF prematuro.
240	Gravação mal sucedida.
243	Excesso de arquivos abertos.

ROTINAS DO TURBO ACCESS

Os vinte procedimentos e funções que formam o sistema **Turbo Access** estão resumidos na Tabela 10-2. Nesta seção, discutiremos brevemente cada uma delas. Ao usar estas rotinas, você precisará também de uma variável global de nome **OK**, definida por **ACCESS.BOX**, para determinar o sucesso ou insucesso de várias destas rotinas.

Tabela 10-2 Resumo dos procedimentos e funções do **Turbo Access**

Nome	Uso
procedure **Addkey**	Acrescenta uma chave a um arquivo-índice.
procedure **AddRec**	Acrescenta um registro a um arquivo de dados.
procedure **ClearKey**	Posiciona o indicador de índice para o topo do arquivo-índice.
procedure **CloseFile**	Fecha um arquivo de dados.
procedure **CloseIndex**	Fecha um arquivo-índice.
procedure **DeleteKey**	Retira uma chave de um arquivo-índice.
procedure **DeleteRec**	Retira um registro de um arquivo de dados.
function **FileLen**	Retorna o número de registros de um arquivo de dados.
procedure **FindKey**	Retorna o número do registro associado a uma chave dada.
procedure **GetRec**	Lê um registro especificado.
procedure **InitIndex**	Inicializa variávies-índices do sistema
procedure **MakeFile**	Cria um novo arquivo de dados.
procedure **MakeIndex**	Cria um novo arquivo-índice.
procedure **NextKey**	Retorna o número do registro associado à próxima chave do índice.
procedure **OpenFile**	Abre um arquivo de dados previamente criado.
procedure **OpenIndex**	Abre um arquivo-índice previamente criado.
procedure **PrevKey**	Retorna o número do registro associado à chave anterior do índice
procedure **PutRec**	Escreve um registro num arquivo de dados.
procedure **SearchKey**	Retorna o número do registro associado a uma chave dada, se houver; ou o número-registro associado à primeira chave maior que a chave dada.
function **UsedRecs**	Retorna o número de registros que contém informação válida num arquivo de dados.

AddKey

O procedimento **AddKey** é declarado assim:

```
procedure Addkey(var Arq_ind: IndexFile; var NumReg:integer; var Chave);
```

Addkey é usado para acrescentar uma chave e um número de registro a um arquivo-índice. O parâmetro **NumReg** especifica o número do registro associado à chave no arquivo de dados. O parâmetro **Chave** não tem tipo, mas deve ser uma *string* do tamanho apropriado. Se a operação é bem-sucedida, **OK** recebe o valor **TRUE;** do contrário, recebe **FALSE.**

AddRec

O procedimento **AddRec** acrescenta um novo registro a um arquivo de dados. Sua declaração é:

```
procedure AddRec(var Arq_dad: DataFile; var NumReg: integer; var Buffer);
```

AddRec acrescenta o registro contido em **Buffer** ao arquivo de dados **Arq_dad** e coloca o número do registro acrescentado em **NumReg.** O valor retornado em **NumReg** geralmente é usado para chamar **AddKey** em seguida. A variável **Buffer** não tem tipo, mas deve conter um registro válido. Em caso de sucesso, **OK** é **TRUE**; senão, **OK** é **FALSE.**

ClearKey

O procedimento **ClearKey** é usado para fazer o indicador de um arquivo-índice apontar para o início do arquivo. É declarado assim:

```
procedure ClearKey(var Arq_ind: IndexFile);
```

CloseFile e CloseIndex

Os procedimentos **CloseFile** e **CloseIndex** são usados para fechar arquivos de dados e índices, respectivamente. Suas declarações são:

```
procedure CloseFile(var Arq_dad: DataFile);
procedure CloseIndex(var Arq_ind: IndexFile);
```

Você deve fechar qualquer arquivo aberto pelo Turbo Access. Caso contrário, poderá perder registros de dados e destruir índices.

DeleteKey

O procedimento **DeleteKey** retira uma chave de um arquivo-índice. É declarado assim:

```
procedure DeleteKey(var Arq_ind: IndexFile; var NumReg: integer; var Chave);
```

Se houver chaves duplicadas no arquivo-índice, **NumReg** deve conter o número do registro associado à chave para permitir a identificação correta da chave a ser removida. **OK** é **TRUE** em caso de sucesso; **FALSE**, em caso contrário.

DeleteRec

O procedimento **DeleteRec** apaga um registro específico de um arquivo de dados. Sua declaração é:

```
procedure DeleteRec(var Arq_dad: DataFile; var NumReg: integer);
```

DeleteRec retira o registro identificado pelo número de registro **NumReg.** Se a operação é bem-sucedida, **OK** é **TRUE**; do contrário, **OK** é **FALSE.** Este procedimento remove logicamente o registro do arquivo, mas ele continua existindo fisicamente e é colocado numa lista encadeada de registros removidos, para um eventual uso futuro. É importante não tentar remover um registro já removido, pois você pode destruir a lista de registros removidos.

FileLen

A função **FileLen** retorna o número de registros de um arquivo de dados. É declarada assim:

```
function FileLen(var Arq_dad: DataFile): integer;
```

Lembre-se de que os arquivos de dados do Turbo Access têm informações do sistema no primeiro registro. Por isso, o número de registros que contêm seus dados num determinado arquivo é **FileLen – 1.**

FindKey

O procedimento **FindKey** localiza a chave solicitada num arquivo-índice e retorna o número de registro associado a ela. É declarada assim:

```
procedure FindKey(var Arq_ind: IndexFile; var NumReg: integer; var Chave);
```

Se **Chave** está no arquivo **Arq_ind**, então o número de registro associado a **Chave** é colocado em **NumReg** e **OK** recebe o valor **TRUE**. Do contrário, **OK** é **FALSE.**

GetRec

O procedimento **GetRec** lê o registro de número **NumReg** do arquivo de dados **Arq_dad,** de acordo com a declaração:

```
procedure GetRec(var Arq_dad: DataFile; var NumReg: integer; var Buffer);
```

A informação lida é armazenada na variável **Buffer,** que não tem tipo mas deve ter o tamanho apropriado.

InitIndex

InitIndex serve para inicializar as tabelas usadas pelo Turbo Access, para organizar os arquivos-índices. Não usa parâmetros. **InitIndex** deve ser executado uma vez no início do seu programa, antes de qualquer chamada às demais rotinas do Turbo Access.

MakeFile e MakeIndex

Os procedimentos **MakeFile** e **MakeIndex** servem para criar arquivos de dados e índices, para uso com o Turbo Access. Suas declarações são:

```
procedure MakeFile(var Arq_dad: Data File; NomeArq: string[14];
                   TamReg: integer);
procedure MakeIndex(var Arq_ind: IndexFile; NomeArq: string[14]; TamCha,
                    Dupli: integer);
```

O parâmetro **TamReg** é o comprimento em bytes dos registros a serem armazenados no arquivo **Arq_dad.** O melhor meio de obter este número é usar a função **SizeOf** do Turbo Pascal, em vez de contar os bytes manualmente. O parâmetro **TamCha** em **MakeIndex** é o tamanho da chave em bytes. **Dupli** deve ser 0 (zero) ou 1 (um). Se **Dupli** for 1 (um), então o índice poderá conter chaves duplicadas; caso contrário, cada chave aparecerá só uma vez no índice. Ambos os procedimentos colocam **TRUE** na variável **OK** se forem bem-sucedidos, **FALSE** se não forem.

NextKey

NextKey retorna o número de registro associado à chave seguinte do arquivo-índice. É declarado assim:

```
procedure NextKey(Arq_ind: IndexFile; var NumReg: integer; var Chave);
```

Após a execução de **NextKey,** a variável **Chave** conterá a próxima chave encontrada no arquivo-índice **Arq_ind**, e **NumReg** conterá o número do registro associado a esta chave. Se a operação foi bem-sucedida, **OK** é **TRUE;** senão **OK** é **FALSE.**

OpenFile e OpenIndex

Os procedimentos **OpenFile** e **OpenIndex** são usados para abrir arquivos de dados e índices já existentes. Eles são declarados assim:

```
procedure OpenFile(var Arq_dad: DataFile; NomeArq: string[14];
                   TamReg: integer);
```

```
procedure OpenIndex(var Arq_ind: IndexFile; NomeArq: string[14];
                    TamCha, Dupli: integer);
```

O parâmetro **TamReg** é o comprimento em bytes dos registros armazenados no arquivo **Arq_dad**. O parâmetro **TamCha** em **OpenIndex** é o tamanho da chave em bytes. **Dupli** deve ser 0 (zero) ou 1 (um). Se **Dupli** for 1 (um), então o índice poderá conter chaves duplicadas; caso contrário, cada chave aparecerá só uma vez no índice. Ambos os procedimentos colocam **TRUE** na variável **OK** se forem bem-sucedidos, **FALSE** se não forem.

PrevKey

PrevKey retorna o número de registro associado à chave anterior do arquivo-índice. É declarado assim:

```
procedure PrevKey(Arq_ind: IndexFile; var NumReg: integer; var Chave);
```

Após a execução de **PrevKey**, a variável **Chave** conterá a chave anterior à última encontrada no arquivo-índice **Arq_ind**, e **NumReg** conterá o número do registro associado a esta chave. Se a operação foi bem-sucedida, **OK** é **TRUE**; senão **OK** é **FALSE**.

PutRec

O procedimento **PutRec** grava um registro de dados na posição especificada do arquivo de dados. Sua declaração é:

```
procedure PutRec(var Arq_dad: DataFile; var NumReg: integer; var Buffer);
```

A informação contida em **Buffer** é gravada no registro de número **NumReg** do arquivo de dados **Arq_dad**. A variável **Buffer** não tem tipo, mas deve conter um registro do tamanho apropriado.

SearchKey

O procedimento **SearchKey** é usado para localizar, num arquivo índice, a primeira chave igual ou maior à chave especificada. A declaração é:

procedure SearchKey(var Arq_ind: IndexFile; var NumReg: integer; var Chave);

Se há uma chave igual ou maior que o parâmetro **Chave** no arquivo **Arq_ind**, então o número de registro associado à chave localizada é colocado em **NumReg** e **OK** recebe o valor **TRUE**. Do contrário, **OK** é **FALSE.**

UsedRecs

UsedRecs retorna o número de registros contendo informação válida num arquivo de dados. Registros removidos com **DeleteRec** não são contados. A declaração de **UsedRecs** é:

function UsedRecs(var Arq_dad: DataFile) : integer;

EXEMPLO: UMA MALA DIRETA SIMPLES

Como exemplo, o programa de mala direta simples, desenvolvido no Capítulo 3, através de lista encadeada, será adaptado para usar as rotinas do Turbo Access. Primeiro você deve calcular o tamanho do registro que contém o endereço. Depois, use o programa **SETCONST** para calcular valores adequados para as seis constantes usadas pelo Turbo Access.

O Turbo Access usa os dois primeiros bytes de cada registro para indicar a remoção lógica do registro. Por isso, é necessário acrescentar um campo do tipo **integer** no início da definição do registro original. Além disso, os campos **anterior** e **proximo** não são mais necessários. O registro revisto fica assim:

```
type
   endereco = record
            status: integer;    {usado pelo Turbo Access}
            nome: string[30];
            rua: string[40];
            cidade: string[20];
            estado: string[2];
            cep: string[5];
   end;
```

O comprimento em bytes de **endereco,** achado através da função **SizeOf,** é 104. Você terá de fornecer este número para o programa SETCONST.PAS.

O programa SETCONST.PAS determina o valor das seis constantes usadas pelas rotinas do Turbo Access. Ao executá-lo, você verá uma tela como a da Figura 10-2, com os valores default.

```
==        Turbo Access constant determination worksheet, Version 1.10A        ==

Data record size (bytes)                              200       200
Key string length (characters)                         10
Size of the database (records)                       10000
Page size (keys)                                        24
Page stack size (pages)                                 10

Density (Percent of Items in use per average Page)   50%          75%          100%

Total index file pages
Memory used for page stack (bytes)
Index file page size (bytes)
Index file size (bytes)
Data file size (bytes)

Order
MaxHeight
Average searches needed to find a key
Average searches satisfied by page stack
Average disk searches needed to find a key

ESC to end program
```

Figura 10-2 Tela inicial do programa SETCONST.PAS.

O Manual do Turbo Database Toolbox afirma que, na maioria das aplicações, você só precisa alterar os valores do tamanho do registro de dados (*Data record size*) e do comprimento da chave (*Key string length*). O número de registros por arquivo (*Size of the database*) também deve ser alterado se você for trabalhar com mais de 10.000 registros. Os parâmetros *Page size* e *Page stack size* não precisam ser alterados. O tamanho do registro de dados neste exemplo é 104, portanto, este valor deve ser digitado no espaço *Data record size*. Ainda no nosso exemplo, usaremos o campo **nome** como chave, e seu tamanho (30) deve ser digitado no espaço *Key string length*. Pressione RETURN para confirmar os demais valores. Uma vez que isto for feito, o programa apresentará a tela de acordo com a Figura 10-3.

```
==       Turbo Access constant determination worksheet, Version 1.10A       ==

Data record size (bytes)                          108
Key string length (characters)                    30
Size of the database (records)                  10000
Page size (keys)                                   24
Page stack size (pages)                            10

Density (Percent of Items in use per average Page)  50%        75%        100%

Total index file pages                             834        556        417
Memory used for page stack (bytes                 8430       8430       8430
Index file page size (bytes)                       843        843        843
Index file size (bytes)                         703062     468708     351531
Data file size (bytes)                         1080108    1080108    1080108

Order                                              12         12         12
MaxHeight                                           4          4          3
Average searches needed to find a key            3.71       3.19       2.90
Average searches satisfied by page stack         1.75       1.50       1.38
Average disk searches needed to find a key       1.96       1.69       1.52

ESC to end program
```

Figura 10-3 A tela de SETCONST.PAS com os novos valores.

Ao sair de SETCONST.PAS você pode criar automaticamente as declarações de constantes. As declarações do nosso exemplo aparecem a seguir, com comentários colocados pelo autor.

```
Const
   (Estas constantes sao geradas pelo programa
    SETCONST.PAS, fornecido pelo Database Toolbox.)
   MaxDataRecSize = 108;
   MaxKeyLen      =  30;
   PageSize       =  24;
   Order          =  12;
   PageStackSize  =  10;
   MaxHeight      =   4;
```

Dadas estas informações e com os arquivos requeridos pelo Turbo Access já incluídos, a primeira parte do programa de mala direta fica assim:

```
program Exemplo_BD;

Const
   (Estas constantes sao geradas pelo programa
    SETCONST.PAS, parte integrante do DataBase Toolbox.)
   MaxDataRecSize = 108;
   MaxKeyLen      =  30;
   PageSize       =  24;
   Order          =  12;
   PageStackSize  =  10;
   MaxHeight      =   4;
```

```
type
   endereco = record
            status: integer;   {usado pelo Turbo Access}
            nome: string[30];
            rua: string[40];
            cidade: string[20];
            estado: string[2];
            cep: string[5];
   end;
{estes arquivos contem as rotinas de banco de dados}
{$I access.box}  {rotinas basicas}
{$I addkey.box}  {inserir dados}
{$I delkey.box}  {apagar dados}
{$I getkey.box}  {busca na arvore}
var
   arq_dad: DataFile;
   arq_ind: Indexfile;
   feito: boolean;
```

A parte principal do programa, listada a seguir, primeiro inicializa a tabela de índices, usando **InitIndex.** Então, ela abre ou cria o arquivo de dados e o índice apropriado. O *loop* principal do programa é semelhante àquele desenvolvido no Capítulo 3, permitindo ao usuário selecionar diversas opções. Ao final, o arquivo de dados e o índice são fechados.

```
begin
   InitIndex;
   OpenFile(arq_dad, 'mala.dir', SizeOf(endereco));
   if not OK then
   begin
      WriteLn('criando arquivo de indice');
      MakeIndex(arq_ind, 'mala.ind',30,0);
   end;
   feito:=FALSE;
   repeat
      case Menu of
         '1': Digitacao;
         '2': Remove;
         '3': Lista;
         '4': Busca;
         '5': Atualizacao;
         '6': feito:=TRUE;
      end;
   until feito;
   CloseFile(arq_dad);
   CloseIndex(arq_ind);
end.
```

Note que não é mais necessário ler ou gravar a mala direta explicitamente – as rotinas do Turbo Access mantêm o arquivo atualizado automaticamente.

O procedimento **Digitacao,** mostrado a seguir, recebe a informação de um endereço, grava-a no arquivo de dados, e coloca a respectiva chave no arquivo-índice. Note, também, que o campo **status** de cada registro armazena o valor 0. Por convenção, o valor 0 no campo **status** indica registro ativo; valores diferentes de 0 indicam registros removidos logicamente.

```
{entrar dados}
procedure Digitacao;
var
   feito: boolean;
   numreg: integer;
   temp: string[30];
   item: endereco;
begin
   feito:=FALSE;
   repeat
      Write('Nome: ');
      Read(item.nome); WriteLn;
      if Length(item.nome)=0 then feito:=TRUE
      else
      begin
         Write('Rua: ');
         Read(item.rua); WriteLn;
         Write('Cidade: ');
         Read(item.cidade); WriteLn;
         Write('Estado: ');
         Read(item.estado); WriteLn;
         Write('CEP: ');
         Read(item.cep); WriteLn;
         item.status:=0;  {marca como ativo}
         FindKey(arq_ind,numreg,item.nome);
         if not OK then  {verifica se nao existem
                          chaves iguais}
         begin
            AddRec(arq_dad,numreg,item);
            AddKey(arq_ind,numreg,item.nome)
         end else WriteLn('Chaves iguais ignoradas');
      end;
   until feito;
end;  {Digitacao}
```

Como você pode ver, este procedimento verifica se não existem chaves em duplicata. Pelo fato de nomes em duplicata não serem permitidos, primeiramente **Enter** checa para ver se a nova chave se iguala a uma que já esteja no arquivo. Se isto ocorrer, a entrada é ignorada.

O procedimento **ListAll** lista todo o conteúdo do *mailing list:*

```
procedure Lista;
var
   item: endereco;
   len,numreg: integer;
begin
   len:=filelen(arq_dad)-1;
   for numreg:=1 to len do
   begin
      GetRec(arq_dad,numreg,item);
      {mostra, se houver}
      if item.status = 0 then mostra(item);
   end;
end;
```

FileLen retorna o número de registros – ativos ou apagados – do arquivo de dados, incluindo o primeiro registro que é reservado para uso do Turbo Access. Portanto, o número de registros de usuário é **FileLen** – 1. Além disso, como alguns registros podem estar apagados, é preciso checar o campo **status** antes de confiar nesta informação.

A busca de um endereço específico envolve, em primeiro lugar, a localização da chave no índice através de **FindKey**. Quando a chave é localizada, o número do registro de dados associado é retornado e usado com **GetRec** para buscar a informação. O procedimento **Busca**, mostrado a seguir, implementa esta técnica:

```
{acha um dado especifico}
procedure Busca;
var
   nome: string[30];
   item: endereco;
   numreg: integer;
begin
   Write('Nome: ');
   ReadLn(nome);

   {acha a chave, se existir}
   FindKey(arq_ind,numreg,nome);
   if OK then  {se houver uma chave}
   begin
      GetRec(arq_dad,numreg,item);
      {mostra,se houver}
      if item.status = 0 then Mostra(item);
   end else WriteLn('nao encontrado');
end;  {Busca}
```

Finalmente, atualizar um registro existente implica primeiro encontrar o registro, lê-lo, modificá-lo e escrevê-lo novamente no arquivo de dados. O procedimento **Update** ilustra um método simples de atualização, no qual o usuário precisa reinserir toda informação. Um acesso mais sofisticado seria reinserir apenas os arquivos modificados.

```
{Altera o conteudo de um endereco na lista, exceto o nome}
procedure Atualizacao;
var
   feito: boolean;
   numreg: integer;
   temp: string[30];
   item: endereco;
begin
   Write('Nome: ');
   Read(item.nome); WriteLn;
   FindKey(arq_ind,numreg,item.nome);
   if OK then
   begin
      Write('Rua: ');
      Read(item.rua); WriteLn;
      Write('Cidade: ');
      Read(item.cidade); WriteLn;
      Write('Estado: ');
      Read(item.estado); WriteLn;
      Write('CEP: ');
      Read(item.cep); WriteLn;
      item.status:=0;  {marca como ativo}
      PutRec(arq_dad,numreg,item);
   end else WriteLn('Key not found');
end;  {Atualizacao}
```

O programa de mala direta completo, usando o **Turbo Access** para manutenção dos arquivos, é mostrado a seguir:

```
program Exemplo_BD;

Const
   {Estas constantes sao geradas pelo programa
    SETCONST.PAS, parte integrante do DataBase Toolbox.}
   MaxDataRecSize = 108;
   MaxKeyLen      =  30;
   PageSize       =  24;
   Order          =  12;
   PageStackSize  =  10;
   MaxHeight      =   4;

type
   endereco = record
            status: integer;   {used by Turbo Access}
            nome: string[30];
            rua: string[40];
            cidade: string[20];
            estado: string[2];
            cep: string[5];
   end;

{estes arquivos contem as rotinas de banco de dados}
{$I access.box}  {rotinas basicas}
{$I addkey.box}  {inserir dados}
{$I delkey.box}  {apagar dados}
{$I getkey.box}  {busca na arvore}
```

```
var
   arq_dad: DataFile;
   arq_ind: Indexfile;
   feito: boolean;

function Menu:char;  {retorna opcao do usuario}

var
   car: char;

begin
   WriteLn('1. Digitar nomes');
   WriteLn('2. Apagar um nome');
   WriteLn('3. Ver lista');
   WriteLn('4. Procurar um nome');
   WriteLn('5. Atualizar lista');
   WriteLn('6. Fim');
   repeat
      WriteLn;
      Write('Digite a sua opcao: ');
      Read(car); car:=UpCase(car); WriteLn;
   until (car >= '1') and (car <= '6');
   Menu:=car;
end;  {Menu}

{entrar dados}
procedure Digitacao;
var
   feito: boolean;
   numreg: integer;
   temp: string[30];
   item: endereco;
begin
   feito:=FALSE;
   repeat
      Write('Nome: ');
      Read(item.nome); WriteLn;
      if Length(item.nome)=0 then feito:=TRUE
      else
      begin
         Write('Rua: ');
         Read(item.rua); WriteLn;
         Write('Cidade: ');
         Read(item.cidade); WriteLn;
         Write('Estado: ');
         Read(item.estado); WriteLn;
         Write('CEP: ');
         Read(item.cep); WriteLn;
         item.status:=0;  {marca como ativo}
         FindKey(arq_ind,numreg,item.nome);
         if not OK then  {verifica se nao existem
                          chaves iguais}
         begin
            AddRec(arq_dad,numreg,item);
            AddKey(arq_ind,numreg,item.nome)
         end else WriteLn('Chaves iguais ignoradas');
      end;
   until feito;
end;  {Digitacao}
```

```
{Altera o conteudo de um endereco na lista, exceto o nome}
procedure Atualizacao;
var
   feito: boolean;
   numreg: integer;
   temp: string[30];
   item: endereco;
begin
   Write('Nome: ');
   Read(item.nome); WriteLn;
   FindKey(arq_ind,numreg,item.nome);
   if OK then
   begin
      Write('Rua: ');
      Read(item.rua); WriteLn;
      Write('Cidade: ');
      Read(item.cidade); WriteLn;
      Write('Estado: ');
      Read(item.estado); WriteLn;
      Write('CEP: ');
      Read(item.cep); WriteLn;
      item.status:=0;  {marca como ativo}
      PutRec(arq_dad,numreg,item);
   end else WriteLn('chave nao encontrada');
end;  {Atualizacao}

{Retira um endereco da lista}
procedure Remove;
var
   numreg: integer;
   nome: string[30];
   item: endereco;
begin
   Write('Digite o nome a apagar: ');
   Read(nome); WriteLn;
   FindKey(arq_ind,numreg,nome);
   if OK then
   begin
      DeleteRec(arq_dad,numreg);
      DeleteKey(arq_ind,numreg,nome);
   end else WriteLn('nao encontrado');
end; {Remove}


procedure Mostra(item: endereco);
begin
   WriteLn(item.nome);
   WriteLn(item.rua);
   WriteLn(item.cidade);
   WriteLn(item.estado);
   WriteLn(item.cep); WriteLn;
end;  {Mostra}
```

```
procedure Lista;
var
   item: endereco;
   len,numreg: integer;
begin
   len:=filelen(arq_dad)-1;
   for numreg:=1 to len do
   begin
      GetRec(arq_dad,numreg,item);
      {mostra, se houver}
      if item.status = 0 then mostra(item);
   end;
end;

{acha um dado especifico}
procedure Busca;
var
   nome: string[30];
   item: endereco;
   numreg: integer;
begin
   Write('Nome: ');
   ReadLn(nome);

   {acha a chave, se existir}
   FindKey(arq_ind,numreg,nome);
   if OK then  {se houver uma chave}
   begin
      GetRec(arq_dad,numreg,item);
      {mostra,se houver}
      if item.status = 0 then Mostra(item);
   end else WriteLn('nao encontrado');
end;  {Busca}

begin
   InitIndex;
   OpenFile(arq_dad, 'mala.dir', SizeOf(endereco));
   if not OK then
   begin
      WriteLn('criando arquivo de indice');
      MakeIndex(arq_ind, 'mala.ind',30,0);
   end;
   feito:=FALSE;
   repeat
      case Menu of
         '1': Digitacao;
         '2': Remove;
         '3': Lista;
         '4': Busca;
         '5': Atualizacao;
         '6': feito:=TRUE;
      end;
   until feito;
   CloseFile(arq_dad);
   CloseIndex(arq_ind);
end.
```

EXEMPLO: INVENTÁRIO SIMPLES

Para mostrar como é fácil criar novas aplicações, uma vez que você conhece as regras básicas do **Turbo Access**, eis um programa de inventário simples. O registro usado para guardar as informações é o seguinte:

```
type
   inv = record
        status: integer;
        nome: string[30];
        descricao: string[40];
        quantidade: integer;
        preco: real;
   end;
```

Com **SizeOf**, calculamos o seu tamanho, que é 82. Usando este tamanho, e sabendo que o comprimento da chave é 30, **SETCONST.PAS** cria as seguintes definições de constantes:

```
Const
   {Estas constantes sao geradas pelo programa
    SETCONST.PAS, parte integrante do DataBase Toolbox.}
   MaxDataRecSize = 108;
   MaxKeyLen      =  30;
   PageSize       =  24;
   Order          =  12;
   PageStackSize  =  10;
   MaxHeight      =   4;
```

As únicas mudanças necessárias para converter as rotinas do programa de mala direta para uso neste programa são alterações nas mensagens que aparecem na tela. O programa completo de inventário é mostrado a seguir:

```
program Controle_Estoque;

Const
   {Estas constantes sao geradas pelo programa
    SETCONST.PAS, parte integrante do DataBase Toolbox.}
   MaxDataRecSize = 108;
   MaxKeyLen      =  30;
   PageSize       =  24;
   Order          =  12;
   PageStackSize  =  10;
   MaxHeight      =   4;


type
   estoque = record
        status: integer;
        nome: string[30];
```

```
        descricao: string[40];
        quantidade: integer;
        preco: real;
    end;

{estes arquivos contem as rotinas de banco de dados}
{$I access.box}  {rotinas basicas}
{$I addkey.box}  {inserir dados}
{$I delkey.box}  {apagar dados}
{$I getkey.box}  {busca na arvore}

var
    arq_dad: DataFile;
    arq_ind: Indexfile;
    feito: boolean;

function Menu:char;  {retorna opcao do usuario}

var
    car: char;

begin
    WriteLn('1. Digitar produto');
    WriteLn('2. Apagar um produto');
    WriteLn('3. Ver lista de produtos');
    WriteLn('4. Procurar um produto');
    WriteLn('5. Atualizar lista');
    WriteLn('6. Fim');
    repeat
        WriteLn;
        Write('Digite a sua opcao: ');
        Read(car); car:=UpCase(car); WriteLn;
    until (car >= '1') and (car <= '6');
    Menu:=car;
end;  {Menu}

{entrar dados}
procedure Digitacao;
var
    feito: boolean;
    numreg: integer;
    temp: string[30];
    item: estoque;
begin
    feito:=FALSE;
    repeat
        Write('Produto: ');
        Read(item.nome); WriteLn;
        if Length(item.nome)=0 then feito:=TRUE
        else
        begin
            Write('Descricao: ');
            Read(item.descricao); WriteLn;
            Write('Quantidade: ');
            Read(item.quantidade); WriteLn;
            Write('Preco: ');
            Read(item.preco); WriteLn;
            item.status:=0;  {marca como ativo}
```

```
         FindKey(arq_ind,numreg,item.nome);
         if not OK then  {verifica se nao existem
                          chaves iguais}
         begin
            AddRec(arq_dad,numreg,item);
            AddKey(arq_ind,numreg,item.nome)
         end else WriteLn('Chaves iguais ignoradas');
      end;
   until feito;
end;  {Digitacao}

{Altera o conteudo de um item na lista, exceto o nome do produto}
procedure Atualizacao;
var
   feito: boolean;
   numreg: integer;
   temp: string[30];
   item: estoque;
begin
   Write('Produto: ');
   Read(item.nome); WriteLn;
   FindKey(arq_ind,numreg,item.nome);
   if OK then
   begin
      Write('Descricao: ');
      Read(item.descricao); WriteLn;
      Write('Quantidade: ');
      Read(item.quantidade); WriteLn;
      Write('Preco: ');
      Read(item.preco); WriteLn;
      item.status:=0;  {marca como ativo}
      PutRec(arq_dad,numreg,item);
   end else WriteLn('chave nao encontrada');
end;  {Atualizacao}

{Retira um produto da lista}
procedure Remove;
var
   numreg: integer;
   nome: string[30];
   item: estoque;
begin
   Write('Digite o produto a apagar: ');
   Read(nome); WriteLn;
   FindKey(arq_ind,numreg,nome);
   if OK then
   begin
      DeleteRec(arq_dad,numreg);
      DeleteKey(arq_ind,numreg,nome);
   end else WriteLn('nao encontrado');
end; {Remove}

procedure Mostra(item: estoque);
begin
   WriteLn('Produto:  ',item.nome);
   WriteLn('Descricao: ',item.descricao);
   WriteLn('Quantidade em estoque: ',item.quantidade);
   WriteLn('Preco inicial:  ',item.preco:10:2);
```

```
    WriteLn;
end;  {Mostra}

procedure Lista;
var
    item: estoque;
    len,numreg: integer;
begin
    len:=filelen(arq_dad)-1;
    for numreg:=1 to len do
    begin
        GetRec(arq_dad,numreg,item);
        {mostra, se houver}
        if item.status = 0 then mostra(item);
    end;
end;

{acha um dado especifico}
procedure Busca;
var
    nome: string[30];
    item: estoque;
    numreg: integer;
begin
    Write('Produto: ');
    ReadLn(nome);

    {acha a chave, se existir}
    FindKey(arq_ind,numreg,nome);
    if OK then  {se houver uma chave}
    begin
        GetRec(arq_dad,numreg,item);
        {mostra,se houver}
        if item.status = 0 then Mostra(item);
    end else WriteLn('nao encontrado');
end;  {Busca}

begin
    InitIndex;
    OpenFile(arq_dad, 'ctr.est', SizeOf(estoque));
    if not OK then
    begin
        WriteLn('criando arquivo de indice');
        MakeIndex(arq_ind, 'mala.ind',30,0);
    end;
    feito:=FALSE;
    repeat
        case Menu of
            '1': Digitacao;
            '2': Remove;
            '3': Lista;
            '4': Busca;
            '5': Atualizacao;
            '6': feito:=TRUE;
        end;
    until feito;
    CloseFile(arq_dad);
    CloseIndex(arq_ind);
end.
```

O programa de mala direta e o programa de inventário usam o mesmo esquema básico. Eles podem ser modificados para resolver virtualmente qualquer problema de manutenção de bancos de dados.

TURBOSORT

O Database Toolbox inclui a função **TurboSort**, que é um **QuickSort** genérico. Ela pode ser usada para ordenar qualquer tipo de informação, desde que armazenada em pelo menos dois bytes. O algoritmo *QuickSort* foi usado por ser o método de ordenação mais rápido na maioria dos casos, como vimos no Capítulo 2. O **TurboSort** vem no arquivo **SORT.BOX**, que deve ser incluído nos programas que o utilizarem. Sua definição é a seguinte:

```
function TurboSort(tam_item: integer) : integer;
```

O parâmetro **tam_item** é o tamanho dos itens a serem ordenados; este número deve ser calculado com o auxílio de **SizeOf.** O valor retornado por **TurboSort** deve ser interpretado de acordo com a Tabela 10-3.

TurboSort pode ordenar até 32.767 itens. Em geral, a ordenação será feita em RAM (para uma maior velocidade), mas um arquivo temporário em disco será criado, se necessário.

Tabela 10-3 Códigos de retorno do TurboSort

Valor	Significado
0	Ordenação bem-sucedida.
3	Memória insuficiente.
8	Tamanho do item menor que 2.
9	Mais de 32.767 itens para ordenação.
10	Erro de gravação.
11	Erro de leitura.
12	Impossível criar arquivo temporário.

InP, OutP e Less

TurboSort tem três fases de operação: entrada dos dados a serem ordenados, ordenação dos dados e saída dos dados ordenados. Para auxiliar TurboSort em sua tarefa, você deve criar três rotinas chamadas **InP, OutP** e **Less.** Estas rotinas são declaradas **forward** no arquivo **SORT.BOX**, mas você deve implementá-las.

O procedimento **InP** é usado para fornecer os dados para o **TurboSort,** um item de cada vez. A passagem propriamente dita ocorre no procedimento **SortRelease,** também definido em **SORT.BOX.** Sua declaração é:

```
procedure SortRelease(item);
```

Como o parâmetro **item** não tem tipo, qualquer tipo de dado pode ser ordenado. Um exemplo de procedimento **InP** que lê 10 números inteiros do teclado, passando-os para **TurboSort,** é dado a seguir:

```
procedure InP;
var
   i: integer;
begin
   for i:=1 to 10 do ReadLn(dado[i]);
   for i:=1 to 10 do SortRelease(dado[i]);
end;  {InP}
```

O procedimento **OutP** é usado para ler os dados ordenados pelo TurboSort, um item de cada vez, usando **SortReturn,** definido em **SORT.BOX.** Sua declaração é:

```
procedure SortReturn(item);
```

Como o parâmetro **item** não tem tipo, qualquer tipo de dado pode ser retornado. O procedimento **OutP** não tem informação sobre o número de itens a serem devolvidos. Por isso, a função **SortEOS** é usada para verificar o término dos dados. O seguinte exemplo de **OutP** pode ser usado com os números inteiros, gerados pelo processamento **InP**, visto anteriormente:

```
procedure OutP;
var
   dado: integer;
begin
   repeat
      SortReturn(dado);
      write(dado,' ');
   until SortEOF;
end;  {OutP}
```

A função **Less** é a mais crítica das três rotinas fornecidas pelo programador porque é executada toda vez que dois dados devem ser comparados pelo TurboSort. A função **Less** retorna o valor **TRUE** se o primeiro argumento for menor que o segundo. **Less** é declarada em **SORT.BOX** com dois parâmetros, chamados **X** e **Y**, que devem ser armazenados no mesmo endereço que duas variáveis locais do mesmo tipo do dado a ser ordenado. Isto é feito através do comando **absolute.** Neste exemplo, devemos usar a seguinte função **Less** para comparar dois inteiros:

```
function Less;
var
   primeiro: char absolute X;
   segundo: char absolute Y;
begin
   less:= primeiro < segundo;
end;  {Less}
```

Para ver como estas rotinas se encaixam, o programa ordenador, a seguir, lê 10 números inteiros, ordena-os e os exibe na tela:

```
program Ordenacao_Simples;

var
   dado: array[1..10] of integer;
   resultado: integer;

{$I sort.box}  {le as rotinas de ordenacao}

procedure InP;
var
   i: integer;
begin
   for i:=1 to 10 do ReadLn(dado[i]);
   for i:=1 to 10 do SortRelease(dado[i]);
end;  {InP}

function Less;
var
   primeiro: char absolute X;
   segundo: char absolute Y;
begin
   less:= primeiro < segundo;
end;  {Less}

procedure OutP;
var
   dado: integer;
begin
   repeat
      SortReturn(dado);
      write(dado,' ');
   until SortEOF;
end;  {OutP}
```

```
begin
   resultado:= TurboSort(sizeof(integer));
   WriteLn('Resultado da ordenacao: ',resultado);
end.
```

GINST

O programa **GINST**, incluído no Database Toolbox permite que você crie programas de instalação que podem ser fornecidos para que os usuários de seus programas possam instalá-los em seus computadores. Há muitos tipos diferentes de computadores, monitores, placas de vídeo etc. O **GINST** permite a criação de programas, em Turbo Pascal, que podem ser instalados em diversas configurações de *hardware*. Esta possibilidade é de grande interesse para o programador profissional, mas é de natureza muito técnica. Desta forma, não discutiremos este programa aqui. O leitor interessado deve consultar o *Manual do Turbo Database Toolbox*.

CAPÍTULO **11**

O TURBO PASCAL GRAPHIX TOOLBOX

O Turbo Pascal Graphix Toolbook é uma extensão do Turbo Pascal e contém uma grande variedade de rotinas gráficas. Estas rotinas se dividem nas cinco categorias seguintes:

- Gráficos básicos (pontos, linhas, *boxes* e círculos)
- Texto
- Janelas
- Gráficos de setores e de barra
- Desenho de curvas

O grande número de procedimentos, funções e opções fornecidos torna impossível a discussão de todo o Graphix Toolbox em apenas um capítulo. (O *Manual do Graphix Toolbox* tem 256 páginas!) Entretanto, uma rápida revisão de cada uma das cinco áreas, acompanhada de exemplos, pode dar a você uma idéia das capacidades deste poderoso pacote gráfico.

O HARDWARE BÁSICO

Para o IBM PC e seus compatíveis, o Graphix Toolbox requer a presença de uma das seguintes placas gráficas no sistema:

– CGA (*Color Graphics Adapter*)
– EGA (*Enhanced Graphics Adapter*)
– *Hercules Monochrome Graphics Adapter*

O Manual do Usuário do Graphix Toolbox especifica o modo de instalação do programa para cada uma das diferentes placas. Também são necessários 192K de RAM.

SISTEMAS DE COORDENADAS E MUNDOS

Todas as rotinas gráficas do Toolbox usam dois sistemas diferentes de coordenadas. O primeiro é chamado *sistema de coordenadas absolutas*, e o segundo *sistema de coordenadas de mundo*.

O sistema de coordenadas absolutas é dado pela placa gráfica que estiver sendo usada. Ele representa o número de *pixels* existentes tanto na horizontal como na vertical. (Um *pixel* é o menor ponto de endereçamento possível na tela.) Por exemplo, a placa CGA em modo 6 possui 640 *pixels* de largura e 200 *pixels* de altura. As rotinas do Toolbox usam o sistema de coordenadas XY, com o eixo X representando a dimensão horizontal e o eixo Y representando a direção vertical. Por convenção, o ponto 0,0 se localiza no canto superior esquerdo da tela. No caso da CGA, o ponto extremo do canto inferior direito é 639.199. Apesar do Toolboox poder ser usado com coordenadas absolutas, isto raramente é feito, devido às grandes vantagens oferecidas pelo sistema de coordenadas de mundo.

O sistema de coordenadas de mundo é definido com o procedimento **DefineWorld** do Toolbox, o qual especifica os pontos iniciais e finais do sistema de coordenadas de um mundo. Por exemplo

```
DefineWorld(1,0,0,1000,1000);
```

define um sistema de coordenadas para o mundo número 1. Neste sistema, o ponto mais alto e mais à esquerda da tela tem coordenadas 0,0, e o ponto mais baixo e mais à direita tem coordenadas 1000,1000. Uma vez feito isto, se este mundo for selecionado, todas as rotinas gráficas do Toolbox serão convertidas para este sistema de coordenadas, tornando-as independentes da placa gráfica que estiver sendo usada. Isto permite a você criar programas gráficos sem se preocupar com o *hardware* gráfico a ser usado, ou seja, seus programas tornam-se independentes do *hardware*. O mesmo programa pode ser

executado tanto com uma placa CGA com resolução de 640 x 200, quanto numa EGA em modo 640 x 350, sem qualquer mudança.

O uso de um sistema de coordenadas de mundo envolve três etapas diferentes. Primeiro, o mundo é definido com o procedimento **DefineWorld.** Em seguida, um mundo já definido deve ser selecionado, usando **SelectWorld.** Finalmente, uma janela deve ser selecionada para o mundo, usando **SelectWindow.** Estes procedimentos são declarados assim:

```
procedure DefineWorld(Numero_do_Mundo: integer; X_inicial, Y_inicial,
                      X_final, Y_final: real);
procedure SelectWorld(Numero_do_Mundo: integer);
procedure SelectWindow(Numero_da_Janela: integer);
```

X_inicial e **Y_inicial** especificam os valores do canto superior esquerdo; **X_final** e **Y_final** especificam os valores do canto inferior direito. A janela selecionada deve ter o mesmo número do mundo selecionado.

O fragmento abaixo define dois mundos e seleciona o mundo número 1 como ambiente de trabalho corrente:

```
DefineWorld(1,0,0,1000,1000);
DefineWorld(2,0,0,2000,2000);

SelectWorld(1);
SelectWindow(1);
```

Depois disto as rotinas do Toolbox estarão operando em um espaço de 1000 x 1000. Dadas estas referências, o ponto 500,500 estará no centro da tela.

O sistema de coordenadas de mundo tem ainda uma vantagem adicional, ele permite que seja dado um *zoom* em um desenho ou gráfico. Isto é feito diminuindo as coordenadas do mundo e deixando todas as outras variáveis iguais. Mais tarde, veremos um exemplo de como isto é feito.

A INICIALIZAÇÃO DO GRAPHIX TOOLBOX

Os arquivos **TYPEDEF.SYS, GRAPHIX.SYS** e **KERNEL.SYS** devem ser incluídos em

qualquer programa que use as rotinas do Graphix Toolbox. A ordem de inclusão é importante e deve ser exatamente como mostrado a seguir:

```
{$I typedef.sys}
{$I graphix.sys}
{$I Kernel.sys}
```

Algumas das rotinas do Toolbox requerem a inclusão de outros arquivos além destes.

Antes que qualquer rotina do Graphix Toolbox possa ser usada, uma chamada a **InitGraphic** deve ser feita, para inicializar o sistema gráfico. Após o uso dos gráficos, a chamada **LeaveGraphic** devolve a tela ao modo de texto.

Tabela 11-1 Rotinas gráficas básicas

Nome	Função
DrawPoint	Desenha um ponto numa localização especificada.
DrawLine	Desenha uma linha numa localização especificada.
DrawSquare	Desenha um quadrado numa localização especificada.
DrawCircle	Desenha um círculo numa localização especificada.
DrawCircleSegment	Desenha um arco numa localização especificada.
SetAspect	Define a proporção (vertical/horizontal) para as rotinas de círculo.
GetAspect	Dá a proporção atual das rotinas de círculo.

GRÁFICOS BÁSICOS

As rotinas básicas, fornecidas pelo Graphix Toolbox, parecem, à primeira vista, apenas uma duplicação das rotinas já existentes no Pascal. Entretanto, não é assim. As rotinas do Toolbox podem operar em sistemas de coordenadas de mundo, característica ausente nas versões do Turbo Pascal. A Tabela 11-1 resume os procedimentos gráficos básicos.

O programa a seguir seleciona um mundo e uma janela, desenhando, em seguida, círculos, quadrados e uma linha. O resultado aparece na Figura 11-1.

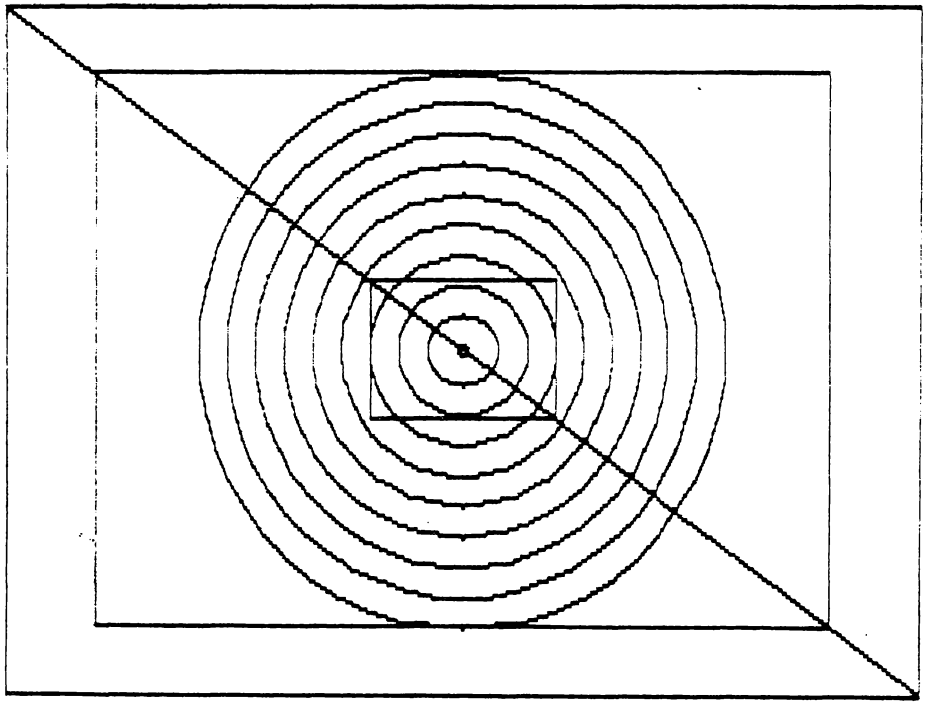


Figura 11-1 Círculos, quadrados e linha num mundo de 1000 x 1000.

```
program Graficos_Simples;

{$I typedef.sys}
{$I graphix.sys}
{$I Kernel.sys}

var
   raio: real;
   i: integer;

begin
   InitGraphic;
   DefineWorld(1,0,0,1000,1000);
   SelectWorld(1);
   SelectWindow(1);
   DrawBorder;   {desenha a moldura da janela}

   SetAspect(1);
   raio:=0.05;

   for i:=1 to 10 do
   begin
```

```
      DrawCircle(500,500,raio);
      raio:=raio+0.2;
   end;
   repeat until KeyPressed;
   ReadLn;
   DrawSquare(100,100,900,900,false);
   DrawSquare(400,400,600,600,false);
   repeat until KeyPressed;
   ReadLn;
   DrawLine(0,0,1000,1000);
   repeat until KeyPressed;
   LeaveGraphic;
end.
```

O procedimento **SetAspect** é usado para definir e modificar a proporção dos círculos a serem desenhados. Qualquer argumento diferente de 1 produz uma elipse em vez de um círculo. O procedimento **DrawBorder** desenha uma moldura ao redor da janela ativa.

Para entender o efeito das coordenadas do mundo, mude a linha do comando **DefineWorld** para

```
DefineWorld(1,0,0,2000,2000);
```

e rode o programa novamente. O resultado será parecido com o desenho mostrado na Figura 11-2. Note que nem todos os círculos cabem no mundo de 2000 x 2000. Quando isto ocorre, o Toolbox adapta a figura, cortando os lados.

PROCEDIMENTOS GRÁFICOS DE TEXTO

O Graphix Toolbox permite que se mostre texto numa tela gráfica de dois modos diferentes. O primeiro método é usando **Write** e **WriteLn,** os procedimentos padrões de I/O do Pascal, que produzem um conjunto de caracteres dependentes da máquina. Um conjunto de caracteres dependentes da máquina é determinado pelo *hardware* do computador, e contém o tipo de caractere que você normalmente vê. Entretanto, o Toolbox também permite que você desenhe caracteres independentes da máquina, usando **DrawText** e **DrawTextW,** procedimentos que desenham letras de tamanho variável em uma tela gráfica ou numa janela. Estas rotinas são de grande interesse.

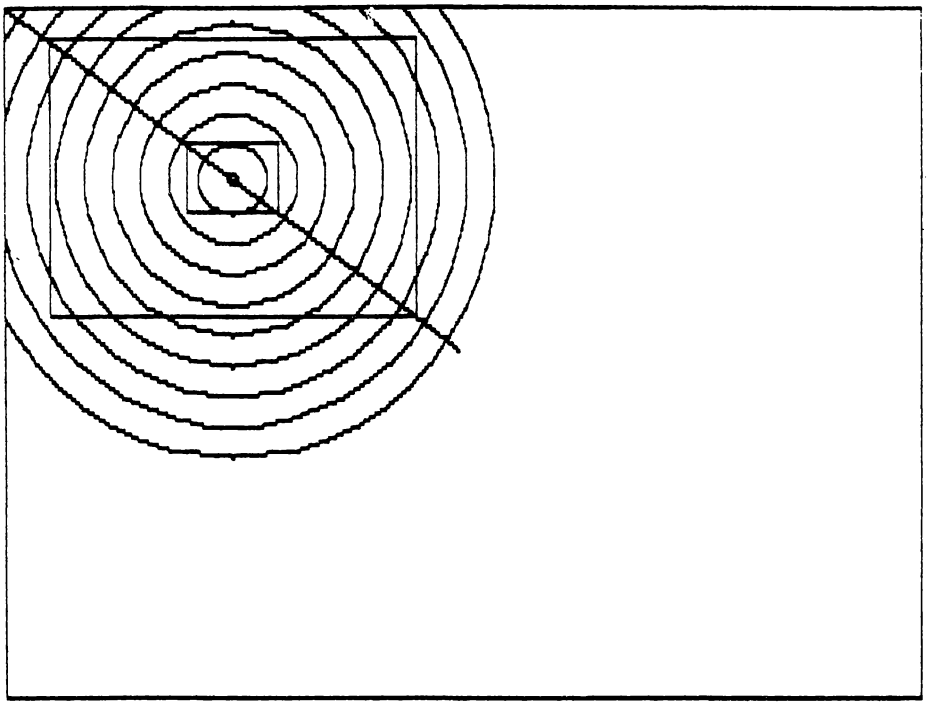


Figura 11-2 Círculos, quadrados e linha num mundo de 2000 x 2000.

O CONJUNTO DE CARACTERES INDEPENDENTES DA MÁQUINA

Cada caractere independente da máquina é construído usando-se uma matriz de *pixels* de 4 x 6. Por exemplo, a Figura 11-3 mostra como a letra "E" é construída. Como o conjunto de caracteres independentes da máquina é desenhado pelo próprio Toolbox, é possível variar seu tamanho usando uma escala.

DrawText e DrawTextW

Os procedimentos **DrawText** e **DrawTextW** são declarados assim:

```
procedure DrawText(X,Y,Escala: integer; Msg: texto);
procedure DrawTextW(X,Y,Escala: integer; Msg: texto);
```

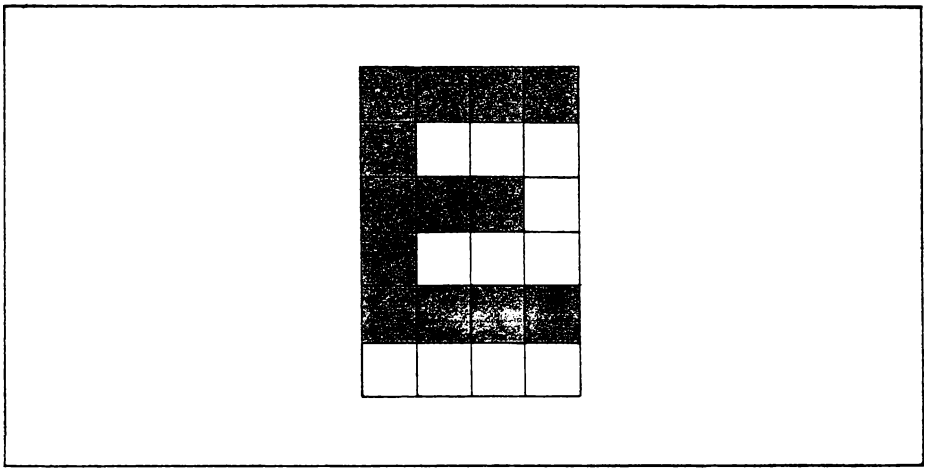


Figura 11-3 A letra "E" desenhada numa matriz de pixels de 4 x 6.

Estes procedimentos escrevem a seqüência de caracteres contida em **Msg** nas coordenadas especificadas em **X,Y**, do tamanho especificado em **Escala.** O Toolbox declara **Msg** como sendo uma **string** de tamanho máximo. Entretanto, você pode usar o tipo de seqüência de caracteres que for conveniente.

O pequeno programa a seguir mostra os primeiros seis tamanhos de texto. O resultado aparece na Figura 11-4.

```
program Graficos_Texto;

{$I typedef.sys}
{$I graphix.sys}
{$I kernel.sys}

var
   i: integer;

begin
   InitGraphic;
   DefineWorld(1,0,0,1000,1000
   SelectWorld(1);
   SelectWindow(1);
   DrawBorder;    {desenha a moldura da janela
   for i:=1 to 6 do
   begin
      DrawText(10,i*20,i, Testando 1,2,3
   end;
   repeat until KeyPressed;
   LeaveGraphic;
end.
```

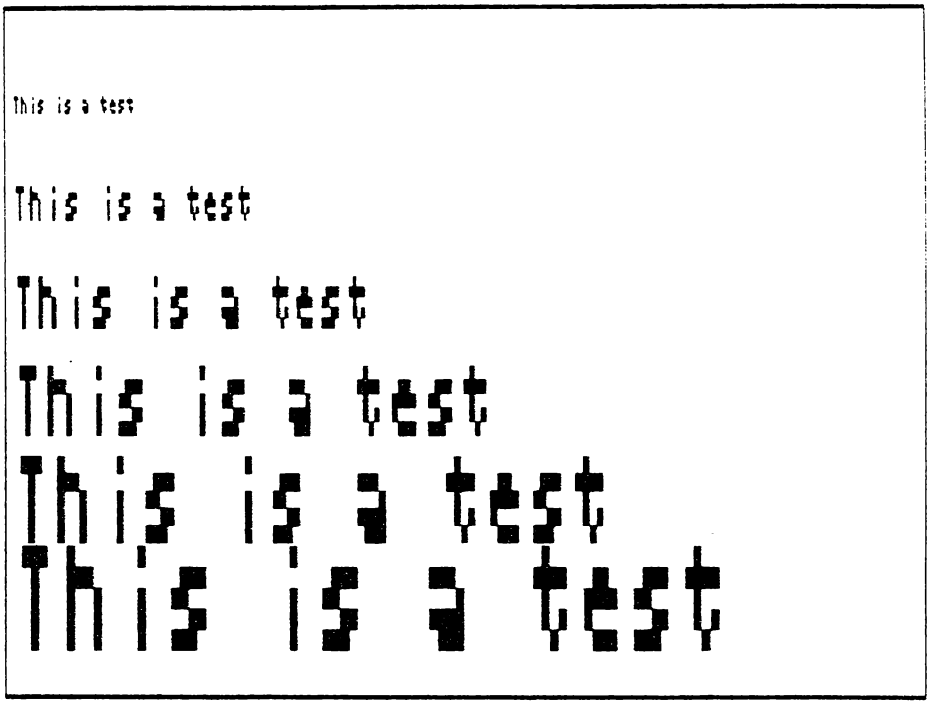


Figura 11-4 Caracteres independentes da máquina.

A maior vantagem de se usar caracteres independentes da máquina no lugar dos caracteres definidos pelo computador é a possibilidade de mudar seu tamanho de acordo com as necessidades de cada programa.

JANELAS

O Graphix Toolbox permite que você crie e mantenha uma ou mais janelas. Cada janela pode ser associada a seu próprio sistema de coordenadas de mundo. Os dois procedimentos-chaves na criação de janelas são **DefineWindow** e **SelectWindow.** Eles são declarados do seguinte modo:

```
procedure DefineWindow(Numero_da_Janela,X1,Y1,X2,Y2: integer);
procedure SelectWindow(Numero_da_Janela: integer);
```

Em **DefineWindow, X1,Y1** é a localização absoluta do canto superior esquerdo, e **X2,Y2** são as coordenadas do canto inferior direito. Um aspecto incomum de **DefineWindow** é que a coordenada **X** é medida em unidades de 8 *pixels*. Assim, o comando

```
DefineWindow(1,0,0,10,10);
```

define uma janela de 10 *pixels* de altura por 80 de comprimento. (A coordenada X é dada em unidades de 8 *pixels* porque todas as janelas devem estar alinhadas com as fronteiras de byte na RAM de vídeo.)

Para associar um mundo a uma janela deve-se fazer o seguinte:

1. Selecionar um mundo.
2. Selecionar uma janela.

Para incluir um cabeçalho com uma mensagem, o cabeçalho deve ser associado à janela e então "ligado". Para executar isto são usados os procedimentos **DefineHeader** e **SetHeaderOn.** Estes procedimentos são declarados assim:

```
procedure DefineHeader(Numero_da_Janela: integer; Msg: Texto);
procedure SetHeaderOn;
```

Uma chamada a **DrawBorder** desenha uma moldura em torno da janela ativa. **DrawBorder** não requer nenhum parâmetro.

O programa a seguir mostra a ordem correta em que os vários procedimentos devem ser chamados para criar uma janela com moldura e cabeçalho. O resultado é mostrado na Figura 11-5.

```
program Uma_Janela;

{$I typedef.sys}
{$I graphix.sys}
{$I kernel.sys}

var
   i: integer;

begin
   InitGraphic;
   DefineWorld( 1,0,0,1000,1000 );
   DefineWindow( 1,20,20,40,100 );
   DefineHeader( 1,'Cabecalho' );
   SetHeaderOn;
```

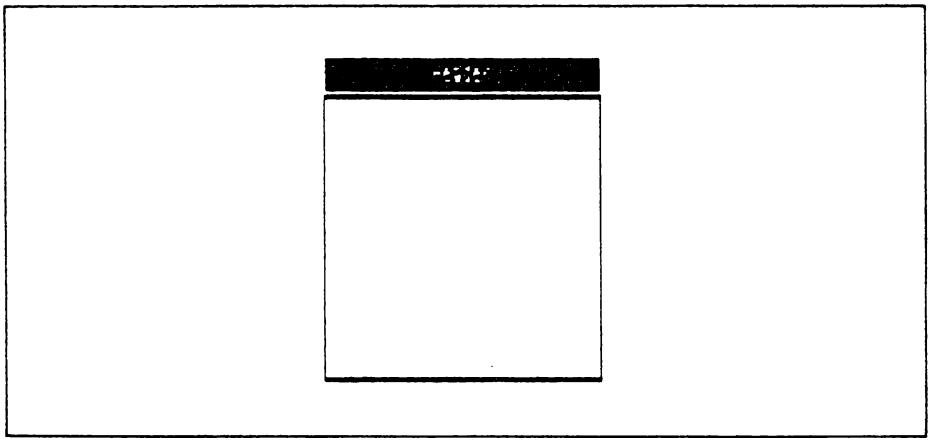


Figura 11-5 Uma janela simples com uma moldura e um cabeçalho.

```
   SelectWorld(1);
   SelectWindow(1);
   DrawBorder;   {desenha a moldura da janela}
   repeat until KeyPressed;
   LeaveGraphic;
end.
```

CRIANDO UM "ZOOM" COM AS COORDENADAS DE MUNDO

Criando janelas com diferentes coordenadas de mundo você pode criar um efeito *zoom* numa tela gráfica. Observe o seguinte comando, que desenha uma linha:

```
DrawLine(0,0,100,100);
```

Se o mundo ativo onde esta linha será desenhada estiver definido como

```
DefineWorld(1,0,0,100,100);
```

então a linha será uma diagonal, cortando a tela ou a janela de um canto a outro. Entretanto, se o mundo for definido como

```
DefineWorld(1,0,0,200,200);
```

então a mesma linha cobrirá apenas a metade da distância entre um canto e outro. Objetos gráficos serão sempre desenhados na proporção do sistema de coordenadas do mundo. Num mundo maior, os objetos parecerão menores; num mundo menor, os objetos parecerão maiores.

O programa a seguir cria um *zoom* num canto de um quadrado, permitindo um exame mais cuidadoso de uma pequena linha. O resultado gráfico do programa é mostrado na Figura 11-6.

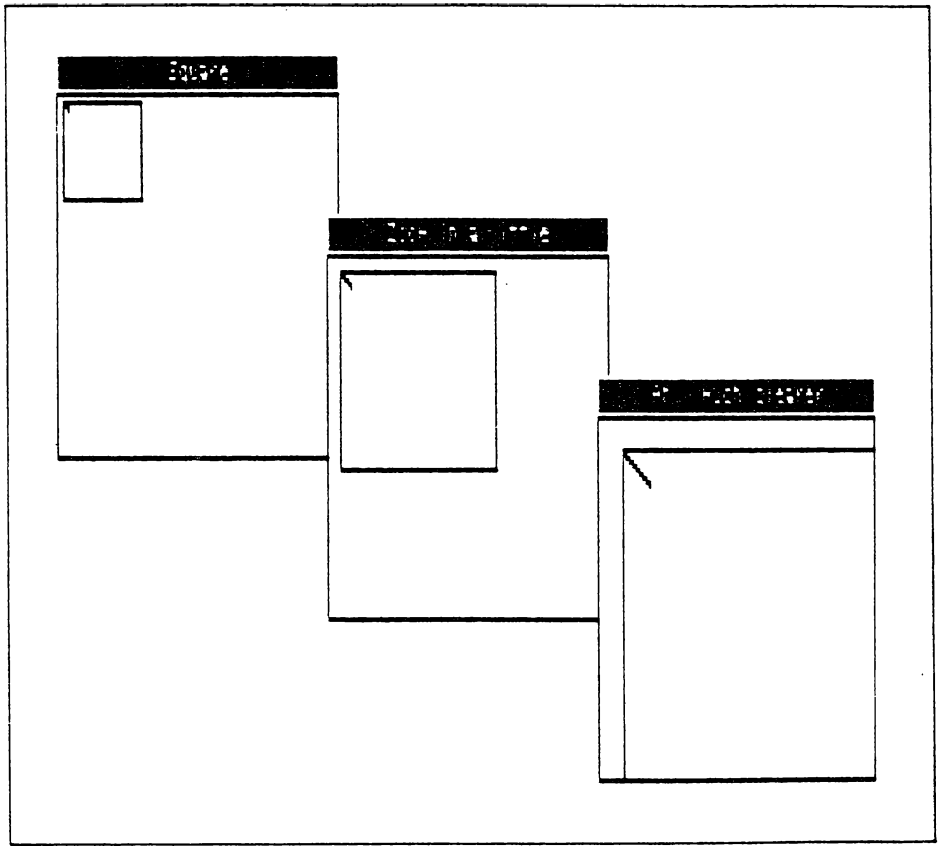


Figura 11-6 O efeito *zoom* usando janelas.

```
program Janelas;

{$I typedef.sys}
{$I graphix.sys}
{$I Kernel.sys}
{$I windows.sys}

var
   i: integer;

procedure Faz_Janela;
begin

   DefineWindow(1,0,0,20,100);
   DefineHeader(1,'Quadrado');
   DefineWorld(1,0,0,400,400);
   SelectWorld(1);
   SelectWindow(1);
   SetHeaderOn;
   SetBackground(0);
   DrawBorder;   {desenha a moldura da janela}

   DefineWindow(2,20,40,40,140);
   DefineHeader(2,'Um pequeno zoom');
   DefineWorld(2,0,0,200,200);
   SelectWorld(2);
   SelectWindow(2);
   SetHeaderOn;
   SetBackground(0);
   DrawBorder;   {desenha a moldura da janela}

   DefineWindow(3,40,80,60,180);
   DefineHeader(3,'Ah...muito melhor');
   DefineWorld(3,0,0,100,100);
   SelectWorld(3);
   SelectWindow(3);
   SetHeaderOn;
   SetBackground(0);
   DrawBorder;   {desenha a moldura da janela}
end;

begin
   InitGraphic;
   Faz_Janela;

   {Zoom num canto da janela}
   for i:=1 to 3 do
   begin
      SelectWorld(i);
      SelectWindow(i);
      DrawSquare(10,10,120,120,false);
      DrawLine(10,10,20,20);
   end;
   repeat until KeyPressed;
   LeaveGraphic;
end.
```

GRÁFICOS DE BARRA E DE SETORES

O Graphix Toolbox fornece rotinas que permitem a construção de gráficos de barra e de setores ("pizza"). Existem dois procedimentos de gráfico de setores, **DrawPolarPie** e **DrawCartPie**. A diferença entre eles está no modo pelo qual a localização e o raio gráfico são especificados, em coordenadas polares ou cartesianas. O mais simples de usar é **DrawPolarPie**. Ele é definido como

```
procedure DrawPolarPie(X,Y,Raio,Theta,Interno,Externo:real,Info: PieArray;
                       Num,Opcao,Tamanho_do_Texto: integer);
```

X,Y é o centro do gráfico, **raio** é o raio e **Theta** é o ângulo, em graus, do primeiro segmento do gráfico. Os parâmetros **Interno** e **Externo** especificam o tamanho das linhas que relacionam os segmentos do gráfico às legendas. **Info** é uma matriz do tipo **PieArray**, a qual é definida assim:

```
type
   Graf_Pizza = record
                Area: real;
                Texto: wrstring;
   end;

   Matriz_Pizza = array[1..MaxPiesGlb] of Graf_Pizza;
```

O parâmetro **Num** deve especificar o número de segmentos, e **Opção** é escolhido de acordo com a tabela a seguir:

Opção	Significado
0	Nenhuma legenda.
1	Apenas legenda de texto.
2	Legenda de texto e valores.
3	Apenas legenda de valores.

Finalmente, **Tamanho_do_Texto** especifica o tamanho dos caracteres independentes da máquina a serem usados nas legendas. Programas que usem **DrawPolarPie** devem, também, incluir os dois arquivos seguintes, nesta ordem:

```
{$I circsegm.hgh}
{$I pie.hgh}
```

Para usar **DrawPolarPie** você primeiro carrega os valores dos segmentos do gráfico de setores em **Info.Area** e as legendas associadas a estes valores em **Info.Texto.** Finalmente, os valores dos outros valores devem ser especificados, e o gráfico desenhado. O procedimento a seguir desenha o gráfico de setores mostrado na Figura 11-7:

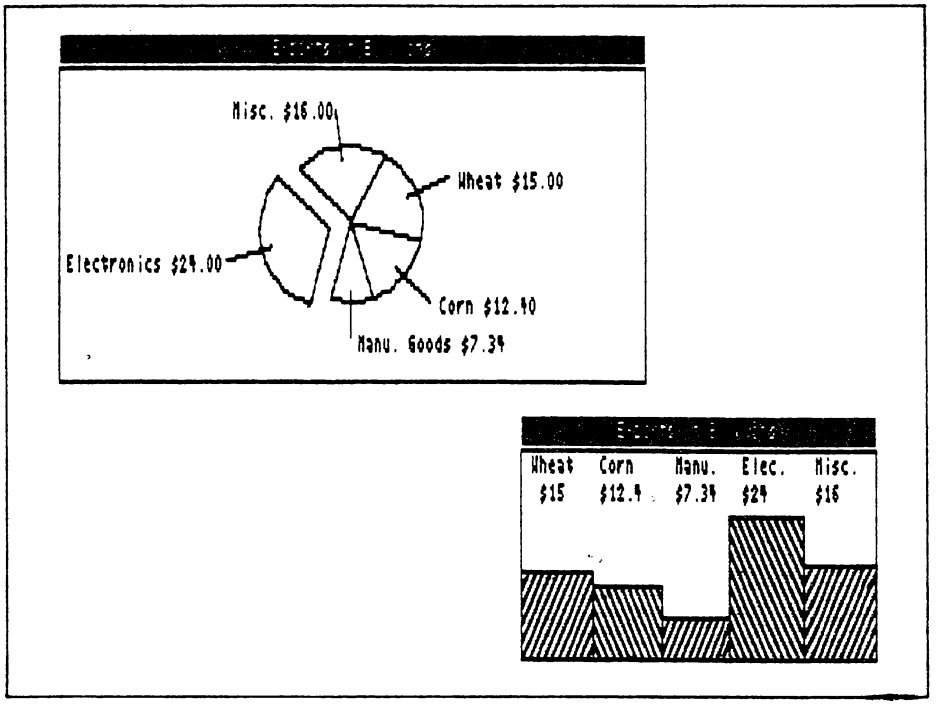


Figura 11-7 Um gráfico de setores e um gráfico de barras.

```
procedure Pizza;
var
   Raio,Theta,Interno,Externo: real;
   Modo,Tamanho:integer;
   Produtos: Matriz_Pizza;

begin
   DefineWindow(1,0,0,50,100);
   DefineHeader(1,'Exportacao (em milhoes de dolares');
   DefineWorld(1,0,0,1000,1000);
   SelectWorld(1);
   SelectWindow(1);
   SetHeaderOn;
   DrawBorder;
   Produto[1].Texto:='Cafe''s';
   Produto[2].Texto:='Soja s';
```

```
    Produto[3].Texto:='Aluminio $';
    Produto[4].Texto:='Ferro $';
    Produto[5].Texto:='Outros $';
    Produto[1].Area:=15;
    Produto[2].Area:=12.4;
    Produto[3].Area:=7.34;
    Produto[4].Area:=-24;  {para o outro lado}
    Produto[5].Area:=16;

    Raio:=125;
    Theta:=60;

    SetAspect(1.0);

    Interno:=0.85;
    Externo:=1.5;

    Modo:=2;
    Tamamnho:=1;

    DrawPolarPie(500,500,Raio,Theta,Interno,
                 Externo,Produtos,5,Modo,Tamanho);
end;  {Pizza}
```

Para desenhar um gráfico de barras, o procedimento **DrawHistogram** deve ser usado. Ele é declarado assim:

```
procedure DrawHistogram(Info: PlotArray; Num: integer; Preenche:
                                    booleano; Densidade: integer);
```

O parâmetro **Info** contém o valor associado a cada barra. Ele é uma matriz bidimensional, onde, qualquer que seja **i, Info[i,1]** é reservado para uso interno e **Info[i,2]** contém o valor da barra "i". Se **Preenche** é **TRUE**, cada barra será preenchida. O parâmetro **Densidade** indica a densidade do preenchimento de cada barra, com o valor 1 indicando o preenchimento mais denso. Não é possível desenhar legendas diretamente, usando **DrawHistogram.**

Qualquer programa que use **DrawHistogram** deve incluir os dois arquivos seguintes, nesta ordem:

```
{$I hatch.hgh}
{$I histogrm.hgh}
```

O procedimento a seguir exemplifica o uso de **DrawHistogram.** Ele produz o gráfico mostrado na Figura 11-7:

```
procedure Barra;
var
   Produtos: PlotArray;
begin
   DefineWindow(2,40,110,70,180);
   DefineHeader(1,'Exportacao (em milhoes de dolares');
   DefineWorld(2,0,0,30,35);
   SelectWorld(2);
   SelectWindow(2);
   SetHeaderOn;
   SetBackgroundOn;
   DrawBorder;

   Produto[1,2].Area:=15;
   Produto[2,2].Area:=12.4;
   Produto[3,2].Area:=7.34;
   Produto[4,2].Area:=-24;  {para o outro lado}
   Produto[5,2].Area:=16;

   DrawHistogram(Produtos,5,true,5)
   DrawTextW(1,2,1, 'Cafe''     Soja      Aluminio       Ferro      Outros');
   DrawTextW(1,7,1, ' $15       $12.4     $7.34          $24        $16 ');
end;  {Barra}
```

Note que as legendas são introduzidas manualmente, usando-se para isto o procedimento **DrawTextW.**

Para sua conveniência, os procedimentos gráficos de barra e setores foram incluídos, aqui, num programa simples:

```
program Demo_Graficos;

{$I typedef.sys}
{$I graphix.sys}
{$I kernel.sys}
{$I circsegm.hgh}
{$I pie.hgh}
{$I hatch.hgh}
{$I histogrm.hgh}

procedure Pizza;
var
   Raio,Theta,Interno,Externo: real;
   Modo,Tamanho:integer;
   Produto: PieArray;

begin
   DefineWindow(1,0,0,50,100);
   DefineHeader(1,'Exportacao (em milhoes de dolares)');
   DefineWorld(1,0,0,1000,1000);
   SelectWorld(1);
   SelectWindow(1);
   SetHeaderOn;
   DrawBorder;
   Produto[1].Text:='Cafe''$';
```

```
    Produto[2].Text:='Soja $';
    Produto[3].Text:='Aluminio $';
    Produto[4].Text:='Ferro $';
    Produto[5].Text:='Outros $';
    Produto[1].Area:=15;
    Produto[2].Area:=12.4;
    Produto[3].Area:=7.34;
    Produto[4].Area:=-24;  {retira fatia}
    Produto[5].Area:=16;

    Raio:=125;
    Theta:=60;

    SetAspect(1.0);

    Interno:=0.85;
    Externo:=1.5;

    Modo:=2;
    Tamanho:=1;

    DrawPolarPie(500,500,Raio,Theta,Interno,
                 Externo,Produto,5,Modo,Tamanho);
end;  {Pizza}


procedure Barra;
var
    Produto: PlotArray;
begin
    DefineWindow(2,40,110,70,180);
    DefineHeader(2,'Exportacao (em milhoes de dolares)');
    DefineWorld(2,0,35,30,0);
    SelectWorld(2);
    SelectWindow(2);
    SetHeaderOn;
    SetBackground(0);
    DrawBorder;

    Produto[1,2]:=15;
    Produto[2,2]:=12.4;
    Produto[3,2]:=7.34;
    Produto[4,2]:=24;
    Produto[5,2]:=16;

    DrawHistogram(Produto,5,true,3);
    DrawTextW(0,2,1,  ' Cafe''   Soja   Aluminio  Ferro   Outros');
    DrawTextW(1,7,1,  '  $15   $12.4   $7.34      $24      $16 ');
end;  {Barra}

begin
    InitGraphic;
    Pizza;
    Barra;
    repeat until KeyPressed;
    LeaveGraphic;
end.
```

TRAÇADO E AJUSTE DE CURVAS

O Graphix Toolbox contém um excelente conjunto de rotinas de traçado e ajuste de curvas. Dois procedimentos, **DrawPoly** e **Spline**, serão examinados aqui.

O procedimento **DrawPoly** é usado para traçar qualquer forma arbitrária, dadas as coordenadas de seus vértices, na tela. Ele é declarado assim:

```
procedure DrawPoly(Info: PlotArray; Comeco,Fim,Codigo,Escala,
                   Linha: integer);
```

Info é a matriz do tipo **PlotArray** que contém as coordenadas **X,Y** de cada ponto a ser desenhado. **Comeco** e **Fim** são índices da matriz, indicando o primeiro e o último ponto a serem desenhados. O parâmetro **Codigo** indica o símbolo a ser usado para representar os pontos na tela. Os valores possíveis de **Codigo** são dados na tabela a seguir:

Código	**Significado**
0	Usa linhas entre os pontos
1	+
2	X
3	Box vazado
4	Box cheio
5	Diamante
6	Y
7	*
8	0
9	

Escala determina o tamanho do símbolo que representa os pontos. **Linha** é usado para especificar como as linhas são traçadas de cada um dos eixos para cada ponto. Se **Linha** for menor que 0, linhas são traçadas a partir do ponto 0 do eixo Y; se **Linha** for 0, nenhuma linha será traçada; e se **Linha** for maior que 0, linhas serão traçadas de baixo para cima, a partir do lado inferior da tela. Em geral, **Linha** é 0.

O procedimento **Spline** transforma poucos pontos em muitos pontos, estes últimos representando uma curva ajustada. Declara-se assim:

```
procedure Spline(Entrada: PlotArray; Pontos: integer; Inicio,Fim:
                 real; var Saida: PlotArray; Pontos_Saida: integer);
```

A informação inicial é especificada em **Entrada,** sendo **Pontos** o número de pontos na matriz. **Inicio** e **Fim** indicam as coordenadas no eixo Y do primeiro e do último ponto. Os pontos ajustados são o resultado de **Saida,** e o número de pontos ajustados é especificado em **Pontos_Saida.**

O seguinte programa demonstra a excelente performance destes procedimentos:

```
program Ajuste_de_Curvas;

{$I typedef.sys}
{$I graphix.sys}
{$I Kernel.sys}
{$I polygon.hgh}
{$I spline.hgh}

procedure Ajuste;
var
   N,i: integer;
   A,B: PlotArray;

begin
   N:=10;

   {Gera alguns pontos}
   for i:=1 to N do
   begin
      A[i,1]:=i-1;
      A[i,2]:=random;
   end;
   DefineHeader(1,'Ajuste de Curva');
   DefineWorld(1,-1,1.1,10,0);
   SelectWorld(1);
   SelectWindow(1);
   SetHeaderOn;
   DrawBorder;

   DrawPolygon(A,1,N,-7,2,0);          {desenha os pontos}
   spline(A,N,A[1,1],a[N,1],B,50);     {ajuste}
   DrawPolygon(B,1,50,0,0,0);          {desenha a curva}
end;  {Ajuste}

begin
   InitGraphic;
   Ajuste;
   repeat until KeyPressed;
   LeaveGraphic;
end.
```

Como os pontos são gerados por **Random,** o gerador de números aleatórios, cada execução produz uma curva diferente. A Figura 11-8 mostra uma tela criada com este programa.

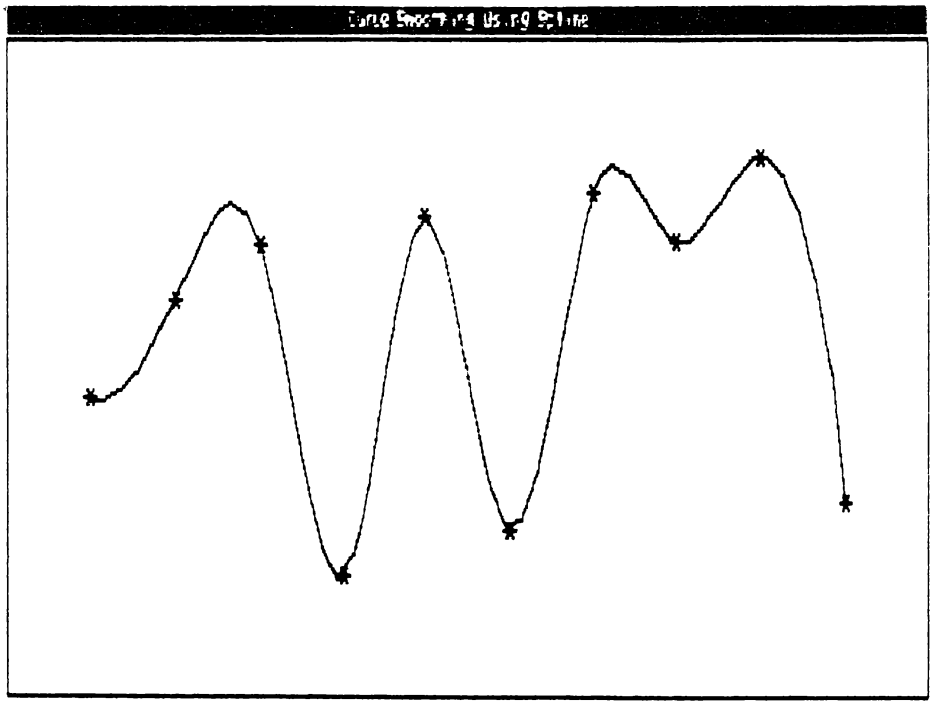


Figura 11-8 Uma tela do programa de ajuste de curva.

CAPÍTULO **12**

EFICIÊNCIA, PORTABILIDADE E DEPURAÇÃO

A habilidade para escrever programas sem defeitos, que façam uso eficiente dos recursos disponíveis e sejam fáceis de transpor de uma máquina para outra, é característica do programador profissional. É esta habilidade que transforma a ciência da computação na arte da computação, pois poucas são as técnicas disponíveis nesta área. Este capítulo apresenta alguns métodos através dos quais eficiência, portabilidade e exatidão podem ser conseguidas.

EFICIÊNCIA

O termo *eficiência,* quando aplicado a programas de computador, refere-se ao uso dos recursos do sistema, à velocidade de execução, ou a ambos. Por recursos do sistema entende-se a RAM, o espaço de disco, o papel impresso e tudo o que possa ser controlado, alocado e usado. A eficiência ou ineficiência de um programa dependerá de cada situação particular. Considere um programa que use 47 K de memória RAM, 2 megabytes de disco, com um tempo médio de execução de 7 minutos. Se este for um programa de ordenação de matrizes rodando num Apple II, ele provavelmente não é muito eficiente. Entretanto, um programa de previsão do tempo com estas características, rodando num supercomputador Cray, é, com certeza, bastante eficiente.

Um ponto a ser considerado, quando se está buscando eficiência, é que a otimização de um aspecto do programa pode, muitas vezes, piorar outros. Por exemplo, tornar um programa mais rápido, com o uso de rotinas em linguagem de máquina, em geral o tornará também maior. Comprimir a informação no disco tornará o acesso a esta informação mais vagaroso. Estas e outras trocas feitas em nome da eficiência podem ser muito frustrantes, especialmente para o usuário final, que não pode ver como uma coisa afeta outra.

A esta altura você pode estar se perguntando se é possível, ou mesmo útil, discutir eficiência nestes termos. Existem, porém, algumas práticas de programação mais eficientes que outras. Existem mesmo algumas técnicas capazes de tornar um programa mais rápido e menor.

EVITANDO REPETIÇÃO DE INSTRUÇÕES

Mesmo os melhores programadores às vezes escrevem programas contendo *redundâncias*. Não estamos nos referindo aqui a partes do programa que possam ser transformadas em sub-rotinas, pois mesmo programadores inexperientes entendem isto com facilidade. Redundância aqui é a duplicação desnecessária de instruções dentro de uma rotina. Examine o seguinte fragmento:

```
Read(a,y);
if a < 10 then WriteLn('Erro');
if Length(y)=0 then WriteLn('Erro');
```

O comando **WriteLn('Erro')** aparece duas vezes. Tal repetição não é necessária, pois o mesmo efeito poderia ser atingido deste modo:

```
Read(a,y);
if (a < 10) or (Length(y)=0) then WriteLn('Erro');
```

A rotina acima não só é menor como também mais rápida que a primeira. Apenas um teste **if/then** é executado.

Casos muito similares ao deste exemplo dificilmente ocorrerão em programas reais, visto que as instruções redundantes estão muito próximas, sendo facilmente detectáveis. Em programas reais, rotinas e comandos redundantes estarão, em geral, distantes uns dos outros.

A redundância também pode ser causada pelo método escolhido para escrever

uma rotina. As duas funções a seguir, por exemplo, procuram por uma palavra numa matriz de seqüências:

```
type
   str80 = string[80];
   Matriz_Str = array[1..100] of str80;

function Busca1(str:Matriz_Str; palavra: str80): boolean;
{metodo correto, nao-redundante}
var
   t: integer;
begin
   Busca1:=FALSE;
   for t:=1 to 100 do
      if str[t]=palavra then Busca1:=TRUE;
end;

function Busca2(str:Matriz_Str; palavra: str80): boolean;
{metodo redundante,incorreto}
var
   t: integer;
begin
   t:=1;
   Busca2:=FALSE;
   if str[t]=palavra then Busca2:=TRUE
   else
   begin
      t:=2;
      while (t <= 100) do
      begin
         if str[t]=palavra then Busca2:=TRUE;
      end;
   end;
end;
```

O método usado para escrever a segunda função não apenas duplicou os comandos **if/then** como também apresenta duas atribuições (**t:=1** e **t:=2**) essencialmente idênticas. A primeira versão é mais rápida e utiliza muitos menos espaço na memória.

Em suma, redundâncias são causadas por falta de atenção ao escrever o programa ou pela escolha do método errado para implementar uma rotina. De qualquer modo, esta é uma falha a ser evitada.

O USO DE PROCEDIMENTOS E FUNÇÕES

Procedimentos e funções, apoiados em variáveis locais, formam a base de toda a programação estruturada. Por este mesmo motivo, procedimentos e funções são os blocos fundamentais do Turbo Pascal, bem como seus mais poderosos recursos. Portanto, esta

seção deve ser lida e estudada cuidadosamente. Examinaremos aqui certos aspectos das funções do Turbo Pascal, bem como seus efeitos sobre o tamanho e a velocidade dos programas.

O Turbo Pascal é uma linguagem que usa o *stack* de modo intensivo. Todas as variáveis locais e os parâmetros das funções usam o *stack* para armazenamento temporário. Além disto, quando uma função é chamada, o endereço de retorno também é colocado no *stack*. Isto permite à sub-rotina voltar ao local que a chamou. Quando uma função retorna ao endereço de origem, tal endereço deve ser removido do *stack*, juntamente com as variáveis locais e o parâmetro referentes àquela função. O processo de empilhar informação no *stack* é conhecido por *seqüência de chamada*, e o processo de retirada de informação do *stack*, *seqüência de retorno*. Estes processos gastam tempo, o qual aumenta proporcionalmente ao número de variáveis temporárias e parâmetros usados. Os dois exemplos a seguir dão uma idéia de como uma chamada de função pode tornar um programa lento:

```
Versao 1

for x:=1 to 100 do
   t:=calcula(x)

function calcula(q: integer): real;
var
   t: real;
begin
   calcula:=Abs(Sin(q)/100/3.1416);
end;
```

```
Versao 2

for x:=1 to 100
        t:=Abs(Sin(q)/100/3.1416);
```

A versão 2 é muito mais rápida, pois o excesso de seqüências de chamada e retorno foi eliminado. O programa seguinte, escrito num *pseudo-assembly*, mostra as seqüências de chamada e de retorno da função **calcula:**

```
; sequencia de chamada
move A,x        ;poe o valor de x no acumulador
push A
call calcula    ;a instrucao de chamada poe
                ;o endereco de retorno no stack

; sequencia de retorno
; o valor de retorno da funcao deve ser colocado num
; registrador - usaremos B
move B,stack-1  ;poe o valor em t
;return         ;volta a rotina de chamada
;a rotina de chamada entao
pop A           ;limpa o parametro usado na chamada
```

O uso da função **calcula** dentro do *loop* **for/do** causa a execução de 100 seqüências de chamada e 100 seqüências de retorno. Se você estiver querendo uma rotina rápida, esta não é a melhor maneira de escrevê-la.

Neste momento, você pode estar considerando a hipótese de escrever seus programas com apenas algumas grandes rotinas, que então rodarão rapidamente. Porém, na maioria dos casos, o pequeno diferencial de tempo assim ganho não será significativo, em comparação à imensa perda em termos de estrutura que esta prática acarreta. E existe ainda um outro problema. Se um subprograma qualquer for muito usado, ele precisará ser repetido várias vezes, tornando o programa inteiro muito grande. As sub-rotinas servem para racionalizar o uso da memória. Como regra geral, tornar um programa mais rápido o torna também maior, e tornar um programa menor o torna mais lento. Só faz sentido repetir subprogramas em vez de usar funções onde a velocidade for prioridade máxima. Em qualquer outro caso, o uso de funções e procedimentos é sempre mais recomendável.

CASE *VERSUS* A ESCADA IF/THEN/ELSE

Os seguintes fragmentos de programa são funcionalmente equivalentes, mas um deles é muito mais eficiente. Você saberia dizer qual?

```
case car of
   'a': f1(car);
   'b': f2(car);
   'c': f3(car);
   'd': f4(car);
end;
```

```
if car='a' then f1(car)
else if car='b' then f2(car)
else if car='c' then f3(car)
else if car='d' then f4(car);
```

O fragmento da esquerda é mais eficiente, pois, em geral, o comando **case** gera um programa-objeto mais compacto e rápido que uma série de comandos **if/then/else.**

O arranjo de **if/then/else** mostrado acima é conhecido como "escada **if/then/else**", pois o programa parece descer a seqüência degrau por degrau. A escada **if/then/else** é importante por permitir a tomada de decisões multilaterais, usando uma variedade de tipos de dados que um comando **case** não admite. Entretanto, se você estiver trabalhando com escalares ordinais (inteiros, caracteres, enumerações etc.), o comando **case** poderá ser usado.

TRANSPONDO PROGRAMAS

Muitas vezes ocorre de um programa, escrito numa determinada máquina, ser transposto para outro computador, com um processador diferente, um sistema operacional diferente, ou ambos. Portar (este é o nome deste processo) um programa pode tanto ser muito fácil como extremamente difícil, dependendo basicamene da forma original do programa a ser portado. Um programa facilmente transposto é dito portável. Um programa não será facilmente transposto se contiver muitas dependências de máquina, isto é, partes que só funcionam com um certo sistema operacional ou com um processador específico. O Turbo Pascal foi desenvolvido de modo a permitir a fácil transposição de programas escritos em qualquer de suas versões. Isto, entretanto, requer muito cuidado, muita atenção a detalhes e, em geral, o sacrifício da eficiência ótima, tudo devido a diferenças entre os diversos sistemas operacionais.

Portar programas de outro compilador Pascal para o Turbo Pascal pode apresentar problemas, se um conjunto diferente de expansões tiver sido usado. O inverso também é verdadeiro. Se qualquer das expansões do Turbo Pascal tiver sido usada num programa, este programa deverá ser modificado antes de poder ser executado em outro compilador Pascal.

Além de oferecer soluções para alguns dos problemas específicos desta área, esta seção também mostra como escrever programas portáveis em Pascal.

O USO DE CONSTANTES

O meio mais simples de escrever um programa portável talvez seja juntar todos os números dependentes do sistema ou do processador numa declaração **const.** Tais números incluem tamanhos de registros para acesso ao disco, comandos especiais de tela, informações sobre alocação de memória e qualquer outro número que tenha a menor chance de mudar quando o programa for transposto. O uso de declarações **const** torna a função destes números óbvia para quem estiver transpondo o programa, além de facilitar o processo de edição.

Como exemplo, aqui estão duas declarações de matrizes, e dois procedimentos que as usam. O primeiro procedimento **não usa const.**

```
{primeira versao}
var
   conta: array[1..100] of integer;
```

```
procedure f1;
var
   t: integer;
begin
   for t:=1 to 100 do conta[t]:=t;
end;

{segunda versao}
const
   MAX = 100;
var
   conta: array[1..MAX] of integer;

procedure f2;
var
   t: integer;
begin
   for t:=1 to MAX do conta[t]:=t;
end;
```

A segunda versão é obviamente melhor. Se o programa for transposto para uma máquina que permita matrizes maiores, apenas **MAX** precisará ser modificado. Esta versão não só é mais fácil de modificar como também ajuda a evitar erros de edição. Casos similares estão sempre aparecendo em programas reais, onde o ganho em portabilidade será substancial.

DEPENDÊNCIAS DO SISTEMA OPERACIONAL

Virtualmente, todos os programas comerciais usam características específicas de um determinado sistema operacional. Uma planilha eletrônica pode usar a memória de vídeo do IBM PC para permitir rápidas mudanças de tela. Um pacote gráfico usará certos comandos presentes apenas em um dado sistema operacional. Algumas dependências do sistema operacional servem para tornar os programas mais rápidos (e comercialmente viáveis). Não há, porém, razão para usar mais dependências que o estritamente necessário.

Se você precisar acessar o sistema operacional diversas vezes, o melhor será fazer isto através de um procedimento-mestre, de modo que só este procedimento tenha de ser modificado quando o programa for transposto. Por exemplo, se você for usar rotinas do sistema operacional para limpar a tela, limpar o fim de uma linha e colocar o cursor na coordenada **X, Y**, o procedimento a seguir, chamado **Chamada_SisOp**, servirá como procedimento-mestre:

```
procedure Chamada_SisOp(op,x,y: integer);
{interfaciamento com o sistema operacional}
begin
   case op of
      1: ClearScreen;
      2: ClearEOL;
      3: GotoXY(x,y);
end;
```

Apesar de tais chamadas serem padronizadas para todas as versões do Turbo Pascal, esta técnica será útil caso o programa deva ser transposto para outro compilador Pascal. Apenas os comandos seriam mudados, permanecendo a função intacta.

EXTENSÕES DO TURBO PASCAL

Se você estiver portando um programa de outro Pascal para o Turbo Pascal, as extensões deste último podem tornar o trabalho mais fácil, e até melhorar o programa em questão. As extensões referentes a seqüências de caracteres, por exemplo, tornam a manipulação destas muito mais simples.

Porém, ao portar um programa em Turbo Pascal para outro Pascal, você certamente terá de remover todas as extensões usadas. Todos os tipos **string** deverão ser mudados para **array** (matrizes de caracteres). Pior, todas as rotinas de manipulação de seqüências de caracteres, tais como **Copy, Concat** e **Pos,** estarão perdidas. Se seu programa precisar de tais procedimentos você deverá criá-los sozinho na nova versão.

Naturalmente, se você souber de antemão que o programa será transposto para outro compilador Pascal, deverá evitar o uso das extensões do Turbo Pascal.

DEPURAÇÃO

Parafraseando Thomas Edison, pode-se dizer que um programa é composto de 10% de inspiração e 90% de depuração. Bons programadores são também bons caçadores de erros. Esta seção trata de certos tipos de erros que surgem com muita freqüência no uso do Turbo Pascal.

PROBLEMAS COM PONTEIROS

É muito comum no Turbo Pascal o uso incorreto de ponteiros. Problemas com ponteiros se dividem em dois grupos: erros quanto aos conceitos e operadores de ponteiros e uso acidental de ponteiros inválidos. Para resolver problemas do primeiro tipo, basta entender o funcionamento dos ponteiros na linguagem Pascal. O segundo tipo de problema resolve-se pela verificação prévia da validade ou não do ponteiro a ser usado.

O programa abaixo ilustra um típico erro no uso de ponteiros no Turbo Pascal:

```
program ERRADO; {este programa esta errado}

type
   ponteiro = ^objeto;

   objeto = record
      x: char;
      y: integer;
      nome: string[80];
   end;

var
   p: ponteiro;

begin
   p^.nome:= 'tomas';
   p^.x:='g';
   p^.y:=100;
end.
```

Este programa tem toda a chance de entrar em colapso (e talvez até levar o sistema operacional junto). Isto ocorre porque nenhum valor foi atribuído ao ponteiro **p** (com o comando **New**). **p** então contém um valor aleatório, e pode estar apontando para qualquer ponto da memória. Como certamente não é isto que se deseja, a seguinte linha deve ser acrescentada ao programa (antes do primeiro uso de **p**):

```
New(p);
```

Ponteiros "chucros" são extremamente difíceis de rastrear. Caso você esteja atribuindo valores a uma variável ponteiro que não contenha um endereço válido, seu programa pode funcionar corretamente algumas vezes e não funcionar de jeito nenhum em outras vezes. Estatisticamente, quanto menor um programa maior a chance dele funcionar corretamente, mesmo com um ponteiro errado, pois pouca memória estará sendo usada. Conforme o programa cresce e as falhas se tornam mais freqüentes, você tentará corrigi-las

pensando nas mudanças e acréscimos recentes, e não em problemas com ponteiros. Assim, o mais provável é que você busque o erro onde ele não está.

Um outro problema pode ocorrer num programa que use ponteiros. Ao chamar um **New** durante a execução, a memória pode acabar. Isto causa um erro de tempo de execução, e o programa pára. Para evitar este erro, o Turbo Pascal dispõe da função predefinida **MemAvail. MemAvail** dá o número de bytes (num sistema de 8 bits) ou de parágrafos (em sistemas de 16 bits um parágrafo é composto por 16 bytes) restantes no *heap*. Para corrigir completamente o programa você deve então checar a memória disponível antes de alocá-la. Isto requer que se saiba o número de bytes necessário para cada tipo de dado a ser alocado. Este número pode mudar para cada microprocessador ou sistema operacional (o *Manual de Referência do Turbo Pascal* traz todas estas informações). No caso do programa do exemplo anterior, num IBM PC, é preciso que existam 6 parágrafos livres (96 bytes). A quantidade real de memória requerida pelo registro é de apenas 84 bytes (81 para a seqüência de caracteres, 1 para o caractere e 2 para o inteiro). Mas como a menor alocação possível num sistema de 16 bits é um parágrafo, os 84 bytes devem ser arredondados para cima. O programa correto é o seguinte:

```
program RIGHT; {This program is OK}

type
   pntr = ^object;

   object = record
      x: char;
      y: integer;
      name:string[80];
   end;

var
   p:pntr;
begin
   if MaxAvail>=96 then
   begin  {there is memory available}
      New(p);
      p^.name:='tom';
      p^.x:='g';
      p^.y:=100;
   end;
end.
```

O comportamento errático, num programa, é um sintoma de problemas com ponteiros. O programa funcionará uma vez, e não funcionará em seguida. Outras vezes aparecerão valores incompreensíveis em algumas variáveis. Quando problemas como estes ocorrerem, verifique os ponteiros. Aliás, sempre que começarem a surgir falhas em seus programas, tenha como procedimento padrão a verificação de todos os ponteiros.

Apesar de tudo o que foi dito, os ponteiros são um dos mais úteis e poderosos aspectos da linguagem Pascal, e valem qualquer problema que possam causar. O ideal é aprender a usá-los corretamente desde o princípio.

A REDEFINIÇÃO DE PROCEDIMENTOS E FUNÇÕES PREDEFINIDOS

Apesar de não permitir a redefinição de palavras reservadas, o Turbo Pascal permite que palavras referentes aos procedimentos e funções padrões sejam redefinidas. Alguns programadores às vezes acham esta uma boa idéia, por várias razões. Entretanto, isto só causará problemas. Aqui está um exemplo do tipo de problema que ocorre com a redefinição de procedimentos predefinidos:

```
program ERRADO; {este programa esta errado}
var
   t: integer;

procedure WriteLn(t:integer);
begin
   write(t,' ','bytes de memoria no heap');
end;

begin
   {calcula a quantidade de memoria restante no heap}
   WriteLn(MemAvail);
   WriteLn('Pronto');
end;
```

Neste exemplo, o programador redefiniu o procedimento padrão **WriteLn,** mas esqueceu-se de que a mensagem "Pronto" deveria ser impressa. Assim, pela nova definição de **WriteLn,** a mensagem referente à memória disponível será mostrada duas vezes.

O problema ocorrido no exemplo acima é bastante visível. Erros muito piores podem ocorrer quando um procedimento ou uma função são redefinidos mas não são usados imediatamente. Mais tarde, quando o programa for modificado ou aumentado, o procedimento ou função redefinidos serão usados como se ainda fossem o original. Veja o exemplo:

```
Function MemAvail: boolean;
{verifica se ha espaco na matriz global conta}
var
   t:integer;
begin
   MemAvail:=FALSE;
   for t:=1 to MAX do if conta[t]=0 then MemAvail:=TRUE;
end;
```

O programa funciona perfeitamente. Entretanto, se posteriormente a matriz **conta** for modificada de variável global para variável dinâmica alocada no *heap*, o problema surgirá. Quando você tentar usar **MemAvail** para saber a quantidade de memória restante no *heap*, o programa falhará.

O melhor modo de evitar este tipo de erro é nunca dar a um procedimento ou a uma função, escritos por você, o mesmo nome de um procedimento ou função predefinidos. Em caso de dúvida, adicione suas iniciais ao nome dado (**HSMemAvail**, em vez de **MemAvail**).

ERROS DE SINTAXE INESPERADOS

Ocasionalmente, pode surgir um erro de sintaxe que você não consiga entender ou mesmo reconhecer como erro. Um erro particularmente perturbador ocorrerá se você tentar compilar este programa:

```
program ERROS {este programa nao compilara}
var
   s: string[80];
procedure F1(x: string[80]);
begin
   WriteLn(x);
end;

begin
   ReadLn(s);
   F1(s);
end.
```

O resultado da compilação seria o surgimento da seguinte mensagem de erro na tela:

```
Error 36: Type identifier expected
```

Pressionando a tecla ESC, você encontrará o Turbo Pascal, apontando para a linha abaixo, com o cursor na posição indicada pela seta:

```
procedure F1(x:·string[80]);
                ↑
```

E a mensagem de erro fala de um identificador de tipo esperado! A declaração **string[80]** não fornece este identificador? Há um erro no Turbo Pascal? Não. No Turbo Pascal não se pode usar o tipo **string** numa chamada de procedimento ou função. Uma definição de tipo do usuário deve ser usada. Neste exemplo, você primeiro declara um tipo chamado **str80**, assim:

```
type
   str80 = string[80];
```

Então **str80** é usado como tipo de parâmetro para a função **F1**. O programa, corrigido, seria este:

```
program CORRIGIDO; {Este programa compilara normalmente}

type
   str80 = string[80];

var
   s: string[80];

procedure F1(x: str80);
begin
   WriteLn(x);
end;

begin
   ReadLn(s);
   F1(s);
end.
```

Um outro erro de sintaxe, capaz de causar muita confusão, é o seguinte:

```
program ERRO; {este programa nao compilara}

procedure F2;
var
   t: integer;
begin
   for t:=1 to 10 do WriteLn('Ola''');
end

begin
   F2;
end.
```

Falta o ponto-e-vírgula após o **end** do procedimento **F2.** Porém, o Turbo Pascal apontará para o **begin** seguinte. Aqui, o erro é bastante visível, mas em certas situações você terá de revisar várias linhas antes de encontrar o local onde falta o ponto-e-vírgula.

ERROS IF/THEN, IF/ELSE

Mesmo programadores experientes ocasionalmente caem nas garras de erros **if/then/if/else.** Por exemplo, você é capaz de dizer exatamente o que o fragmento de programa a seguir faz?

```
if conta < 100 then
   if conta > 50 then F1
else F2;
```

Não permita que a tabulação inadequada o engane! O **else** não está associado ao primeiro **if,** mas ao segundo. Um **else** sempre se associará com o **if** mais próximo. No exemplo, quando **conta** for maior que 100 o Turbo Pascal não fará nada. **F2** só será executado se **conta** for menor que 100 e então menor ou igual a 50. É fácil notar isto com o fragmento corretamente formatado:

```
if conta < 100 then
   if conta > 50 then F1
   else F2;
```

Se o objetivo era fazer com que **F2** fosse executado quando **conta** fosse maior que 100, então um bloco **begin/end** deveria ser usado:

```
if conta < 100 then
begin
   if conta > 50 then F1;
end
else F2;
```

O PARÂMETRO VAR EM PROCEDIMENTOS E FUNÇÕES

Muitas vezes, no calor da programação, esquece-se de que se um procedimento ou função muda seu argumento, tal argumento deve ser especificado como um parâmetro **var.** O esquecimento deste detalhe pode gerar resultados bizarros e frustrantes horas de busca do erro. Considere o programa a seguir:

```
program ERRO;  {este programa esta incorreto}
var
   t: integer;

procedure F1(x: integer);
begin
   Write('Digite um numero: ');
   ReadLn(x);
end;

begin
   F1(t);  {pede um valor para t}
   WriteLn('O valor de t e'': ',t);
end;
```

Este programa não funciona. Apenas à variável local x é atribuído um valor, e quando F1 retorna, t não foi modificado. Para que o programa funcione, x deve ser declarado dentro de F1 como um parâmetro **var**. Isto assegura que a variável de chamada t será modificada. O programa ficaria assim:

```
program CORRIGIDO;  {este programa esta correto}
var
   t: integer;

procedure F1(var x: integer);
begin
   Write('Digite um numero: ');
   ReadLn(x);
end;

begin
   F1(t);  {pede um valor para t}
   WriteLn('O valor de t e'': ',t);
end.
```

Parece simples, mas em programas grandes este erro pode ser muito difícil de ser encontrado.

TEORIA GERAL DA DEPURAÇÃO

Cada programador tem sua visão particular sobre programação e depuração, mas algumas técnicas se mostraram melhores que outras. No caso da depuração de programas, testes sucessivos são considerados o método mais rápido e barato, apesar de parecerem tornar o desenvolvimento mais lento.

O método dos testes sucessivos implica apenas sempre ter um programa funcionando. Assim que você tiver um fragmento executável de programa, execute-o, testando toda a seção. Conforme o programa for crescendo, teste as novas seções, bem como o comportamento de todas as seções em conjunto. Este método garante que os erros porventura existentes se concentrem em uma pequena parte do programa.

Os testes sucessivos se baseiam em probabilidades e no conceito de área. A área é um espaço bidimensional. A adição de um comprimento dobra a área. Assim, a área de um programa a ser depurado equivale a n^2. Ao depurar um programa, o ideal é trabalhar sobre a menor área possível de cada vez. Os testes sucessivos permitem que cada área já testada seja subtraída da área total, diminuindo o tamanho da região que pode conter erros.

CONCLUSÃO

Neste livro, vários algoritmos e técnicas foram discutidos, alguns detalhadamente. A ciência da computação é tanto uma ciência teórica como uma ciência empírica. Apesar de ser muito fácil saber qual de dois algoritmos é o melhor, é difícil dizer o que torna bom um determinado programa. Nas áreas de eficiência, portabilidade e depuração, experimentos práticos muitas vezes serão mais produtivos que a reflexão teórica.

Programar é uma ciência e uma arte. É uma ciência, pois é preciso conhecer lógica, e saber como e porque algoritmos funcionam. É uma arte, pois o programador cria sua obra, o programa. Programar computadores é um dos melhores trabalhos existentes; o programador anda sobre a fronteira que separa a ciência da arte, e tem o melhor destes dois mundos à sua disposição.

APÊNDICE **A**

CONVERTENDO C E BASIC PARA TURBO PASCAL

A maior parte dos programadores gasta muito do seu tempo convertendo programas de uma linguagem para outra. Isto é chamado *tradução*. Você pode achar o processo fácil ou difícil, dependendo dos métodos que utilize e do conhecimento que possua das linguagens fonte e destino. Este apêndice apresenta tópicos e técnicas que o ajudarão a converter programas em C e BASIC (inclusive Turbo C e Turbo BASIC) para Turbo Pascal.

Se um programa está escrito em uma linguagem, por que alguém irá querer traduzi-lo para outra? Uma razão é a facilidade de *manutenção*: um programa escrito em uma linguagem não estruturada, como BASIC, é difícil de manter e de ampliar. Outras razões são a velocidade e a eficiência. O Turbo Pascal é uma linguagem muito eficiente e várias tarefas se tornam mais rápidas se traduzidas para esta linguagem. A terceira razão é a *praticidade*: um usuário pode ver um programa útil listado em uma linguagem, mas pode possuir ou utilizar um compilador de outra linguagem. Provavelmente você vai achar que precisa traduzir um programa para Turbo Pascal por uma ou mais dessas razões.

C e BASIC foram escolhidas de uma lista de quase uma centena de linguagens porque são linguagens populares entre usuários de microcomputadores e por representarem extremos opostos do espectro das linguagens de programação. C é uma linguagem estruturada que possui muitas semelhanças com o Turbo Pascal; já o BASIC é uma linguagem não estruturada e não possui qualquer semelhança com o Turbo Pascal. Embora este apêndice não possa apresentar a tradução destas linguagens em todos os detalhes, examinará diversos dos problemas mais importantes. Você já deverá estar familiarizado

tanto com C como com BASIC; este apêndice não lhe ensinará qualquer uma das duas linguagens.

CONVERTENDO C PARA TURBO PASCAL

C e Turbo Pascal possuem muitas semelhanças, especialmente nas suas estruturas de controle e na utilização de sub-rotinas independentes com variáveis locais. Isto possibilita que se façam muitas traduções diretas, ou seja, substituições de uma palavra-chave ou função do Pascal por uma palavra-chave equivalente em C. Com a tradução direta você pode usar o computador para assisti-lo no processo de tradução. Mais adiante, será desenvolvido um programa de tradução simples.

Embora C e Turbo Pascal sejam similares, há grandes diferenças entre eles. Primeiro, C não utiliza uma verificação exaustiva de tipos enquanto o Pascal usa. Portanto, para que haja concordância de todos os tipos de operadores, alguns tipos de rotinas escritas em C têm de ser modificados. Por exemplo, em C, tipos **character** e **integer** podem ser misturados livremente; em Turbo Pascal, isso não pode ser feito sem que se usem algumas funções de conversão de tipo. (Lembre-se, entretanto, de que a checagem forte de tipo – ou a ausência dela – não faz com que uma linguagem se torne necessariamente melhor ou pior. Ela apenas muda a maneira como um programador pensa sobre uma tarefa.)

Uma segunda diferença mais importante é que C não é formalmente *estruturada em blocos*, enquanto Turbo Pascal o é. O termo *estrutura em blocos* refere-se à habilidade de uma linguagem em criar unidades de código logicamente conectadas, que possam ser referidas conjuntamente. O termo significa também que os procedimentos podem ter outros procedimentos encaixados nele, os quais serão conhecidos apenas pelo procedimento externo. Apesar de C permitir que blocos de algoritmos sejam facilmente criados e por isso ser comumente considerada como estruturada em blocos, ela não permite que se definam funções dentro de outras funções. Por exemplo, o seguinte código em C requer duas funções:

```
A( )
{
   printf("iniciando A\n");
   B( );
}
B( )
{
   printf("dentro de B\n");
}
```

Você teria também de certificar-se de que não existem outras funções chamadas de B( ) em qualquer outro lugar do programa. De qualquer modo, quando este programa é traduzido para Turbo Pascal, ele pode ficar assim:

```
procedure A;
var
   x: integer;

   procedure B;
   begin
      WriteLn('dentro do procedimento B');
   end;

begin
   WriteLn('iniciando A');
   B;
end;
```

Aqui, o procedimento **B** é definido dentro do procedimento **A**. Isto significa que o procedimento **B** é conhecido apenas pelo procedimento **A** e pode ser usado apenas pelo procedimento **A**. Fora do procedimento **A**, poderia ser usado um outro procedimento **B** sem conflito.

Uma terceira diferença entre C e Turbo Pascal é que todas as variáveis de C devem ser declaradas antes de serem usadas, mas as referências futuras a funções, porém, não são restringidas e são, na verdade, muito comuns. Entretanto, em Pascal devem ser declaradas todas as variáveis, funções e procedimentos antes de serem usados. Em Pascal padrão, isso quer dizer que você deverá usar um comando **forward** se tiver de se referir a uma função ou procedimento antes dele ter sido declarado.

Uma quarta diferença entre C e Pascal é que em C a compilação separada e a "linkagem" são encorajadas, enquanto o Turbo Pascal não é capaz de fazê-las. Compilação separada é o processo de compilar um programa em pedaços e depois ligá-los com um *linker*. Como foi discutido no Capítulo 5, o Turbo Pascal permite que sub-rotinas externas sejam combinadas com um programa de Pascal, mas não por "linkagem" e compilação separada, que é a maneira utilizada pela maioria dos compiladores de C.

UMA COMPARAÇÃO ENTRE C E TURBO PASCAL

A Figura A-1 compara palavras-chaves de Turbo Pascal com palavras-chaves e operadores de C. Como você pode ver, muitas palavras-chaves de Turbo Pascal não possuem equivalente em C. Isso deve-se em parte ao fato do Turbo Pascal usar palavras-chaves em lugares onde C usa operadores para executar os mesmos passos. Às vezes, Turbo Pascal é apenas mais "prolixo" do que C.

Turbo Pascal	C	Turbo Pascal	C
and	&&	mod	%
array		nil	(sometimes \0)
begin	{	not	!
case	switch	of	
const		or	||
div	/	packed	
do		procedure	
downto		program	
else	else	record	struct
end	}	repeat	do
file		set	
forward	extern (on occasion)	then	
for	for	type	
function		to	
goto	goto	until	while (as in do/while)
if	if	var	
in		while	while
label		with	

Figura A-1 Uma comparação entre as palavras-chaves de Turbo Pascal e C.

Além das palavras-chaves, o Turbo Pascal possui diversos identificadores padrão que podem ser usados diretamente num programa. Estes identificadores podem ser procedimentos e funções (como **WriteLn**) ou variáveis globais (como **MaxInt**) que armazenam informações sobre o estado do sistema. Do mesmo modo, o Turbo Pascal usa identificadores padrão para especificar cada tipo de dado como **real, integer, boolean** e **char.** A Figura A-2 mostra diversos dos identificadores mais comuns de C com os seus equivalentes em Pascal padrão.

C	Turbo Pascal
char or int	boolean
char	byte
char	char
EOF	EOF
0	FALSE
flush (in library)	flush
integer	integer
scanf( ) and others	Read or ReadLn
float	real
any one-zero value	TRUE
printf( )	Write or WriteLn

Figura A-2 Equivalentes de C de alguns identificadores de Pascal.

Além destas, qualquer outra função interna equivalente a C pode ser encontrada na biblioteca padrão de C; entretanto, elas podem variar de um compilador para outro.

C também difere do Turbo Pascal nos seus operadores. A Figura A-3 mostra os operadores de C e seus equivalentes em Turbo Pascal.

C	Turbo Pascal	Significado
+	+	Adição
−	−	Subtração
*	*	Multiplicação
/	/	Divisão
/	div	Divisão Inteira
%	mod	Resto de Divisão Inteira
	^	Exponenciação
=	:=	Atribuição
==	=	Igual
<	<	Menor que
>	>	Maior que
>=	>=	Maior que ou igual
<=	<=	Menor que ou igual
!=	<>	Diferente

Figura A-3 Operadores C e seus equivalentes em Turbo Pascal.

CONVERTENDO LOOPS DE C PARA LOOPS DE TURBO PASCAL

Pelo fato dos *loops* de controle serem fundamentais em muitos programas, esta seção compara os *loops* de C e os de Turbo Pascal. C possui três *loops* internos: **for**, **while** e **do/while.**

O *loop* **for** de C possui a forma geral

for*(inicialização, condição, incremento) comando;*

O **for** de C é um comando muito mais geral que o **for/do** do Turbo Pascal: a condição de teste não precisa ser um valor alvo, como no Turbo Pascal; ao contrário, pode ser qualquer expressão booleana. Também não existe qualquer mecanismo que diga se o *loop* está correndo positiva ou negativamente, pois C não emprega a convenção do Turbo Pascal, **to** e **downto.** Outra diferença é que tanto as seções de inicialização e incremento podem ser compostas – algo que não tem paralelo em Turbo Pascal. Entretanto, apesar dessas diferenças, um *loop* **for** de C terá freqüentemente a forma **for/do** do Pascal padrão, o que faz da tradução uma questão simples. Por exemplo, o *loop* **for** de C

```
for (x = 10; x <= 100;++x) printf("%d\n",x);
```

pode ser traduzido para Pascal como

```
for x:=10 to 100 do WriteLn(x);
```

O **while** de C e o **while/do** do Turbo Pascal são virtualmente o mesmo. Entretanto, o **do/while** de C e o **repeat/until** do Turbo Pascal exigem que você use palavras-chave diferentes e reverta o teste de *loop*. Isso se deve ao fato de que o **do/while** de C continua rodando *enquanto* a condição é verdadeira, já o **until** do Turbo Pascal roda *até* que algo se *torne* verdade. Aqui está uma amostra da tradução de ambos os tipos de *loops*:

```
C                                   Turbo Pascal

while(x < 5)                        while x < 5 do
{                                   begin
   printf("%d\n",x);                   WriteLn(x);
   x = getnum();                       Read(x);
}                                   end;

do  {                               repeat
    x = getnum();                      Read(x);
    printf("%d\n",x);                  WriteLn(x);
} while (x <= 5);                   until x > 5;
```

Seja cuidadoso na tradução do **do/while** em **repeat/until**: você deve inverter o sentido da condição de teste. (Tem havido alguma discussão filosófica em relação ao **do/while** *versus* **repeat/until**. O **do/while** é considerado positivo pois roda enquanto a condição for **TRUE**; o **repeat/until** foi chamado de comando negativo, pois o *loop* é executado desde que a condição seja **FALSE**. Sugeriu-se que a escolha do *loop* mais fácil de usar depende de você ser otimista ou pessimista – mas isto ainda não foi provado.)

UMA TRADUÇÃO AMOSTRA

Para ter uma idéia do processo de tradução, siga os passos da conversão de um programa em C para Turbo Pascal. A seguir, há um programa simples em C.

```
float qwerty;

main( )
{
    qwerty = 0;
    printf("%f",qwerty);
    printf("oi!\n");
    tow(25);
    printf("%f\n",Ken(10));
    printf("%2.4f\n",qwerty);
}

tow(x)
int x;
{
    printf("%d",x * 2)
}

float Ken(w)
float w;
{
    qwerty = 23.34;
    return w/3.1415;
}
```

Este programa em C possui três funções declaradas. (Tenha em mente que todas as sub-rotinas de C são funções, quer o valor de retorno seja usado ou não.) O primeiro passo na tradução deste programa para Turbo Pascal é determinar quais funções de C deverão se tornar funções de Turbo Pascal (ou seja, aquelas que retornarão um valor), e quais serão simplesmente procedimentos.

Você pode determinar isso olhando para a palavra-chave **return** de C. Se ela estiver presente em uma função, você poderá esperar que será retornado um valor.

(Tecnicamente, este nem sempre é o caso, mas é suficiente para este exemplo.) A única função que usa um **return** é **ken.** Portanto, o equivalente em Turbo Pascal de **ken** é

```
function ken(w : real) : real;
begin
   qwerty := 23.34;
   ken := w / 3.1415;
end;
```

A função **tom** de C não retorna um valor, por isso, em Turbo Pascal, torna-se um procedimento.

```
procedure tom(x : integer);
begin
   writeln(x * 2);
end;
```

Em seguida a função **main** tem de ser convertida no algoritmo do programa para a versão em Turbo Pascal mostrada abaixo:

```
begin
   qwerty := 0;
   writeln(qwerty);
   writeln('oi!');
   tom(25);
   writeln(ken(10));
   writeln(qwerty:2:4);
end.
```

Finalmente, você deverá declarar a variável global **qwerty** como **real,** e acrescentar o cabeçalho do programa. Quando você faz isso e junta as partes, a tradução do programa de C para o Turbo Pascal fica assim:

```
program teste(input,output);

var qwerty : real;

procedure tom(x : integer);
begin
   writeln(x * 2);
end;

function ken(w : real) : real;
begin
   qwerty := 23.34;
   ken := w / 3.1415;
end;
```

```
begin
   qwerty := 0;
   writeln(qwerty);
   writeln('oi!');
   tom(25);
   writeln(ken(10));
   writeln(qwerty:2:4);
end.
```

USANDO O COMPUTADOR PARA AJUDÁ-LO A CONVERTER C EM TURBO PASCAL

É possível construir um programa de computador que aceite programas fonte em uma linguagem e o retorna na saída em uma outra linguagem. A melhor maneira de se fazer isso é pela implementação de um analisador (*parser*) de linguagem completo para a linguagem fonte – mas em vez de gerar um programa-objeto, ele sairá com a linguagem destino. Ocasionalmente você poderá encontrar anúncios de tais produtos em revistas de computadores, e seus altos preços refletem a complexidade da tarefa.

Uma abordagem menos ambiciosa é construir um programa simples para ajudá-lo nos seus esforços de conversão de programas através da execução de algumas das tarefas mais simples de tradução. Esta "assistência computadorizada" pode facilitar muito os trabalhos de conversão.

Um tradutor assistido por computador aceita como entrada um programa em linguagem fonte e automaticamente executa todas as traduções diretas para a linguagem destino, deixando as conversões mais difíceis para você. Em C, por exemplo para atribuir a **conta** o valor 10, você escreveria

```
count=10;
```

Em Turbo Pascal, o comando é o mesmo, exceto por haver dois pontos próximos ao sinal =. Portanto, o tradutor auxiliado por computador pode trocar o sinal da declaração de atribuição de C (=) por := em Turbo Pascal. Outro exemplo é o *loop* de C **while**: a palavra-chave **while** é usada do mesmo modo no Turbo Pascal.

Entretanto, C e Turbo Pascal acessam arquivos em disco de maneiras diferentes, e não há um modo fácil de se realizar tal conversão automaticamente. Além disso, a conversão do **do/while** de C para o **repeat/until** do Turbo Pascal também não pode ser facilmente automatizada. Daí essas traduções complicadas são deixadas para você.

Aqui estão os passos para se criar um tradutor de C para Turbo Pascal. Em primeiro lugar, um tradutor de C para Turbo Pascal precisa de uma função que retorne um sinal de cada vez do programa em C. Como é mostrado aqui, pode-se para isso modificar a função **Pega_Simb** desenvolvida no Capítulo 9:

```
procedure Pega_Simb;
var
   temp:str80;

begin
     simbolo:='';  {sequencia de caracteres nula}
     while(Branco(prog[t])) do t:=t+1; {retira os espacos antes da expressao}
     if prog[t]='$' then simbolo:='$';
     if (pos(prog[t],'.#:;,+!()&~-*/!%^=(){}')()>0)
     or (prog[t]=chr(39)) then
     begin
        if (prog[t]='{') or (prog[t]='}') then
           begin
              Tipo_Simb:=NOME;
              if prog[t]='{' then
                 begin
                    simbolo := 'begin';
                    contador := contador + 1;
                 end else
                 begin
                    simbolo := 'end';
                    contador := contador - 1;
                 end;
           end
           else
           begin
              Tipo_Simb := DELIMITADOR;
              simbolo := prog[t]; { e' um operador }
           end;
           t := t + 1
     end
     else
     if Letra(prog[t]) then
     begin
        while (not Delim(prog[t])) do
          begin
             simbolo := concat(simbolo,prog[t]); {constroi simbolo}
             t := t + 1;
          end;
          Tipo_Simb := NOME;
     end
     else
     if Digito(prog[t]) then
     begin
        while (not Delim(prog[t])) do
          begin
             simbolo := concat(simbolo,prog[t]); {constroi numero}
             t := t + 1;
             Tipo_Simb := NUMERO;
          end;
```

```
      end
      else
      if prog[t] = '"' then {e' string}
      begin
         t := t + 1;
         simbolo := chr(39);  {uma '}
         while prog[t] <> '"' do
           begin
              simbolo := concat(simbolo,prog[t]);
              t := t + 1;
              Tipo_Simb := SEQ;
           end;
          t := t + 1; {}
          simbolo := concat(simbolo,chr(39));
      end;
end;   {Pega_Simb}
```

Dentro do **Pega_Simb** os símbolos { e } de C são transformados em **begin** e **end**, seus equivalentes em Pascal, para mais tarde simplificar outras áreas do programa.

O segundo passo na criação do tradutor é prover uma rotina que traduza elementos da linguagem C para suas contrapartidas em Turbo Pascal. O procedimento **traduz** mostrado aqui não é a melhor maneira de codificar tal rotina, mas é suficiente para os propósitos do tradutor.

```
procedure Traduz; {traduz de C para Pascal}
begin
   case simbolo[1] of
      '~': simbolo := 'not';
      '=': {ve o proximo para verificar se e' ==}
           begin
              Pega_Simb;
              if simbolo = '=' then simbolo := '=='
              else
              begin
                 Devolve;  {devolve simbolo}
                 simbolo := ':=';  {sinal de atribuicao}
              end;
           end;
      '!': {ve o proximo para verificar se e' <> ou NOT}
           begin
              Pega_Simb;
              if simbolo = '=' then simbolo := '<>'
              else
              begin
                 Devolve;  {devolve simbolo}
                 simbolo := 'not';  {sinal de atribuicao}
              end;
           end;
      '%': simbolo := 'mod';
      '|': begin
              Pega_Simb;
              if simbolo <> '|' then Devolve;  {nao e' um ou dobrado}
              simbolo := 'or';
```

```
            end;
        '&': begin
                Pega_Simb;
                if simbolo <> '&' then Devolve;  {nao e' um e dobrado}
                simbolo := 'and';
             end;
        '~': simbolo := 'xor';
    end;

    {agora verifica as palavras reservadas}
    if simbolo = 'switch' then simbolo := 'case'
    else if simbolo = 'struct' then simbolo := 'record'
    else if simbolo = 'int' then simbolo := 'integer'
    else if simbolo = 'float' then simbolo := 'real'
    else if simbolo = 'printf' then simbolo := 'write'
    else if simbolo = 'extern' then simbolo := 'foward'
    else if simbolo = 'case' then simbolo := ' ';
end;  {Traduz}
```

Note que em muitos casos, palavras-chave e operadores de C são simplesmente saltados, pois são os mesmos que no Turbo Pascal. Entretanto, no caso dos *loops* **while** e **do/while** de C, é impossível com esse método saber se você precisa de um *loop* **while/do** ou um **repeat/until.**

Aqui está o programa de tradução completo:

```
  program C_para_Pascal;

type
    str80=string[80];
    Tipo=(DELIMITADOR, NOME, NUMERO, SEQ);
var
   nome_ent,nome_sai,simbolo,prog: string[255];
   Tipo_Simb: Tipo;
   arq_ent,arq_sai: text;
   contador,t: integer;

function Letra(car:char):boolean;
{responde TRUE se car e' uma letra do alfabeto}
begin
     Letra:=(UpCase(car)>='A') and (UpCase(car)<='Z');
end;   {Letra}

function Branco(car:char):boolean;
{TRUE se newline,espaco ou tab}
begin
     Branco:=(car=' ') or (car=chr(9)) or (car=chr(13));
end;   {Branco}

function Delim(car:char):boolean;
{TRUE se for um delimitador}
begin
     if car= chr(39) then Delim := TRUE   {uma '}
```

```
     else if pos(car,' #:.,;()|&~+-/¥%^=!()$[]') <> 0 then Delim:=TRUE
     else Delim:=FALSE;
end;  {Delim}

function Digito(car:char) :boolean;
{TRUE se for um digito entre 0 e 9}

begin
     Digito:=(car>='0') and (car<='9');
end;   {Digito}

procedure Pega_Simb;
var
   temp:str80;

begin
     simbolo:='';  {sequencia de caracteres nula}
     while(Branco(prog[t])) do t:=t+1; {retira os espacos antes da expressao}
     if prog[t]='$' then simbolo:='$';
     if (pos(prog[t],'.#:;,+!()&~-¥/!%^=(){}')<>0)
     or (prog[t]=chr(39)) then
     begin
        if (prog[t]='{') or (prog[t]='}') then
           begin
              Tipo_Simb:=NOME;
              if prog[t]='{' then
                 begin
                    simbolo := 'begin';
                    contador := contador + 1;
                 end else
                 begin
                    simbolo := 'end';
                    contador := contador - 1;
                 end;
           end
           else
           begin
              Tipo_Simb := DELIMITADOR;

              simbolo := prog[t]; { e' um operador }
           end;
           t := t + 1
     end
     else
     if Letra(prog[t]) then
     begin
        while (not Delim(prog[t])) do
          begin
             simbolo := concat(simbolo,prog[t]) ; {constroi simbolo}
             t := t + 1;
          end;
          Tipo_Simb := NOME;
     end
     else
     if Digito(prog[t]) then
     begin
        while (not Delim(prog[t])) do
```

```
          begin
             simbolo := concat(simbolo,prog[t]); {constroi numero}
             t := t + 1;
             Tipo_Simb := NUMERO;
          end;
     end
     else
     if prog[t] = '''' then {e' string}
     begin
        t := t + 1;
        simbolo := chr(39);  {uma '}
        while prog[t] <> '''' do
          begin
             simbolo := concat(simbolo,prog[t]);
             t := t + 1;
             Tipo_Simb := SEQ;
          end;
         t := t + 1; {}
         simbolo := concat(simbolo,chr(39));
     end;
end;   {Pega_Simb}

procedure Devolve; {devolve simbolo nao usado}
begin
   t := t - length(simbolo);
end;   {Devolve}

procedure Traduz; {traduz de C para Pascal}
begin
   case simbolo[1] of
      '~': simbolo := 'not';
      '=': {ve o proximo para verificar se e' ==}
           begin
              Pega_Simb;
              if simbolo = '=' then simbolo := '=='
              else
              begin
                 Devolve;  {devolve simbolo}
                 simbolo := ':=';  {sinal de atribuicao}
              end;
           end;
      '!': {ve o proximo para verificar se e' <> ou NOT}
           begin
              Pega_Simb;
              if simbolo = '=' then simbolo := '<>'
              else
              begin
                 Devolve;  {devolve simbolo}
                 simbolo := 'not';  {sinal de atribuicao}
              end;
           end;
      '%': simbolo := 'mod';
      '|': begin
              Pega_Simb;
              if simbolo <> '|' then Devolve;  {nao e' um ou dobrado}
              simbolo := 'or';
```

```
            end;
        '&': begin
                Pega_Simb;
                if simbolo <> '&' then Devolve;  {nao e' um e dobrado}
                simbolo := 'and';
             end;
        '~': simbolo := 'xor';
    end;

    {agora verifica as palavras reservadas}
    if simbolo = 'switch' then simbolo := 'case'
    else if simbolo = 'struct' then simbolo := 'record'
    else if simbolo = 'int' then simbolo := 'integer'
    else if simbolo = 'float' then simbolo := 'real'
    else if simbolo = 'printf' then simbolo := 'write'
    else if simbolo = 'extern' then simbolo := 'foward'
    else if simbolo = 'case' then simbolo := ' ';
end;  {Traduz}

procedure Converte;
var
    conta: integer;
begin
    Pega_Simb;
    for conta := 1 to contador do
        Write(arq_sai,'    ');
    while simbolo <> '$' do
    begin
        case Tipo_Simb of
            SEQ: write(arq_sai,simbolo);
            NOME: begin
                    Traduz;
                    write(arq_sai,simbolo,' ');
                  end;
            DELIMITADOR: begin
                           Traduz;
                           write(arq_sai,simbolo);
                         end;
            NUMERO: write(arq_sai,simbolo);
        end;
        Pega_Simb;
    end;
    simbolo := ' ';
    writeln(arq_sai,simbolo);
end;   {Converte}

begin   {main}
    write('Arquivo de entrada: ');
    readln(nome_ent);
    write('Arquivo de saida: ');
    readln(nome_sai);
    assign(arq_ent,nome_ent);
    assign(arq_sai,nome_sai);
    reset(arq_ent);
    rewrite(arq_sai);

    contador := 0;  {contador para cada BEGIN e END}
    while not EOF(arq_ent) do
```

```
   begin
      t := 1;  {inicializa indice}
      readln(arq_ent,prog);
      prog := concat(prog,'s');
      Converte;
   end;
   simbolo := '.';
   writeln(arq_sai,simbolo);
   close(arq_ent);
   close(arq_sai);
end.
```

Neste programa, a variável global **contador** é usada para tabular o programa automaticamente, colocando três espaços para cada **begin** e remover três espaços para cada **end.** Isso permite que a saída em pseudo-Pascal já saia adequadamente formatada.

O programa de assistente de conversão de C para Turbo Pascal basicamente lê uma linha do algoritmo fonte em C, tira dele um sinal de cada vez, executa tantas conversões quanto possível e escreve uma versão em Turbo Pascal. Para verificar como este programa simples pode facilitar traduções de C para Turbo Pascal, passe este programa em C pelo tradutor:

```
main()
{
   int t,a;

   t=getnum();
   if(t=10) then process(t);
   else {
      a=t-100;
      print("%d",a);
   }
}

process(x)
int x;
{
   int count;

   for(count=0;count;count++)
      printf("this is x*%d: %d\n",count,x*count);

}
```

Depois de você tê-lo rodado pelo programa tradutor, a saída em pseudo-Pascal será a seguinte:

```
main ()
   begin
   integer t ,a ;
```

```
    t :=getnum ();
    if (t :=10)then process (t );
    else begin
       a :=t -100;
       print ('%d',a );
    end
end

process (x )
integer x ;
    begin
    integer count ;

    for (count :=0;count ;count ++)
    Write ('this is x*%d: %d\n',count ,x *count );

end.
```

Este, como você pode ver, não é o programa em Turbo Pascal, mas você economizou muita digitação. Em seguida, tudo o que você tem a fazer é editar uma linha de cada vez para corrigir as diferenças.

CONVERTENDO BASIC PARA TURBO PASCAL

A tarefa de converter BASIC para Turbo Pascal é muito mais difícil do que a de converter C para Turbo Pascal. O BASIC não é uma linguagem estruturada e traz pouca semelhança com o Turbo Pascal. Isto quer dizer que ele não possui uma série completa de estruturas de controle e, o mais importante, não possui sub-rotinas independentes com variáveis locais. A tarefa de tradução é muito sutil: geralmente requer conhecimentos extensivos tanto de BASIC quanto de Turbo Pascal, e uma compreensão do programa, pois em essência você estará apenas usando a versão em BASIC como guia para reescrever o programa, agora em Turbo Pascal. Por causa da complexidade da tarefa, esta seção olha para algumas das traduções mais problemáticas e oferece sugestões.

CONVERTENDO LOOPS DE BASIC PARA LOOPS DE TURBO PASCAL

O *loop* **for/next** é a única forma de *loop* de controle em muitas versões de BASIC. A forma geral do *loop* **for/next** do BASIC é geralmente similar ao *loop* **for/do** do Turbo Pascal: há uma inicialização e um valor alvo mas, ao contrário da opção **STEP** no BASIC, os únicos incrementos possíveis no Turbo Pascal são 1 e –1. O *loop* **for/do** do Turbo

Pascal é muito mais sofisticado e flexível do que o *loop* **for/next** do BASIC, pois permite que se use como controle qualquer tipo escalar. Você deve recodificar todos os *loops* **for/next** que usem a opção **step** em *loops* do Pascal como **while/do** ou **repeat/until.** Entretanto, para esta discussão, o exemplo usa apenas *loops* de BASIC que não empregam a opção **STEP**. Por exemplo, o *loop* **for/next** do BASIC

```
10 FOR X=1 TO 100
20 PRINT X
30 NEXT
```

é traduzido para Pascal como

```
for x:=1 to 100 do WriteLn(x);
```

Como você pode ver, a substituição é executada uma-a-uma. O verdadeiro segredo na conversão do *loop* **for/next** é assegurar que a variável de controle do *loop* não seja modificada dentro do *loop*. Muitas formas de BASIC permitem que a variável de controle seja alterada por comando dentro do *loop*, assim:

```
10 FOR CONTR=10 TO 0 STEP -1
20 INPUT R
30 PRINT R*CONTR
40 IF R=100 THEN CONTR=0
50 NEXT
```

O comando **IF/THEN** da linha 40 poderia fazer com que o *loop* terminasse antes. Para traduzir isto adequadamente para o algoritmo do Turbo Pascal, você deverá prever esta contingência. Embora algumas outras implementações do Pascal possam permiti-lo, o Turbo Pascal não, pois após o comando **for/do** ter sido compilado, o número de vezes que o *loop* efetivamente vai se repetir já está fixado – mesmo que você mudasse o valor de controle do *loop* no corpo do próprio *loop*. Portanto, você deve codificar *loops* **for/next** do BASIC desse tipo em um *loop* **while/do** ou em um **repeat/until** do Turbo Pascal.

Algumas formas de BASIC possuem disponível um *loop* **WHILE/WEND.** Neste caso, você usaria um loop **while/do** do Turbo Pascal e sua tradução seria simples. Se o BASIC que você está usando não possui o *loop* **WHILE/WEND** ou se você preferir não usá-lo, seu trabalho será mais difícil, pois você terá de reconhecer um *loop* construído através de comandos **GOTO.** Este também será o caso se um *loop* **repeat/until** tiver sido construído em BASIC. Estes tipos de traduções tornam-se pesadelos pois você precisa realmente entender de que modo trabalha o algoritmo para reconhecer o *loop* e traduzi-lo para uma das estruturas de controle de *loop* internas do Turbo Pascal.

Depois de você ter achado um *loop* construído em BASIC, há uma maneira fácil de dizer se o *loop* deve ser traduzido para um **while/do** ou um **repeat/until** do Turbo Pascal. Lembre-se de que um *loop* **repeat/until** sempre é executado pelo menos uma vez pois a condição do *loop* é checada no fim do *loop*; enquanto que um *loop* **while/do** pode ser ou não executado pois sua condição é checada no topo. Portanto, você deve observar cuidadosamente cada *loop* construído em BASIC para determinar onde é aplicado o teste do *loop*. Por exemplo, o algoritmo em BASIC

```
100 S=S+1
200 Q=S/3.1415
300 PRINT Q
400 IF S < 100 THEN GOTO 100
```

é na verdade um *loop* **repeat/until** disfarçado – ele sempre será executado pelo menos uma vez. Depois de ter sido executada a linha 100, as linhas de 200 a 400 também serão. Se S é menor que 100, o programa voltará para a linha 100. Em Turbo Pascal este trecho seria:

```
repeat
   s:=s+1;
   q:=s/3.1415;
   Write(q);
until s=100;
```

No seguinte exemplo em BASIC, o teste do *loop* é executado no começo:

```
10 A=1
20 IF A > 100 THEN GOTO 80
30 PRINT A
40 INPUT B
50 A=A+B
60 GOTO 20
80 PRINT "FEITO"
```

Isto exige o uso do *loop* **while/do** no equivalente em Turbo Pascal:

```
a:=1;
while a <= 100 do
begin
   WriteLn(a);
   ReadLn(b);
   a:=a+b;
end;
```

Evite colocar por acidente qualquer inicialização dentro do próprio *loop*. Neste exemplo, o comando **a:=1** tem de estar fora do *loop* pois é uma condição de início e não pertence ao *loop* propriamente dito.

CONVERTENDO O IF/THEN/ELSE

A maioria das formas de BASIC possui apenas o comando de linha única **IF/THEN/ELSE**. Isto significa que quando um bloco de comandos deve ser executado na saída de um **if**, tem de ser usado o **goto** ou o **gosub**. Você deve reconhecer esta situação porque quando você for traduzi-lo, vai querer estruturar o código em um comando **if/then** ou **if/then/else** adequado do Turbo Pascal. Como exemplo, considere o fragmento deste algoritmo em BASIC:

```
120 IF T<500 THEN GOTO 500
130 Y=W
140 T=10
150 INPUT A$
.
.
.
500 REM REINICIA LEITURA DO DISCO
```

Para implementar um bloco **if** em um programa em BASIC, a condição **IF** deve ser negativa: o objetivo do **if** não deve ser a condição que provoca a entrada no bloco **if**, mas ao contrário, a que causa um salto por cima dele. Este é um dos piores problemas em BASIC. O uso de rotinas **GOSUB** como alvo de **IF** ou **ELSE** simplifica ligeiramente o problema, mas não completamente. Se o fragmento em BASIC fosse traduzido diretamente para Turbo Pascal ele ficaria assim:

```
if t<500 then {nao faz nada}
else
begin
   y:=w ;
   t:=10;
   ReadLn(a);
end;
{reinicia leitura do disco}
```

Agora você pode ver o problema: o alvo do **if** na realidade é um comando vazio. O único modo de resolver isto é refazendo a condição **if** de modo que se ela for verdadeira, o bloco seja executado. O fragmento do algoritmo então ficará:

```
if t>=500 then
begin
   y:=w ;
   t:=10;
   ReadLn(a);
end;
{reinicia leitura do disco}
```

Agora o algoritmo, do modo como está escrito em Turbo Pascal, faz sentido.

A diferença entre o modo como o **if/then** é usado no BASIC e no Turbo Pascal ilustra que freqüentemente a linguagem de programação direciona a abordagem para a solução do problema. A maioria das pessoas acha a forma positiva do **if** mais natural que a negativa.

CRIANDO SUBPROGRAMAS EM PASCAL A PARTIR DE PROGRAMAS EM BASIC

Um motivo que dificulta a tradução de BASIC para Turbo Pascal é o fato de que o BASIC não sustenta sub-rotinas independentes com variáveis locais. Isto significa que uma tradução literal de um programa de BASIC para Turbo Pascal produz um programa principal muito grande, sem subprogramas. Isto, em primeiro lugar, contraria muitas das razões que você deve ter considerado para querer traduzir o programa: manutenção, estrutura e flexibilidade.

Uma tradução melhor criaria um programa em Turbo Pascal com um programa principal bem menor e muitos outros subprogramas, mas para fazer isso, requer-se conhecimento do programa e um olho vivo para a leitura de algoritmos. Entretanto, aqui vão algumas dicas para auxiliá-lo.

Primeiro, você deve transformar todas as sub-rotinas em subprogramas. Além disso, procure pelas funções similares nas quais tenham sido mudadas apenas as variáveis, e coloque-as em um subprograma com parâmetros. Por exemplo, este programa em BASIC possui duas sub-rotinas – a primeira na linha 100 e a segunda na 200.

```
10 A=10
20 B=20
30 GOSUB 100
40 PRINT A,B
50 C=20
60 D=30
70 GOSUB 200
80 PRINT C,D
90 END
100 A=A*B
110 B=A/B
120 RETURN
200 C=C*D
210 D=C/D
220 RETURN
```

Ambas as sub-rotinas fazem exatamente a mesma coisa, exceto por operarem em diferentes variáveis. Uma tradução correta deste programa para Turbo Pascal possui apenas um procedimento, que usa parâmetros para evitar que sejam empregadas duas funções:

```
program x;

var
   a,b,c,d: real;

procedure f1(var x,y: real);
begin
   x:=x*y;
   y:=x/y;
end;

begin
   a:=10;  b:=20;
   f1(a,b);
   WriteLn(a,' ',b);

   c:=20;  d:=30;
   f1(c,d);
   WriteLn(c,' ',d);
end.
```

Esta tradução para Turbo Pascal aproxima o sentido do algoritmo para o leitor mais proximamente do que a versão em BASIC, uma vez que BASIC implica que haja realmente duas funções separadas.

A segunda regra é que você deve colocar todo o trecho repetido em uma função. Em um programa em BASIC algumas linhas iguais podem ser repetidas. Um programador ocasionalmente faz isso para tornar o algoritmo um pouco mais rápido. Por ser uma linguagem compilada, o uso de funções ao invés de programas principais longos em Turbo Pascal tem muito pouco efeito negativo na velocidade de execução, e o aumento de clareza e estrutura compensam qualquer ganho em velocidade.

LIVRANDO-SE DAS VARIÁVEIS GLOBAIS

Em BASIC, todas as variáveis são globais: elas são conhecidas por todo o programa e podem ser modificadas em qualquer lugar dele. No processo de tradução tente converter tantas dessas variáveis globais quanto possível em variáveis locais, pois isso torna o programa mais elástico e livre de falhas. Quanto mais variáveis globais haja, tanto é mais provável que ocorram efeitos colaterais.

Às vezes é difícil saber quando fazer uma variável local a um subprograma. As escolhas mais fáceis são as que controlam contadores em seções curtas do algoritmo. Por exemplo, neste programa

```
10 FOR X=1 TO 100
20 PRINT X
30 NEXT
```

X está sendo usado claramente para controlar o *loop* **FOR/NEXT** e pode, por isso, ser colocado em uma variável local dentro de um subprograma.

Um outro tipo de variável candidata a tornar-se local é a variável temporária. Uma variável temporária guarda o resultado intermediário de um cálculo. Variáveis temporárias freqüentemente são espalhadas em um programa, e podem ser difíceis de reconhecer. Por exemplo, a variável C12 mostrada aqui guarda um resultado temporário no cálculo.

```
10 INPUT A,B
20 GOSUB 100
30 PRINT C12
40 END
100 C12A*B
110 C12=C12/0.142
120 RETURN
```

O mesmo programa em Turbo Pascal, com C12 como variável local, seria:

```
program x;
var
   a,b: real;

function f2(x,y: real): real;
var
   C12: real;
begin
   C12:=a*b;
   C12:=C12/0.142;
   f2:=C12;
end;

begin
   ReadLn(a,b);
   WriteLn(f2(a,b));
end.
```

Lembre-se de que quanto menos variáveis globais, melhor. Portanto é importante encontrar boas candidatas para variáveis locais.

PENSAMENTOS FINAIS SOBRE TRADUÇÃO

Embora a tradução de programas possa ser a mais tediosa de todas as tarefas de programação, também é uma das mais comuns. Uma boa abordagem é primeiro entender o modo como funciona o programa que você está traduzindo. Uma vez sabendo como ele opera, será fácil reescrevê-lo e você saberá se sua nova versão está funcionando corretamente. Além disso, quando você conhece o programa que está traduzindo, o trabalho torna-se mais interessante pois não se tratará simplesmente de um processo de substituição de sinais.

APÊNDICE **B**

DIFERENÇAS ENTRE AS VERSÕES 3.0 E 4.0

AMBIENTE DE TRABALHO

Houve mudança completa na apresentação do menu, que agora é do tipo "pull-down menu". Desta maneira, ao selecionar um item cmo "file" (arquivo) aparecem os subitens do menu numa janela. Para selecionar um subitem, é só deslocar o cursor com as setas (o fundo da opção selecionada fica reverso) ou digitar seu primeiro caractere.

OPÇÕES DO MENU PRINCIPAL

FILE (arquivo)

- **LOAD** (carregar) – Lê um arquivo do disco para o Editor. Se o arquivo não existir, será criado.
- **PICK** (pegar) – Permite a escolha de um arquivo entre um dos oito últimos arquivos editados recentemente.
- **NEW** (novo) – Limpa o editor e deixa o nome do arquivo como NONAME.PAS.
- **SAVE** (salvar) – Grava o arquivo da memória para o disco. Se o arquivo tiver o nome de NONAME.PAS, será perguntado um novo nome.

. **WRITE TO** (escrever em) – Escreva o conteúdo do Editor num arquivo indicado pelo usuário.

. **DIRECTORY** (diretório) – Mostra o diretório, de acordo com uma máscara especificada.

. **CHANGE DIR** (mudar de diretório) – Permite mudar o subdiretório atual.

. **OS SHELL** (ambiente do Sistema Operacional) – Sai do Turbo Pascal e volta temporariamente para o Sistema Operacional, onde pode-se executar qualquer comando ou outros programas. Para voltar ao Turbo, é só digitar EXIT.

. **QUIT** (sair) – Sai do Turbo Pascal, voltando para o Sistema Operacional.

EDIT (editar)

Semelhante à versão anterior. Permite a edição de programas. Para voltar ao menu principal, tecle ^KD ou F10.

RUN (correr, executar)

Compila, linca e executa seu programa.

COMPILE (compilar)

. **COMPILE** (compilar) – Compila e linca o programa-fonte.

. **MAKE** (fazer) – Cria um arquivo executável tipo .EXE. Ao compilar seu programa, o Turbo checará se alguma unidade (unit) foi alterada desde a última compilação. Se foi alterada, será recompilada.

. **BUILD** (construir) – Funciona de maneira semelhante à opção MAKE, porém forçando a recompilação das unidades.

. **DESTINATION** (destinação) – Permite selecionar a destinação do arquivo compilado entre memória e disco. Para escolher a opção, digite inicial (D) ou **RETURN.**

. **FIND ERROR** (achar erro) – Pode-se achar um erro dando o endereço na forma XXXX:XXXX.

. **PRIMARY FILE** (arquivo primário ou principal) – O mesmo que o "Main File" na versão 3.0. O Turbo sempre começa a compilação por este arquivo.

. **GET INFO** (recuperar informação) – Mostra informações a respeito do arquivo que está sendo trabalhado.

OPTIONS

. **COMPILER** (compilador) – Permite a inserção de diretivas de compilação sem escrevê-las no programa-fonte. É também usada para selecionar necessidades de memória para seu programa.

. **ENVIRONMENT** (ambiente) – Permite a seleção das opções de "Auto Save Edit" (gravação automática do arquivo editado) e "Backup Source File", (cópia de segurança do arquivo-fonte). A opção "Environment" permite a seleção da quantidade de linhas para tela (25, 43 ou 50), janelas para zoom e "toggles" para reter ou não a tela.

. **DIRECTORIES** (diretórios) – Mostra ao Turbo em qual diretório devem ser procurados os arquivos (programas-fontes, arquivos de inclusão, unidades e arquivos-objetos.

. **PARAMETERS** (parâmetros) – Permite a especificação de parâmetros para quando o programa for compilado em memória.

. **LOAD OPTIONS** (opções para carregamento) – Carrega um arquivo de configuração do Turbo Pascal.

. **SAVE OPTIONS** (opções para salvar) – Grava a configuração atual do Turbo Pascal para um arquivo em disco.

UNITS

São unidades de código que contêm procedimentos e funções e são compiladas separadamente do programa principal. A vantagem de se usar as "Units" é que não é necessário recompilar o programa inteiro após uma pequena alteração, somente recompila-se a "Unit" afetada.

Existe também a possibilidade de se compilar, testar e debugar uma unidade separadamente do programa principal. Uma vez testada e compilada, a unidade não

precisará mais ser compilada. Em vez disso, a unidade será lincada com o programa que a usar. Lincar uma unidade é mais rápido que compilá-la.

As unidades são exatamente como programas e possuem o seguinte formato:

. linha com o nome da unidade, que deve ser o mesmo nome do arquivo.

. uma seção de interface, área que se conecta com outros programas ou unidades, contendo o cabeçalho dos procedimentos e funções, variáveis globais e o uso de outras unidades.

. seção de implementação, contendo o corpo dos procedimentos e funções anunciados na seção de interface e as variáveis globais de uso interno pela unidade.

Ao necessitar de algum procedimento ou função contido numa unidade, chame-o com o comando **Uses.**

UNITS COMPILADAS

As unidades podem ser compiladas da mesma maneira que os programas. Uma vez compiladas, serão armazenadas pelo Turbo num arquivo de mesmo nome que a unidade, porém com terminação .TPU.

ESTRUTURA DOS PROGRAMAS

Existem algumas diferenças, entre elas:

– Não existem mais "Overlays" ou troca de programas com "Chain".

– Os programas podem usar toda memória (até mais de 640 K).

– Existência de "UNITS" (ver acima).

PALAVRAS RESERVADAS DO TURBO PASCAL 4.0

Implementation

Declara o começo da seção de implementação numa Unit. A seção de implementação contém o corpo das rotinas declaradas na seção de interface. Só é necessário indicar o nome da rotina e seu corpo, e não seus parâmetros (indicados sempre na seção de interface).

Interface

Declara o começo da seção de interface numa Unit. A seção de interface contém o cabeçalho das rotinas (procedimentos e funções) e seus parâmetros, que serão "visíveis" a outros programas e unidades que utilizem a unidade atual.

Interrupt

Declara um procedimento como um procedimento de interrupção.
Ex.: Procedure handler (Flags, CS, IP, AX, BX, CX, DX, SI,
DI, DS, ES, BP : word); Interrupt;

Unit

Declara que o arquivo é uma Unit.

Uses

Declara quais unidades serão acessadas pelo programa ou unidade.

DIRETIVAS DE COMPILAÇÃO DO TURBO 4.0

DIRETIVA DE COMPILAÇÃO B
Short-circuit Boolean Evaluation (Default é B–)
(Avaliação Booleana Rápida)

Reduz o tempo que o computador leva para determinar se uma expressão é verdadeira ou falsa.

DIRETIVA DE COMPILAÇÃO D
Debug Information (Default é D+)
(Informação para Debug)

O Turbo Pascal gera um arquivo que relaciona o programa-fonte com o programa-objeto quando esta diretiva está ativa. Se ocorrer um erro na execução (*run-time error*), o Turbo Pascal usará esta informação para localizar o erro no programa-fonte.

DIRETIVA DE COMPILAÇÃO F
Force Far Calls (Default é F–)
(Força Chamadas Distantes)

Todas as chamadas a procedimentos e funções são geradas como "chamadas distantes" (*far calls*), quando esta diretiva estiver ativa.

DIRETIVA DE COMPILAÇÃO L
(Link Buffer (Default é L+)
(Buffer para Lincagem)

O buffer para lincagem é uma porção de memória usada para acelerar o processo de lincagem, após a compilação.

DIRETIVA DE COMPILAÇÃO L
Link Object File (Sintaxe é {$L CURSOR.OBJ})
(Linca Arquivo Objeto)

No Turbo Pascal 4.0, procedimentos externos devem ser definidos em arquivos .OBJ e não mais .BIN. A diretiva L avisa ao Turbo que o arquivo indicado deve ser lincado no programa. O módulo-objeto deve ser um "Arquivo Objeto Relocável Intel" (Intel Relocatable Object File).

DIRETIVA DE COMPILAÇÃO M
Memory Allocation Sizes
(Sintaxe é {$M EspaçoStack, HeapMin, HeapMax})
(Espaços para Alocação de Memória)

Permite escolher a quantidade de espaço disponível para o Stack e o Heap. O estado default é {$M 16384, 0, 655360}, que significa 16 K para o Stack, memória mínima para o Heap é 0 K e o máximo é 640 K.

DIRETIVA DE COMPILAÇÃO N
Numeric Co-processor (Default é N–)
(Co-processador aritmético)

Quando a diretiva N está desativada (N–), os cálculos de tipo real são realizados pelo 8088 ou 80x86. Porém, quando ativa (N+), os cálculos de tipo real, usando variáveis declaradas como single, double, extended e comp, são realizados pelo co-processador aritmético.

DIRETIVA DE COMPILAÇÃO S
Stack Overflow Checking (Default é S+)
(Checagem de Overflow do Stack)

Quando ativa, checa as condições de overflow do Stack. No Turbo 3.0, isto era controlado pela diretiva K.

DIRETIVA DE COMPILAÇÃO T
TPM File Generation (Default é T–)
(Geração de Arquivo TPM)

Se a diretiva estiver ativa, um arquivo tipo .TPM será gerado quando um programa for compilado no disco.

DIRETIVA DE COMPILAÇÃO U
Unit File Name (Sintaxe é {$U nomearq})
(Nome do Arquivo que contém a Unidade)

Permite especificar o nome do arquivo onde se encontra a diretiva. O {$U nomearq} deve estar no comando Uses, logo depois do nome da unidade.

NOVOS TIPOS ESCALARES DA VERSÃO 4.0

Word

Pode ser considerado como um inteiro sem sinal. Ocupa dois bytes, porém não usa o bit de maior ordem para sinal, fazendo com que os valores para o tipo **word** fiquem entre 0 e 65535.

ShortInt

O tipo **ShortInt** ("inteiro curto") pode ser considerado um byte com sinal. Ocupa somente um byte de memória e usa o bit mais elevado para sinal, fazendo com que os valores para o tipo **ShortInt** fiquem entre –128 e 127.

LongInt

Utiliza quatro bytes para armazenamento e tem uma faixa de valores entre –2.147.483.648 e 2.147.483.647.

NOVOS TIPOS REAIS DA VERSÃO 4.0

(somente aos permitidos na presença de um co-processador aritmético)

Single

Utiliza quatro bytes para armazenamento e é o menos preciso dos novos tipos reais. Sua faixa de valores varia entre 1.5E-45 e 3.4E+38.

Double

Ocupa oito bytes e varia entre 5.0E–324 e 1.7E308.

Extended

Ocupa dez bytes e varia entre –2E+63+1 e 2E+63–1.

Comp

Tecnicamente este tipo é um real, porém comporta-se como um inteiro (e só pode armazenar inteiros). Ocupa oito bytes e varia entre –2E+63+1 e 2E+63–1.

CONSTANTES

A única diferença com a versão anterior é que agora as constantes tipadas são armazenadas no segmento de dados.

O OPERADOR @

O operador @ é usado para o tratamento de alocação de memória dinâmica e retorna o endereço, no formato de ponteiro, de uma variável declarada formalmente. Ele simplifica o uso de ponteiros quando usados para sobrepor outras variáveis na memória.

ENTRADA E SAÍDA

Existem algumas mudanças:

. A versão 3.0 suportava os dispositivos **CON:**, **TRM:** e **LST:**. Na versão 4.0, foram substituídos pelos dispositivos **CON** ("console": vídeo para saída e teclado para entrada), **PRN**, **LPT1**, **LPT2** e **LPT3** (referentes à impressora).

. O dispositivo **LST** ainda funciona para saída na impressora se usado junto com a unidade **Printer**.

. O Turbo Pascal 4.0 tem a capacidade de escrever diretamente na memória de vídeo ou usar as rotinas do BIOS quando usa a unidade **Crt**. Você controla o método usando a variável **DirectVideo**. Se **DirectVideo** for verdadeira (default), será usado acesso direto, caso contrário, serão usadas chamadas ao BIOS.

. A utilização de uma nova função para leitura de um caractere pressionado no teclado. O **Readkey** retorna um caractere pressionado. Se a tecla gerar um código de "scan", **Readkey** retornará # 0 e uma nova chamada a **Readkey** lerá o caractere seguinte ao código de "scan".

SERVIÇOS DO BIOS E D.O.S.

TIPOS PREDEFINIDOS NA UNIDADE DOS

```
Registers = record
              case Integer of
```

```
                    0: (AX,BX,CX,DX,BP,SI,DI,DS,ES, Flags: Word);
                    1: (AL,AH,BL,BH,CL,CH,DL,DH          : BYTE);
                  END;

FileRec = record
            Handle    : Word;
            Mode      : Word;
            RecSize   : Word;
            Private   : array[1..26] of Byte;
            UserData  : array[1..16] of Byte;
            Name      : array[0..79] of Char;
          end;

TextBuf = array[0..127] of Char;

TextRec = record
            Handle    : Word;
            Mode      : Word;
            BufSize   : Word;
            Private   : Word;
            BufPos    : Word;
            BufEnd    : Word;
            BufPtr    : ^TextBuf;
            OpenFunc  : Pointer;
            InOutFunc : Pointer;
            FlushFunc : Pointer;
            CloseFunc : Pointer;
            UserData  : array[1..16] of Byte;
            Name      : array[0..79] of Char;
            Buffer    : TextBuf;
          end;

SearchRec = record
            Fill      : array[1..21] of Byte;
            Attr      : Byte;
            Time      : Longint;
            Size      : Longint;
            Name      : string[12];
          end:
```

```
DateTime = record
             Year,Month, Day,Hour,Min,Sec: Word;
           end;
```

VARIÁVEL PREDEFINIDA

Contém um código referente ao erro ocorrido, que é armazenado na variável inteira **DosError:**

0 nenhum erro
2 arquivo não foi encontrado
3 path não encontrado
5 acesso proibido
6 handle inválido
8 não há memória suficiente
10 ambiente inválido
11 formato inválido
18 mais nenhum arquivo

PROCEDIMENTOS DE DATA E HORA

GetDate(var Ano, Mes, Dia, DiaSemana: Word);

Retorna a data do sistema nos parâmetros Ano, Mês, Dia, DiaSemana. O ano contém o século (1988 em vez de 88) é o DiaSemana varia de 0 a 6, sendo o domingo representado pelo 0.

SetDate(Ano, Mes, Dia: Word);

Ajusta a data do sistema. O ano deve incluir o século (exemplo: 1988). Se os valores dos parâmetros estiverem errados, a data não será corrigida.

GetTime(var, Hora, Min, Seg, DecSeg: Word);

Retorna a hora do sistema nos parâmetros Hora, Min, Seg, DecSeg.

SetTime(Hora, Min, Seg, DecSeg: Word);

Ajusta a hora do sistema. Se os valores dos parâmetros estiverem errados, a hora não será corrigida.

GetFTime(var F; var Tempo: Longint);

Lê a identificação de tempo do arquivo F. O conteúdo desta identificação, armazenado na variável **Tempo**, contém a data e hora do arquivo. Para decodificar o conteúdo de **Tempo**, deve-se usar o procedimento **UnpackTime.**

SetFTime(var F; Tempo: Longint);

Marca a identificação de tempo do arquivo **F**. Antes de chamar este procedimento, deve-se criar a variável **Tempo** usando o procedimento **PackTime.**

UnpackTime(Tempo: Longint; var DT: DateTime);

Interpreta a variável **Tempo** que contém data e hora de um arquivo. Este procedimento deve ser usado depois de **GetFTime.**

PackTime(var DT: DateTime; var Tempo: Longint);

Compacta os dados de data e hora para a variável **Tempo.** Isto é usado antes do procedimento **SetFTime.**

FUNÇÕES DE CONSULTA SOBRE O ESPAÇO EM DISCO

DiskFree(Drive: Byte): Longint;

Retorna o espaço livre no disco contido em **Drive** (especificado por um número: 0 – unidade em uso, 1 – unidade A, 2 – unidade B etc...).

DiskSize(Drive: Byte): Longint;

Retorna o espaço total do disco contido em **Drive** (especificado por um número: 0 – unidade em uso, 1 – unidade A, 2 – unidade B etc. = ..).

PROCEDIMENTOS PARA ARQUIVOS

GetFAttr(var F; var Atrib: Word);

Retorna o atributo da variável de arquivo F, que já deve ter sido associada (com Assign), mas não aberto.

SetFAttr(var F; Atrib: Word);

Ajusta o atributo da variável de arquivo F, que já deve ter sido associada (com Assign), mas não aberto.

FindFirst(Path: String; Atrib: Word; var F: SearchRec);

Procura no diretório indicado por Path e retorna o primeiro arquivo normal ou arquivo que com atributos que se encaixem naqueles contidos em **Atrib.**

FindNext(var F: SearchRec);

Continua a busca iniciada por **FindFirst.** A variável **DosError** conterá o valor 18 quando não achar mais nenhum arquivo. Se nenhum atributo for indicado (ex.: **atrib**=0), estes procedimentos retornarão somente arquivos normais (aqueles sem atributos).

GRÁFICOS DA VERSÃO 4.0

Todos os programas escritos na versão 3.0 podem ser rodados na 4.0, se usarem a unidade **Graph3.**

Além disto, a versão 4.0 ainda tem uma unidade chamada **Graph** com novas rotinas gráficas.

CONSTANTES GRÁFICAS

Códigos retornados em GraphResult:

```
grOk            = 0;
grNoInitGraph   = -1;
grNotDetected   = -2;
grFileNotFound  = -3
grInvalidDriver = -4;
grNoLoadMem     = -5;
grNoScanMem     = -6;
grNoFloodMem    = -7;
grFontNotFound  = -8;
grNoFontMem     = -9;
grInvalidMod    = -10
grError         = -11;  {Erro genérico}
```

```
grIOerror          = -12;
grInvalidFont      = -13;
grInvalidFontNum   = -14;
grInvalidDeviceNum = -15;
```

Define drivers gráficos

```
Detect     = 0;  {ativa detecção automática da placa em uso}
CGA        = 1;
MCGA       = 2;
EGA        = 3;
EGA64      = 4;
EGAMono    = 5;
RESERVED   = 6;
HercMono   = 7;
ATT400     = 8;
VGA        = 9;
PC3270     = 10;
```

Modos gráficos para cada driver

```
CGACO      = 0;  { 320x200 palette 0: Verde Claro, Vermelho Claro,
                   Amarelo; 1 página }
CGAC1      = 1;  { 320x200 palette 1: Azul Ciano Claro, Magenta Claro,
                   Branco; 1 página }
CGAC2      = 2;  { 320x200 palette 2: Verde, Vermelho, Marrom;
                   1 página }
CGAC3      = 3;  { 320x200 palette 3: Azul Ciano, Magenta, Cinza Claro;
                   1 página }
CGAHi      = 4;  { 640x200: 1 página }
MCGACO     = 0;  { 320x200 palette 0: Verde Claro, Vermelho Claro,
                   Amarelo; 1 página }
MCGAC1     = 1;  { 320x200 palette 1: Azul Ciano Claro, Magenta Claro,
                   Branco; 1 página }
MCGAC2     = 2;  { 320x200 palette 2: Verde, Vermelho, Marrom;
                   1 página }
MCGAC3     = 3;  { 320x200 palette 3: Azul Ciano, Magenta, Cinza Claro;
                   1 página }
```

```
MCGAMed     = 4;  { 640x200: 1 página }
MCGAHi      = 5;  { 640x480: 1 página }
EGALo       = 0;  { 640x200: 16 cores; 4 páginas }
EGAHi       = 1;  { 640x350: 16 cores; 2 páginas }
EGA64Lo     = 0;  { 640x200 16 cores; 1 página }
EGA64Hi     = 1;  { 640x350 4 cores; 1 página }
EGAMonoHi   = 3;  { 640x350 64K de memória na placa, 1 página; 256K,
                    2 páginas }
HercMonoHi  = 0;  { 720x348; 2 páginas }
ATT400C0    = 0;  { 320x200 palette 0: Verde Claro, Vermelho Claro,
                    Amarelo; 1 página }
ATT400C1    = 1;  { 320x200 palette 1: Azul Ciano Claro, Magenta Claro,
                    Branco; 1 página }
ATT400C2    = 2;  { 320x200 palette 2: Verde, Vermelho, Marrom;
                    1 página }
ATT400C3    = 3;  { 320x200 palette 3: Azul Ciano, Magenta, Cinza Claro;
                    1 página }
ATT400Med   = 4;  { 640x200: 1 página }
ATT400Hi    = 5;  { 640x400: 1 página }
VGALo       = 0;  { 640x200: 16 cores; 4 páginas }
VGAMed      = 1;  { 640x350: 16 cores; 2 páginas }
VGAHi       = 2;  { 640x480: 16 cores; 1 página }
PC3270Hi    = 0;  { 720x350: 1 página }
MaxColors   = 15;
```

Estilos de Linha e Espessuras para Get/SetLineStyle

```
SolidLn     = 0;
DottedLn    = 1;
CenterLn    = 2;
DashedLn    = 3;
UserBitLn   = 4;  { Estilo de linha definido pelo usuário }
NormWidth   = 1;
ThickWidth  = 3;
```

Constantes para Set/GetTextStyle

```
DefaultFont = 0;
TriplexFont = 1;
```

```
SmallFont     = 2;
SansSerifFont = 3;
GothicFont    = 4;
HorizDir      = 0;  { Da esquerda para a direita }
VertDir       = 1;  { De baixo para cima }
UserCharSize  = 0;  { Tamanho de caractere definido pelo usuário }
```

Constantes para Clipping

```
ClipOn    = true;
ClipOff   = false;
```

Constantes para Bar3D

```
TopOn     = true;
TopOff    = false;
```

Padrões para Preenchimento com Get/SetFillStyle

```
EmptyFill     = 0;  { preenche área na cor de fundo }
SolidFill     = 1;  { preenche área com cores sólidas }
LineFill      = 2;  { padrão --- }
LtSlashFill   = 3;  { padrão /  /  /  }
SlashFill     = 4;  { padrão ///  com linhas grossas }
BkSlashFill   = 5;  { padrão \\\ com linhas grossas }
LtBkSlashFill = 6;  { padrão \\\  }
HatchFill     = 7;  { hachura horizontal }
XHatchFill    = 8;  { hachura diagonal }
InterleaveFill = 9; { padrão reticulado }
WideDotFill   = 10; { padrão pontilhado muito espaçado }
CloseDotFill  = 11; { padrão pontilhado pcuco espaçado }
UserFill      = 12; { padrão definido pelo usuário }
```

Operadores BitBlt para o PutImage

```
NormalPut = 0;   { MOV }
XORPut    = 1;   { XOR }
OrPut     = 2;   { OR }
AndPut    = 3;   { AND }
NotPut    = 4;   { NOT }
```

Justificação Horizontal e Vertical para SetTextJustify

```
LeftText   = 0;
CenterText = 1;
RightText  = 2;
BottomText = 0;
TopText    = 2;
```

TIPOS GRÁFICOS PREDEFINIDOS

```
PaletteType = record
    Size       : byte
    Colors     : array[0..MaxColors] of shortint;
  end;

LineSettingsType = record
    LineStyle  : word;
    Pattern    : word;
    Thickness  : word;
  end;

TextSettingsType = record
    Font       : word
    Direction  : word;
    CharSize   : word;
    Horiz      : word;
    Vert       : word;
  end;

FillSettingsType = record
    Pattern    : word;
    Color      : word;
  end;

FillPatternType = array[1..8] of byte;

PointType = record
     X, Y : integer;
  end;
```

```
ViewPortType = record
    x1, y1, x2, y2 : integer;
    Clip           : boolean;
  end;

ArcCoordsType = record
    X, Y           : integer;
    Xstart, Ystart : integer;
    Xend, Yend     : integer;
  end;
```

VARIÁVEIS GRÁFICAS PREDEFINIDAS

```
GraphGetMemPtr  : Pointer;
GraphFreeMemPtr : Pointer;
```

PROCEDIMENTOS E FUNÇÕES GRÁFICAS

(Necessitam de **Uses Graph** no programa)

Funções de Manutenção de erros

GraphErrorMsg (CodigoErro : integer) : String;

Retorna uma string contendo uma mensagem para o erro indicado por **CodigoErro.**

GraphResult : Integer;

Retorna um código de erro para a última operação realizada.

Procedimentos e Funções para Detecção, Inicialização e Modos de Vídeo

DetectGraph (var DriverGraf, ModoGraf : integer);

Analisa qual placa de vídeo está sendo usada e recomenda o modo gráfico mais adequado.

```
InitGraph var DriverGraf : integer;
          var ModoGraf   : integer;
          DriverPath     : String);
```

Inicializa o ambiente gráfico e entra em modo gráfico. Se o **DriverGraf** for **Detect** (0), o **InitGraph** vai determinar qual driver e qual modo devem ser usados. **DriverPath** indica qual o caminho (path) para o diretório onde os drivers gráficos se encontram. Deixa a posição atual no topo esquerdo da tela (0,0).

SetGraphMode (Modo : integer);

Ajusta o ambiente gráfico para o **Modo** gráfico escolhido. Deixa a posição atual no topo esquerdo da tela (0,0).

GetGraphMode : integer;

Esta função retorna um valor inteiro que indica o modo gráfico atualmente em uso.

GetModeRange (DriverGraf: integer; var LoMode, HiMode: integer);

Retorna os modos máximo e mínimo para um driver gráfico especificado.

RestoreCrtMode;

Retorna o modo da tela anterior para o modo gráfico.

CloseGraph;

Volta a tela ao modo anterior e retira do heap todo as variáveis dinâmicas usadas pelo modo gráfico.

GraphDefaults;

Muda a posição atual para o topo esquerdo da tela (0,0) e inicializa todos os procedimentos gráficos (viewport, palette, cores, características das linhas, textos gráficos etc...).

GetX : integer;

Função que retorna a coordenada horizontal da posição atual, sempre relativa ao **viewport** em uso. Isto quer dizer que se GetX retornar 0, a posição será no lado esquerdo do **viewport** e não no lado esquerdo da tela.

GetY : integer;

Função que retorna a coordenada vertical da posição atual, sempre relativa ao **viewport** em uso. Isto quer dizer que se GetY retornar 0, a posição será no topo do **viewport** e não no topo da tela.

GetMaxX : integer;

Função que retorna o valor da coordenada horizontal do pixel no canto mais à direita.

GetMaxY . integer;

Função que retorna o valor da coordenada vertical do pixel no canto inferior.

Rotinas de Tela e ViewPort

ClearDevice;

Apaga a tela atual deixa a posição atual no topo esquerdo do viewport (0,0).

SetViewPort (x1, y1, x2, y2 : integer; Clip : boolean);

Define uma porção da tela, de x1, y1 (canto superior esquerdo) até x2,y2 (canto inferior direito). Se **Clip** for verdadeiro, o clipping estará ligado. Depois do **viewport** ligado, todas as coordenadas estarão relacionadas com o **viewport** e não com as coordenadas físicas da tela. Deixa a posição atual no topo esquerdo (0,0).

GetViewSettings (var ViewPort : ViewPortType);

Retorna uma variável-registro predefinida contendo os dados do viewport atual.

ClearViewPort;

Limpa o viewport e assume as cores da palette (0) para a tela. Deixa a posição atual no topo esquerdo da tela (0,0).

SetActivePage (Pagina : word);

Direciona toda saída da tela para a página indicada. Para fazer a página ser visualizada, use o procedimento SetVisualPage.

SetVisualPage (Pagina : word);

Troca a página gráfica visualizada na tela por aquela indicada na variável **pagina.**

Rotinas de Manuseio de Pixels

PutPixel (X, Y : integer; Cor : word);

Coloca um pixel na coordenada x, y. A cor é determinada pela variável **cor.**

GetPixel (X, Y : integer) : word;

Esta função retorna o valor da cor do pixel na coordenada x, y.

Rotinas para Criação de Linhas

Line To (X, Y : integer);

Desenha uma linha da posição atual até x, y.

LineRel (Dx, Dy : integer);

Desenha uma linha da posição atual até a posição relativa definida por Dx, Dy.

MoveTo (X, Y : integer);

Move a posição atual para a coordenada x, y.

MoveRel (Dx, Dy : integer);

Move a posição atual para uma nova posição deslocada Dx, Dy da original.

Line (x1, y1, x2, y2 : integer);
Desenha uma reta da coordenada x1, y1 até x2, y2

GetLineSettings (var InfoLinha : LineSettingsType);
Retorna em **InfoLinha** (variável registro) três dados que determinam a aparência da linha desenhada.

SetLineStyle EstiloLinha : word;
Padrao : word;
Espessura : word);

Define o estilo da linha, o padrão e a espessura a serem usados ao desenhar uma linha.

Rotinas para Criação de Polígonos, Figuras e Preenchimento de Áreas

Rectangle (x1, y1, x2, y2 : integer);

Desenha um retângulo da coordenada x1, y1 (canto superior esquerdo) até a coordenada x2, y2 (canto inferior direito), usando o estilo de linha e cor atuais.

Bar (x1, y1, x2, y2 : integer);

Desenha um retângulo da coordenada x1, y1 (canto superior esquerdo) até a coordenada x2, y2 (canto inferior direito), preenchendo-o com o padrão (pattern) selecionado com **SetFillStyle.**

Bar3D (x1, y1, x2, y2 : integer; Prof : word; Topo : boolean);

Desenha um retângulo da coordenada x1, y1 (canto superior esquerdo) até a coordenada x2,y2 (canto inferior direito), preenchendo-o com o padrão (pattern) selecionado com **SetFillStyle.**

DrawPoly (NumPontos : word; var PontosPoly);

Desenha uma poligonal que liga os pontos definidos na matriz **PontosPoli** (matriz tipo registro com campos para X e Y do tipo word). A quantidade de pontos é definida em **NumPontos.** Em caso de coordenadas erradas na matriz, **GraphResult** retornará o valor –6.

FillPoly (NumPontos : word; var PontosPolig);

Desenha uma poligonal que liga os pontos definidos na matriz **PontosPolig** (matriz tipo registro com campos para X e Y do tipo word), preenchendo-a com o padrão e a cor definidos por **SetFillPatern, SetFillStyle** e **SetColor.** A linha que limita o polígono é desenhada de acordo com o estilo de linha e cor atuais e deve estar perfeitamente fechada. A quantidade de pontos no polígono é definida em **NumPontos.** Em caso de coordenadas erradas na matriz, **GraphResult** retornará o valor –6.

GetFillSettings (var InfoFill : FillSettingsType);

Retorna um registro do tipo **FillSettingsType** contendo dois parâmetros **Pattern** e **Color**, que indicam o tipo de hachura e a cor atuais.

SetFillStyle (Padrao : word; Cor : word);

Escolhe uma cor e um padrão de preenchimento (12 predefinidos indo de 0 a 11).

SetFillPattern (Padrao : FillPatternType; Cor : word);

Define um padrão e uma cor a serem usadas em preenchimento de áreas gráficas. **Padrao** é uma variável do tipo predefinido FillPatternType (uma matriz de bytes com 8 elementos) que gera um padrão de 8x8 pixels, sendo um pixel para cada bit do byte.

GetFillPatern (var PadraoFill : FillPatternType);

Retorna o último padrão usado por **SetFillPattern**

FloodFill (X, Y : integer; Border : word);

Preenche a área fechada pela linha de cor **Borda.** Esta área deve conter o ponto de coordenada x, y.

SetGraphBufSize (TamBuf : word);

Permite mudar o tamanho do buffer (TamBuf) reservado para preenchimento de áreas.

Rotinas de Arcos, Círculos e outras Curvas

Arc (X, Y : integer; AngInic, AngFinal, Raio : word);

Desenhe um arco com centro em x, y, de raio especificado em **Raio** O arco começa em **AngInic** e termina em **AngFinal**, sendo desenhado sempre no sentido anti-horário (ângulo 0 equivale a leste ou 3:00 horas).

GetArcCoords (var CoordArco : ArcCoordsType);

Retorna uma variável-registro contendo dados sobre o último arco criado pelo procedimento Arc, dados estes que podem ser usados para criar uma "fatia" em gráficos de torta (ao ligar linhas do centro do arco para as suas extremidades).

Circle (X, Y : integer; Raio : word);

Desenha um círculo de centro em x, y e raio especificado.

Ellipse (X, Y : integer;
AngiNic, AngFinal : word;
XRaio, YRaio : word);

Produz uma elipse de centro em x, y. O desenho começa em **AngInic** e termina em **AngFinal**, e a elipse torna-se mais alongada quanto maior for a diferença entre **XRaio** e **YRaio**. Além disto, é desenhada sempre no sentido anti-horário (ângulo 0 equivale a leste ou 3:00 horas).

GetAspectRatio (var Xasp, Yasp : word);

A razão de aspecto é a razão entre os pixels horizontais e verticais que produz uma imagem bem proporcionada na tela. Depois de chamar **GetAspectRatio** divida **Xasp** por **Yasp** para determinar a razão de aspecto do seu equipamento.

PieSlice (X, Y : integer; AngInic, AngFinal, Raio : word):

Desenha uma "fatia" para um gráfico tipo torta, com centro em x, y e raio determinado em **Raio.** O desenho começa em **AngInic** e termina em **AngFinal**, sempre em sentido anti-horário (ângulo 0 equivale a leste ou 3:00 horas). A "fatia" será preenchida com o padrão e a cor definidos em **SetFillStyle** e **SetFillPattern.**

ROTINAS PARA PALETTES E CORES

SetBkColor (Cor : word);

Escolhe a cor de fundo indicada por **Cor.** SetBKColor (0) muda a cor de fundo para preto.

SetColor (Cor : word);

Deixa a cor para desenho valendo **Cor.**

GetBkColor : word;

Esta função retorna um número que indica a cor de fundo atual.

GetColor: word;

Esta função retorna um número que indica a cor atual.

GetMaxColor : word;

Esta função retorna o número máximo da cor que pode ser passada como parâmetro em **SetColor.**

SetAllPalette (var Palette);

Troca os dados da palette atual por aqueles contidos na variável-registro predefinida **Palette.**

SetPalette (NumCor : word; Cor : shortint);

Altera a cor de número **NumCor** para a definida em **Cor**, permitindo alterar o padrão da palette.

GetPalette (var Palette : PaletteType);

Retorna os valores atuais da palette numa variável tipo registro predefinida com dois campos: **Size** que indica o número de cores na palette e **Colors** que contém as cores da palette.

ROTINAS DE MANUSEIO DE BLOCOS DE PIXELS

ImageSize (x1, y1, x2, y2 : integer) : word;

Esta função retorna a quantidade de memória necessária para armazenar a imagem definida em x1, y1 e x2, y2.

GetImage (x1, y1, x2, y2 : integer; var BitMap);

Captura a imagem contida em x1, y1 (canto superior esquerdo) e x2, y2 (canto inferior direito) no buffer **BitMap**. Este buffer tem de ser pelo menos do tamanho definido "mais quatro".

PutImage (X, Y : integer; var BitMap; BitBlt : word);

Transfere a imagem contida no buffer **BitMap** para a tela na coordenada x, y, sem fazer "clipping" (ou seja: se a imagem sair fora do limite, não será transferida), **BitBlt** indica como a imagem deve ser transferida, de acordo com as constantes predefinidas NormalPut (MOV), XORPut (XOR), OrPut (OR), AndPut (AND) e NotPut (NOT).

Rotinas para Textos no Modo Gráfico

GetTextSettings (var InfoText : TextSettingsType);

Retorna os dados a respeito do estilo do tipo de letra atual. Os valores são retornados dentro de uma variável-registro de tipo predefinido e dizem respeito a tipo escolhido, direção, tamanho e justificação horizontal ou vertical.

OutText (StringTexto : string);

Escreve a string contida em **StringTexto** na posição atual da tela gráfica.

OutTextXY (X, Y : integer; StringTexto : string);

Escreve a string contida em **StringTexto** na posição x,y da tela gráfica. Não atualiza a posição atual e não faz "clipping" com caracteres gráficos default.

SetTextJustify (Horiz, Vert : word);

Determina a maneira como o texto vai ser escrito em relação à posição atual. Existem constantes predefinidas para valores horizontais (LeftText, CenterText e RightText) e verticais (BottomText, CenterText e TopText).

SetTextStyle (Font, Direcao : word; TamCarac : word);

Define a maneira como o texto vai ser mostrado na tela. Existem cinco tipos de letras disponíveis (de 0 a 4, sendo que do 1 ao 4 ficam armazenados em disco). Existem constantes predefinidas para serem usadas como valor para **Font** (DefaultFont, TriplexFont, SmallFont, SansSerifFont e GothicFont), outras para **Direcao** (HorizDir e VertDir) e também para o tamanho do caractere dado por **TamCarac.**

TextHeight (StringTexto : string) : word;

Esta função retorna a altura em pixels da string contida em **StringTexto.** A altura depende da fonte em uso.

TextWidth (StringTexto : string) : word;

Esta função retorna o comprimento em pixels da string contida em **StringTexto** (depende da fonte em uso).

SetUserCharSize (MultX, DivX, MultY, DivY : byte);

Permite alterar o comprimento ou altura para alguns tipos de letras.

FUNÇÕES ESPECIAIS

RegisterBGIfont (font : pointer): integer;

Esta função retorna um número inteiro que pode indicar se houve um erro (número negativo), ou retorna o número do tipo de letra atual.

RegisterBGIdriver (driver : pointer) : integer;

Esta função retorna um número inteiro que pode indicar se houve um erro (número negativo), ou retorna o número do driver atual.

PROCEDIMENTOS PARA INTERRUPÇÕES, PROGRAMAS RESIDENTES E SAÍDAS PARA O D.O.S.

DosExitCode : Integer

Esta função retorna o código de saída de um subprocesso. O byte mais elevado do inteiro será 0 para terminação normal, 1 para terminação forçada por CTRL-C e 3 quando terminada pelo procedimento **keep.**

GetIntVec (NumInt: Byte; var Vetor: Pointer);

Retorna o endereço usado atualmente pela interrupção **NumInt** no parâmetro **Vetor.**

SetIntVec (NumInt: Byte: Vetor: Pointer);

Troca o endereço atual da interrupção **NumInt** pelo endereço contido em **Vetor.**

Keep (CodSaida: Word);

Permite terminar o programa e deixá-lo residente em memória. **CodSaida** contém o código de saída do DOS depois de terminada a execução do programa.

Exec (PathCommandCom, Comando: String);

Permite a saída do seu programa para o sistema operacional.

PathCommandCom deve indicar o caminho e o nome do Command.Com em seu disco. **Comando** deve conter a ordem a ser dada no sistema operacional. se **Comando** contiver uma string vazia, o Turbo Pascal deixará executar comandos diretamente no prompt do sistema operacional. Neste caso, para retornar ao programa em Pascal, digite **EXIT.**

MENSAGENS DE ERRO DO TURBO 4.0

Houve mudanças referentes às mensagens de erro na execução e erros de Entrada e Saída (I/O):

- Erros de Entrada e Saída são considerados como erros na execução.
- Os códigos dos erros vêm diretamente do DOS.

Erros na Execução (Run-Time Errors)

Decimal	Significado
1	Código de função para DOS errado
2	Arquivo não encontrado
3	Path (caminho) não encontrado
4	Muitos arquivos abertos
5	Negado acesso ao arquivo
6	Handle inválido para o arquivo
8	Pouca memória
12	Código inválido de acesso ao arquivo
15	Número inválido do drive
16	Não pode remover o diretório atual
17	Não pode trocar nomes entre discos
100	Erro na leitura do disco
101	Erro na gravação em disco
102	Arquivo não assinalado
103	Arquivo fechado
104	Arquivo fechado para entrada
105	Arquivo fechado para saída
106	Formato numérico inválido
200	Divisão por zero
201	Erro na checagem de faixa
202	Erro de overflow no Stack
203	Erro de overflow no Heap
204	Operação de pointer (ponteiro) inválida
205	Overflow de ponto-flutuante

TOOLBOXES E A VERSÃO 4.0

Existem algumas mudanças com relação aos toolboxes. Todos eles foram adaptados para as novas características do Turbo Pascal 4.0 (formato usando Units, uso do 80x87 etc...). As mudanças, porém, não afetam a sintaxe das chamadas para manter o máximo de compatibilidade.

- **Numerical Methods Toolbox** – aceita compilação com 80x87 (se presente).
- **Editor's Toolbox** – rotinas reescritas para dar maior velocidade. Processador de textos Microstar completamente reescrito e melhorado. Fornece um editor totalmente escrito em assembly que pode ser incluído em qualquer programa com o comando **Uses**.
- **Graphix Toolbox** – As rotinas Inline foram convertidas para arquivos externos que devem ser lincados com a diretiva ($L).
- **DataBase Toolbox** – Suporta arquivos de banco de dados com mais de 2 bilhões de entradas. Inclui rotinas para ler ou gravar arquivos no formato usado pelo Reflex (banco de dados comercializado pela Borland).

APÊNDICE C

VERSÃO 5.0*

AMBIENTE DE TRABALHO

Para selecionar um item do menu é só deslocar o cursor com as setas (o fundo da opção selecionada fica reverso) ou digitar o primeiro caractere.

OPÇÕES DO MENU PRINCIPAL

FILE (arquivo)

- **LOAD** (carregar) — Lê um arquivo do disco para o Editor. Se o arquivo não existir, será criado.
- **PICK** (pegar) — Permite a escolha de um arquivo entre os oito últimos arquivos editados recentemente.
- **NEW** (novo) — Limpa o editor e deixa o nome do arquivo como NONAME.PAS.
- **SAVE** (salvar) — Grava o arquivo da memória para o disco.
- **WRITE TO** (escreva em) — Escreve o conteúdo do Editor num arquivo indicado pelo usuário.

---

* *Apêndices B e C elaborados por Roberto Bertini Renzetti*

- **DIRECTORY** (diretório) — Somente mostra o diretório, de acordo com uma máscara especificada.
- **CHANGE DIR** (mudar de diretório) — Permite mudar o drive ou diretório atual.
- **OS SHELL** (ambiente do Sistema Operacional) — Sai do Turbo Pascal e volta temporariamente para o Sistema Operacional, onde pode-se executar qualquer comando ou executar alguns programas. Para voltar ao Turbo é só digitar EXIT.
- **QUIT** (sair) — Sai do Turbo Pascal, voltando para o Sistema Operacional.

EDIT (editar)

Permite a edição de programas. Para voltar ao menu principal, tecle ^KD ou F10.

RUN (correr, executar)

- **RUN** — Completa, "linca" e executa seu programa.
- **PROGRAM RESET** — Termina a depuração do programa, livra memória e fecha arquivos abertos.
- **GO TO CURSOR** (vá para o cursor) — Permite executar o programa a partir da posição do cursor de edição.
- **TRACE INTO** — Executa o próximo comando do seu programa ou subprograma (procedimento ou função).
- **STEP OVER** — Executa o próximo comando do seu programa. Os subprogramas chamados serão executados de uma só vez.
- **USER SCREEN** (tela do usuário) — Permite visualizar a tela de saída do programa que está sendo executado pelo depurador ou que já foi executado pelo programa.

COMPILE (compilar)

- **COMPILE** (compilar) — Compila e "linca" o programa-fonte.
- **MAKE** (fazer) — Compila o arquivo, checando se alguma unidade (*unit*) foi alterada desde a última compilação. Se foi alterada, será recompilada.

- **BUILD** (construir) — Funciona de maneira semelhante à opção MAKE, porém forçando a recompilação das unidades.

- **DESTINATION** (destinação) — Permite selecionar a destinação do arquivo compilado entre memória e disco.

- **FIND ERROR** (achar erro) — Pode-se achar um erro de execução (*run-time error*) dando o endereço na forma XXXX:YYYY onde XXXX é segmento e YYYY o offset.

- **PRIMARY FILE** (arquivo primário ou princípal) — O mesmo que o "Main File" na versão 3.0. O Turbo sempre começa a compilação por este arquivo.

- **GET INFO** (recuperar informação) — Mostra informações a respeito do arquivo sendo trabalhado.

OPTIONS

- **COMPILER** (compilador) — Permite a inserção de diretivas de compilação sem escrevê-las no programa-fonte. Também usado para selecionar necessidades de memória para seu programa, permitir (se necessário) que UNITS sejam transformadas em *overlays* e gerenciar a emulação ou não do co-processador aritmético, entre outras coisas.

- **LINKER** — Permite ajustar opções do linker (lincador) como ligar ou desligar o envio de informações a arquivos mapa (.MAP) e também especificar se o buffer de lincagem será na memória (maior velocidade) ou no disco (menor velocidade de lincagem).

- **ENVIRONMENT** (ambiente) — Permite a seleção das opções de gravação automática da configuração do Turbo Pascal e do arquivo editado, criação de cópias de segurança dos arquivos-fontes, seleção da quantidade de linhas para tela (25,43 ou 50 dependendo da sua placa gráfica), tamanho e visualização de janelas para Edit, Watch e Saída.

- **DIRECTORIES** (diretórios) — mostra ao Turbo em qual diretório devem ser procurados e/ ou gravados os arquivos (programas-fontes, arquivos de inclusão, unidades e arquivos-objetos).

- **PARAMETERS** (parâmetros) — Permite a especificação de parâmetros para quando o programa for compilado em memória.

- **SAVE OPTIONS** (opções para salvar) — Grava a configuração atual do Turbo Pascal para um arquivo em disco.
- **RETRIEVE OPTIONS** (opções para recuperar) — Carrega um arquivo de configuração do Turbo Pascal.

DEBUG (depurar, achar erros)

- **EVALUATE** (avaliar) — Permite ver o valor atual de uma variável ou expressão (podendo até ser usado como calculadora se você não estiver depurando o programa) e também permite alterar este valor. Podem ser usados especificadores de formato (*format specifiers*) para controlar melhor a visualização do conteúdo das variáveis.
- **CALL STACK** (chamar o Stack) — Permite visualizar, durante a depuração, as chamadas de procedimentos e funções (com seus parâmetros) realizadas para levá-lo a posição atual no programa.
- **FIND PROCEDURE** (ache o procedimento) — Permite achar um procedimento ou função no seu programa, mesmo que estejam em alguma Unit ou num Arquivo de Inclusão.
- **INTEGRATED DEBUGGING** (depuração integrada) — Permite ligar ou desligar a depuração dentro do ambiente do Turbo Pascal.
- **STAND-ALONE DEBUGGING** — Permite a inclusão de informações no arquivo compilado em disco (.EXE) para futura depuração com o Turbo Debugger da Borland.
- **DISPLAY SWAPPING** (troca de telas) — Permite controlar como será feita a visualização da tela durante a depuração.
- **REFRESH DISPLAY** (refazer a tela) — Refaz a tela do ambiente do Turbo Pascal.

BREAK/ WATCH

- **ADD WATCH** — Permite especificar uma variável e ter seus valores observados na janela **Watch** durante a depuração de um programa.

- **DELETE WATCH** — Permite deletar a expressão que está sendo observada na janela **Watch**. Também podem ser usadas as teclas **Del** ou **CTRL-Y** dentro da janela **Watch** para apagar a expressão desejada.
- **EDIT WATCH** — Permite editar a expressão que está sendo observada.
- **REMOVE ALL WATCHES** — Todas as expressões da janela **Watch**.
- **TOGGLE BREAKPOINT** — Liga ou desliga um ponto de parada (*breakpoint*) na linha atual.
- **CLEAR ALL BREAKPOINTS** — Retira todos os pontos de parada (*breakpoints*) do programa.
- **VIEW NEXT BREAKPOINT** — Leva o cursor até o próximo ponto de parada sem executar o programa.

UNITS

São unidades de código que contêm procedimentos e funções e são compiladas separadamente do programa principal. A vantagem de se usar as "Units" é que não é necessário recompilar o programa inteiro após uma pequena alteração, somente recompila-se a "Unit" afetada.

Existe também a possibilidade de compilar, testar e debugar uma unidade separadamente do programa principal. Uma vez testada e compilada, a unidade não precisará mais ser compilada. Em vez disso, a unidade será lincada com o programa que a use. Lincar uma unidade é mais rápido que compilá-la.

As unidades são exatamente como programas e possuem o seguinte formato:

- linha com o nome da unidade, que deve ser o mesmo nome do arquivo.
- uma seção de interface, área que se conecta com outros programas ou unidades — contendo o cabeçalho dos procedimentos e funções, variáveis globais e o uso de outras unidades.
- seção de implementação, contendo o corpo dos procedimentos e funções anunciados na seção de interface e variáveis globais de uso interno pela unidade.

Ao necessitar de alguns procedimentos ou funções, contidos numa unidade, chame-os com o comando **Uses**.

UNITS COMPILADAS

As unidades podem ser compiladas da mesma maneira que os programas. Uma vez compiladas, serão armazenadas pelo Turbo num arquivo de mesmo nome que a unidade, porém com terminação .TPU.

UNITS PREDEFINIDAS

Existem oito unidades já disponíveis que são: Crt, Dos, Graph, Graph3, Overlay, Printer, System, Turbo3.

A seguir veremos os procedimentos e funções definidos nestas unidades. Se você for utilizar um ou mais destes comandos **UTILIZE O COMANDO Uses SEGUIDO DO NOME DA UNIDADE, APÓS A ÁREA DO CABEÇALHO DO SEU PROGRAMA.**

UNIT system

COMANDOS E FUNÇÕES

Abs, Addr, Append, ArcTan, Assign, BlockRead, BlockWrite, CSeg, ChDir, Chr, Close, Concat, Copy, Cos, DSeg, Dec, Delete, Dispose, Eof, Eof, Eoln, Erase, Exit, Exp, FilePos, FileSize, FillChar, Flush, Frac, FreeMem, GetDir, GetMem, Halt, Hi, IOResult, Inc, Insert, Int, Length, Ln, Lo, Mark, MaxAvail, MemAvail, MkDir, Move, New, Odd, Ofs, Ord, ParamCount, ParamStr, Pi, Pos, Pred, Ptr, Randomize, Random, Readln, Read, Release, Rename, Reset, Rewrite, RmDir, Round, SPtr, SSeg, SeekEof, SeekEoln, Seek, Seg, SetTextBuf, Sin, SizeOf, Sqrt, Sqr, Str, Succ, Swap, Truncate, Trunc, UpCase, Val, Writeln, Write.

VARIÁVEIS

ErrorAddr, ErrorCode, ExitProc, FileMode, FreeMin, FreePtr, HeapError, HeapOrg, HeapPtr, InOutRes, Input, Output, OvrCodeList, OvrDebugPtr, OvrDosHandle, OvrEmsHandle, OvrHeapEnd, OvrHeapOrg, OvrHeapPtr, OvrHeapSize, OvrLoadList, PrefixSeg, RandSeed, SaveInt00, SaveInt02, SaveInt1B, SaveInt23, SaveInt24, SaveInt34, SaveInt35, SaveInt36, SaveInt37, SaveInt38, SaveInt39, SaveInt3A, SaveInt3B, SaveInt3C, SaveInt3D, SaveInt3E, SaveInt3F, SaveInt75, StackLimit, Test8087.

UNIT crt

COMANDOS E FUNÇÕES

AssignCrt, ClrEol, ClrScr, DelLine, Delay, GotoXY, HighVideo, InsLine, KeyPressed, LowVideo, NoSound, NormVideo, ReadKey, Sound, TextBackground, TextColor, TextMode, WhereX, WhereY, Window.

VARIÁVEIS

CheckBreak, CheckEof, CheckSnow, DirectVideo, LastMode, TextAttr, WindMin, WindMax.

UNIT dos

COMANDOS E FUNÇÕES

DiskFree, DiskSize, DosExitCode, DosVersion, EnvCount, EnvStr, Exec, FExpand, FSplit, FindFirst, FindNext, GetCBreak, GetDate, GetEnv, GetFAttr, GetFTime, GetIntVec, GetTime, GetVerify, Intr, Keep, MsDos, PackTime, SetCBreak, SetDate, SetFAttr, SetFTime, SetIntVec, SetTime, SetVerify, SwapVectors, UnPackTime.

VARIÁVEIS

DosError.

UNIT graph

COMANDOS E FUNÇÕES

Arc, Bar3D, Bar, Circle, ClearDevice, ClearViewPort, CloseGraph, DetectGraph, DrawPoly, Ellipse, FillPoly, FloodFill, GetArcCoords, GetAspectRatio, GetBkColor, GetColor, GetDriverName, GetFillPattern, GetFillSettings, GetGraphMode, GetImage, GetLineSettings, GetMaxColor. GetMaxMode, GetMaxX, GetMaxY, GetModeName, GetModeRange, GetPaletteSize, GetPixel, GetTextSettings, GetViewSettings, GetX, GetY, GraphDefaults, GraphErrorMsg, GraphResult, ImageSize, InitGraph, InstallUserDriver, InstallUserFont, LineRel, LineTo, Line, MoveRel, MoveTo, OutTextXY, OutText, PieSlice, PutImage, PutPixel, Rectangle, RegisterBGIdriver, RegisterBGIfont, RestoreCrtMode, Sector, SetActivePage, SetAllPalette, SetAspectRatio, SetBkColor, SetColor, SetFillPattern, SetFillStyle, SetGraphBufSize, SetGraphMode, SetLineStyle, SetPalette, SetRGBPalette, SetTextJustify, SetTextStyle, SetUserCharSize, SetViewPort, SetVisualPage, SetWriteMode, TextHeight, TextWidth.

VARIÁVEIS

GraphGetMemPtr, GraphFreeMemPtr.

UNIT graph3

Esta unidade permite usar os mesmos comandos gráficos existentes na versão 3.0. **Nota:** os gráficos da versão 4.0 (definidos na unit Graph) são mais velozes.

UNIT overlay

COMANDOS

OvrClearBuf, OvrGetBuf, OvrInit, OvrInitEMS, OvrSetBuf.

VARIÁVEIS

OvrResult

UNIT printer

Esta unidade só define **Lst** como sendo um arquivo de texto já direcionado para a impressora.

UNIT turbo3

Criada para manter compatibilidade dos programas com a versão 3.0.

PALAVRAS RESERVADAS DO TURBO PASCAL 4.0 e 5.0

Implementation

Declara o começo da seção de implementação numa Unit. A seção de implementação contém o corpo das rotinas declaradas na seção de interface. Só é necessário indicar o nome da rotina e seu corpo e não os parâmetros (indicados sempre na seção de interface).

Interface

Declara o começo da seção de interface numa Unit. A seção de interface contém o cabeçalho das rotinas (procedimentos e funções) e seus parâmetros, que serão "visíveis" a outros programas e unidades que utilizam a unidade atual.

Interrupt

Declara um procedimento como um procedimento de interrupção. Ex.:

Procedure handler (Flags, CS, IP, AX, BX, CX, DX, SI, DI, DS, ES, BP: word); Interrupt;

Unit

Declara que o arquivo é uma Unit.

Uses

Declara quais unidades serão acessadas pelo programa ou unidade.

DIRETIVAS DE COMPILAÇÃO DO TURBO 5.0

(Existe diferença entre as diretivas das Versões 3.0, 4.0 e 5.0.)

Diretiva de Compilação {$A }
Align Data (Default é {$A}, escopo global)
(Alinhar Dados)

Faz alinhamento de *words* para variáveis e tipos de dados, agilizando assim o tempo de execução do programa em máquinas que usem microprocessadores da família 80x86.

Diretiva de Compilação {$B }
Short-circuit Boolean Evaluation (Default é {$B—}, escopo local)
(Avaliação Booleana Rápida)

Reduz o tempo que o computador leva para determinar se uma expressão é verdadeira ou falsa.

Diretiva de Compilação {$D }
Debug Information (Default é {D+}, escopo global)
(Informação para Debug)

Quando ativa, gerará informações relacionando o programa-fonte com o programa-objeto. Se ocorrer um erro na execução (*run-time error*), o Turbo Pascal usará esta informação para localizar o erro no programa-fonte. Se o fonte for uma Unit, as informações serão anexadas no arquivo compilado .TPU, aumentando assim o tamanho do objeto.

Diretiva de Compilação {$E }
Emulation (Default é {$E+}, escopo global)
(Emulação do 80x87)

Permite a lincagem no seu programa de bibliotecas que emularão o co-processador aritmético 80x87, quando este não estiver presente na máquina. Veja também a diretiva {$N+}.

Diretiva de Compilação {$F }
Force Far Calls (Default é {$F—}, escopo local)
(Força Chamadas Distantes)

Todas as chamadas a procedimentos e funções são geradas como "chamadas distantes" (*far calls*), quando esta diretiva estiver ativa. Quando for compilar programas ou units com overlays ou variáveis procedurais, a Borland sugere usar a diretiva {$F+}.

Diretiva de Compilação {$I }
Input/ Output Checking (Default é {$I+}, escopo global)
(Checagem de Entrada e Saída)

Faz o programa parar ou não se houver um erro de entrada ou saída. Com {$I—} o programa não pára e o código do erro de entrada e saída será armazenado na função **IOResult**.

Diretiva de Compilação {$I nome-do-arquivo}
Include File (escopo local)
(Arquivo de Inclusão)

Permite incluir um arquivo (de extensão .PAS) na compilação de um programa, contanto que a diretiva não esteja colocada na área de comandos (entre BEGIN e END).

Diretiva de Compilação {$L }
Local Symbol Information (Default é {$L+}, escopo global)
(Informações sobre Símbolos Locais)

Controla a geração ou não de informações sobre símbolos locais (nomes e tipos de todas as variáveis locais e constantes).

Em uma Unit, as informações serão anexadas no arquivo compilado .TPU, aumentando assim o tamanho do objeto.

Diretiva de Compilação {$L nome-do-arquivo}
Link Object File (escopo local)
(Lincar Arquivo-Objeto)

Permite lincar um arquivo-objeto (extensão .OBJ) em subprogramas declarados como **external**.

Diretiva de Compilação {$M }
Memory Allocation Sizes
(Sintaxe é {$M EspaçoStack. HeapMin, HeapMax})
(Default é {$M 16384, 0, 655360}, escopo global)
(Espaços para Alocação de memória)

Permite escolher a quantidade de espaço disponível para o Stack e o Heap. O estado default é {$M 16384, 0, 655360}, que significa 16 K para o Stack. Memória mínima para o Heap é 0 K e máxima é 640 K.

EspaçoStack pode ter valor inteiro entre 1024 a 65520. HeapMin pode variar entre 0 a 655360 e HeapMax varia entre HeapMin a 655360.

Não tem efeito nenhum em Units.

Diretiva de Compilação {$N }
Numeric Processing (Default é {$N—}, escopo global)
(Processamento numérico)

{$N—} faz com que o programa compilado utilize rotinas em software para execução das operações de ponto flutuante. Já a diretiva {$N+} faz com que o programa compilado use o co-processador aritmético 80x87 para executar operações com ponto flutuante (desde que utilizem os tipos próprios como **single, double, extended, comp**).

Veja também a diretiva {$E+}.

Diretiva de Compilação {$O }
Overlay Code Generation (Default é {$O—}, escopo global)
(Geração de Overlay)

Permite a geração de overlays (se necessário) a partir da compilação de uma Unit quando a diretiva de compilação for {$O+}.

Veja também a diretiva {$F+}.

Diretiva de Compilação {$O nome_unit}
Overlay Unit Name (escopo local)
(Tornar Unit em Overlay)

A diretiva {$O nome_da_unit} permite especificar no programa qual *unit* deve ser compilada e transformada em *overlay*. Extensão compilada com a diretiva {$O+} e o uso da diretiva {$O nome_da_unit} deve ser feita em seguida à área do Uses no programa.

Diretiva de Compilação {$R }
Range-Checking (Default é {$R—}, escopo global)
(Checagem de Faixa)

Permite a geração de código para checagem de faixas para índices de matrizes e strings e também atribuições a variáveis escalares. Faz o programa ficar mais extenso e mais lento, porém, se um erro de faixa acontecer, será exibida uma mensagem indicando-o.

Diretiva de Compilação {$S }
Stack Overflow Checking (Default é {S+}, escopo local)
(Checagem de Overflow do Stack)

Quando ativa, checa as condições de overflow do Stack. Uma mensagem de erro será apresentada se o procedimento ou função compilados com {$S+} forem chamados e não houver espaço no Stack.

Diretiva de Compilação {$V }
Var-String Checking (Default é {$V+}, escopo local)
(Checagem de Variáveis String)

Controla a checagem das strings passadas como parâmetros em procedimentos ou funções, exigindo que os parâmetros formais (nas definições dos procedures) e atuais (nas chamadas aos procedimentos) sejam absolutamente idênticos.

NOVOS TIPOS ESCALARES DA VERSÃO 4.0 e 5.0

WORD

Inteiro sem sinal. Ocupa dois bytes, e permite valores entre 0 e 65535.

SHORTINT

O tipo **ShortInt** ("inteiro curto") ocupa somente um byte de memória e permite valores entre -128 e 127.

LONGINT

Utiliza quatro bytes para armazenamento e tem uma faixa de valores entre -2.147.483.648 a 2.147.483.647.

NOVOS TIPOS REAIS DA VERSÃO 4.0

(Somente permitidos na presença de um co-processador aritmético ou se a diretiva de emulação do co-processador estiver ligada.)

SINGLE

Utiliza quatro bytes para armazenamento e permite valores entre 1.5E-45 a 3.4E+38 com 7 a 8 dígitos significativos.

DOUBLE

Ocupa oito bytes e varia entre 5.0E-324 a 1.7E+308 com 15 a 16 dígitos significativos.

EXTENDED

Ocupa dez bytes e varia entre 3.4E-4932 a 1.1E+4932 com 19 a 20 dígitos significativos.

COMP

É um tipo real que só armazena inteiros. Ocupa oito bytes e varia entre -2E+63+1 a 2E+63—1.

CONSTANTES

A única diferença com a versão 3.0 é que agora as constantes tipadas são armazenadas no segmento de dados.

O OPERADOR @

O operador @ (*address of* — endereço do) é usado para o tratamento de alocação de memória dinâmica e retorna o endereço, no formato de ponteiro, de uma variável declarada formalmente. Ele simplifica o uso de ponteiros quando usados para sobrepor outras variáveis na memória.

ENTRADA E SAÍDA

- A versão 3.0 suportava os dispositivos CON:, TRM: E LST:. Na versão 4.0 foram substituídos pelos dispositivos CON ("console": vídeo para saída e teclado para entrada), PRN, LPT1, LPT2 e LPT3 (referentes à impressora).
- O dispositivo LST ainda funciona para saída na impressora se a unidade **Printer** for especificada.
- O Turbo Pascal 4.0 e 5.0 tem a capacidade de escrever diretamente na memória de vídeo ou usar as rotinas do BIOS quando usa a unidade **Crt**. Você controla o método usando a variável **DirectVideo**. Se **DirectVideo** for verdadeira (default), será usado acesso direto, caso contrário serão usadas chamadas ao BIOS.
- A utilização de uma nova função para leitura de um caractere pressionado no teclado. O **ReadKey** retorna um caractere pressionado. Se a tecla gerar um código de "scan", **ReadKey** retornará #0 e uma nova chamada a **ReadKey** lerá o caractere seguinte ao código de "scan".

UNIT DOS

VARIÁVEL PREDEFINIDA DOSERROR (unit DOS)

Conterá um código referente ao erro ocorrido, que será armazenado na variável inteira **DosError**:

0 nenhum erro

2 arquivo não foi encontrado

3 path não encontrado

5 acesso proibido

6 handle inválido

8 não há memória suficiente

10 ambiente inválido

11 formato inválido

18 mais nenhum arquivo

PROCEDIMENTOS DE DATA E HORA

GetDate (var Ano, Mes, Dia, DiaSemana: Word);

Retorna a data do sistema nos parâmetros Ano, Mes, Dia, DiaSemana. O ano contém o século (1988 em vez de 88) e o DiaSemana varia de 0 a 6, sendo o domingo representado pelo 0.

GetFTimes (var F; var Tempo: Longint);

Lê a identificação de tempo do arquivo F. O conteúdo desta identificação, armazenado na variável **Tempo**, contém a data e hora do arquivo. Para descodificar o conteúdo de **Tempo** deve-se usar o procedimento **UnpackTime**.

GetTime (var Hora, Min, Seg, DecSeg: Word);

Retorna a hora do sistema nos parâmetros Hora, Min, Seg, DecSeg.

PackTime (var DT: DateTime; var Tempo: Longint);

Compacta os dados de data e hora para a variável **Tempo**. Isto é usado antes do procedimento **SetFTime**.

Set Date (Ano, Mes, Dia: Word);

Ajusta a data do sistema. O ano deve incluir o século (ex.: 1988). Se os valores dos parâmetros estiverem errados, a data não será corrigida.

SetFTime (var F; Tempo: Longint);

Marca a identificação de tempo do arquivo F. Antes de chamar este procedimento, deve-se criar a variável **Tempo** usando o procedimento **PackTime**.

SetTime (Hora, Min, Seg, DecSeg: Word);

Ajusta a hora do sistema. Se os valores dos parâmetros estiverem errados, a hora não será corrigida.

UnpackTime (Tempo: Longint; var DT: DateTime);

Interpreta a variável **Tempo** que contém data e hora de um arquivo. Este procedimento deve ser usado depois de **GetFTime**.

FUNÇÕES DE CONSULTA SOBRE O ESPAÇO EM DISCO

DiskFree (Drive: Byte): Longint;

Retorna o espaço livre no disco contido em **Drive** (especificado por um número: 0 — unidade em uso, 1 — unidade A, 2 — unidade B etc...).

DiskSize (Drive: Byte): Longint;

Retorna o espaço total do disco contido em **Drive** (especificado por um número: 0 — unidade em uso, 1 — unidade A, 2 — unidade B etc...).

PROCEDIMENTOS PARA ARQUIVOS

GetFAttr (var F; var Atrib: Word);

Retorna o atributo da variável de arquivo F, que já deve ter sido associado (com Assign) mas não aberto.

FindFirst (Path: String; Atrib: Word; var F: SearchRec);

Procura no diretório indicado por Path e retorna o primeiro arquivo normal ou arquivo com atributos que se encaixem naqueles contidos em **Atrib**.

FindNext (var F: SearchRec);

Continua a busca iniciada por **FindFirst**. A variável **DosError** conterá o valor 18 quando não achar mais nenhum arquivo. Se nenhum atributo for indicado (ex.: **Atrib** = 0), estes procedimentos retornarão somente arquivos normais (aqueles sem atributos).

SetFAttr (var F; Atrib: Word);

Ajusta o atributo da variável de arquivo F, que já deve ter sido associado (com Assign) mas não aberto.

PROCEDIMENTOS PARA INTERRUPÇÕES, PROGRAMAS RESIDENTES, SAÍDAS PARA O DOS E OUTROS

DosExitCode: Integer;

Esta função retorna o código de saída de um subprocesso. O byte mais elevado do inteiro será 0 para terminação normal, 1 para terminação forçada por CTRL-C e 3 quando terminada pelo procedimento **Keep**.

DosVersion: Word;

Retorna o número da versão do DOS, separado no byte superior (número da atualização) e no byte inferior (número da versão do DOS).

EnvCount: Integer;

Retorna o número de strings contidos no ambiente DOS.

EnvStr (Índice: Integer): String;

Retorna uma string contida no ambiente DOS. Esta string deve ser indicada pelo **Índice** ao chamar a função e este valor pode variar entre 1 (para a primeira string) até o valor máximo contido em **EnvCount**.

Exec (PathCommandCom, Comando: String);

Permite a saída do seu programa para o sistema operacional. **PathCommandCom** deve indicar o caminho e o nome do Command.Com em seu disco. **Comando** deve conter a ordem a ser dada no sistema operacional. Se **Comando** contiver uma string vazia, o Turbo Pascal deixará executar comandos diretamente no prompt do sistema operacional. Neste caso, para retornar ao programa em Pascal, digite EXIT. É necessário ajustar a memória com a diretiva {**$M**}.

FExpand (Path: PathStr): PathStr;

Esta função retorna o conteúdo do parâmetro **Path** escrito da maneira mais completa possível: com drive, subdiretórios, nome e extensão. O tipo PathStr definido na Unit DOS equivale a string[79].

FSplit (Path: PathStr; var Diretorio: DirStr;
var Nome: NameStr; ver Extensao: ExtStr);

Retorna o nome passado no parâmetro **Path** dividido nas suas três partes básicas: diretório, nome e extensão. Os tipos utilizados já foram definidos na Unit DOS. Veja também a função **FExpand**.

GetCBreak (var Status: Boolean);

Retorna no parâmetro **Status** o estado atual da checagem de CTRL-BREAK feita pelo DOS. Se **Status=true** então a checagem é realizada a cada chamada do sistema, caso contrário (**Status=false**), a checagem só é realizada nas operações de entrada e saída (I/O).

GetEnv (v__amb: string): string;

Retorna a string que contém o valor da variável do ambiente DOS especificada em **v__amb**.

GetIntVec (NumInt: Byte; var Vetor: Pointer);

Retorna o endereço usado atualmente pela interrupção **NumInt** no parâmetro **Vetor**.

GetVerify (var Status: Boolean);

Retorna no parâmetro **Status** o estado atual do flag de verificação do DOS. Se **Status=true** então a verificação para escrita em disco é realizada, caso contrário (**Status=false**), a verificação não é realizada.

Keep (CodSaída: Word);

Permite terminar o programa e deixá-lo residente em memória. **CodSaída** contém o código de saída do DOS depois de terminada a execução do programa.

SetCBreak (Status: Boolean);

Ajusta o estado da checagem de CTRL-Break no DOS através do parâmetro **Status**. Veja também **GetCBreak**.

SetIntVec (NumInt: Byte; Vetor: Pointer);

Troca o endereço atual da interrupção **NumInt** pelo endereço contido em **Vetor**.

SetVerify (Status: Boolean);

Ajusta o estado de verificação de escrita em disco pelo DOS através do parâmetro **Status**. Veja também **GetVerify**.

SwapVectors;

Troca o conteúdo dos **SaveIntXX** pelos vetores de interrupção atuais.

GRÁFICOS DA VERSÃO 4.0 e 5.0

Todos os programas escritos na versão 3.0 podem ser rodados com poucas modificações nas versões 4.0 e 5.0 se usarem a unidade **Graph3**.

Além disto, as versões 4.0 e 5.0 ainda têm uma nova unidade chamada **Graph** com novas rotinas gráficas poderosas e rápidas.

CONSTANTES GRÁFICAS

Códigos retornados em GraphResult

```
grOk              =  0;
grNoInitGraph     = -1;
grNotDetected     = -2;
grFileNotFound    = -3
grInvalidDriver   = -4;
grNoLoadMem       = -5;
grNoScanMem       = -6;
grNoFloodMem      = -7;
grFontNotFound    = -8;
grNoFontMem       = -9;
grInvalidMode     = -10;
grError           = -11;  {Erro genérico}
grIOerror         = -12;
grInvalidFont     = -13;
grInvalidFontNum  = -14;
grInvalidDeviceNum = -15;
```

DEFINE DRIVERS GRÁFICOS

```
CurrentDriver = -128;  {usado em GetModeRange}
Detect        = 0;     {ativa detecção automática da placa em uso}
CGA           = 1;
MCGA          = 2;
EGA           = 3;
```

```
EGA64      = 4;
EGAMono    = 5;
IBM8514    = 6;
HercMono   = 7;
ATT400     = 8;
VGA        = 9;
PC3270     = 10;
```

MODOS GRÁFICOS PARA CADA DRIVER

```
CGACO      = 0;  {320x200 palette 0: Verde Claro,
                  Vermelho Claro, Amarelo; 1 página}
CGAC1      = 1;  {320x200 palette 1: Azul Ciano Claro, Magenta Claro, Branco;
                  1 página}
CGAC2      = 2;  {320x200 palette 2: Verde, Vermelho, Marrom; 1 página}
CGAC3      = 3;  {320x200 palette 3: Azul Ciano, Magenta, Cinza Claro; 1 página}
CGAHi      = 4;  {640x200 1 página}
MCGACO     = 0;  {320x200 palette 0: Verde Claro, Vermelho Claro, Amarelo; 1
                  página}
MCGAC1     = 1;  {320x200 palette 1: Azul Ciano Claro, Magenta Claro, Branco;
                  1 página}
MCGAC2     = 2;  {320x200 palette 2: Verde, Vermelho, Marrom; 1 página}
MCGAC3     = 3;  {320x200 palette 3: Azul Ciano, Magenta, Cinza Claro; 1 página}
MCGAMed    = 4;  {640x200 1 página}
MCGAHi     = 5;  {640x480 1 página}
EGALo      = 0;  {640x200 16 cores 4 páginas}
EGAHi      = 1;  {640x350 16 cores 2 páginas}
EGA64Lo    = 0;  {640x200 16 cores 1 página}
EGA64Hi    = 1;  {640x350 4 cores 1 página}
EGAMonoHi  = 3;  {640x350 64K de memória na placa, 1 página; 256K, 2 páginas}
HercMonoHi = 0;  {720x348 2 páginas}
ATT400C0   = 0;  {320x200 palette 0: Verde Claro, Vermelho Claro. Amarelo;
                  1 página}
ATT400C1   = 1;  {320x200 palette 1: Azul Ciano Claro, Magenta Claro, Branco;
                  1 página}
ATT400C2   = 2;  {320x200 palette 2: Verde, Vermelho, Marrom; 1 página}
ATT400C3   = 3;  {320x200 palette 3: Azul Ciano, Magenta, Cinza Claro; 1 página}
```

```
ATT400Med   = 4;   }640x200 1 página}
ATT400Hi    = 5;   {640x400 1 página}
VGALo       = 0;   {640x200 16 cores 4 páginas}
VGAMed      = 1;   {640x350 16 cores 2 páginas}
VGAHi       = 2;   {640x480 16 cores 1 página}
PC3270Hi    = 0;   {720x350 1 página}
IBM8514LO   = 0;   {640x480 256 cores}
IBM8514HI   = 1;   {1024x768 256 cores}
MaxColors   = 15;
```

ESTILOS DE LINHA E ESPESSURAS PARA Get/ SetLineStyle

```
SolidLn     = 0;
DottedLn    = 1;
CenterLn    = 2;
DashedLn    = 3;
UserBitLn   = 4;   {Estilo de linha definido pelo usuário}
NormWidth   = 1;
ThickWidth  = 3;
```

CONSTANTES PARA Set/ GetTextStyle

```
DefaultFont  = 0;
TriplexFont  = 1;
SmallFont    = 2;
SansSerifFont = 3;
GothicFont   = 4;
HorizDir     = 0   {Da esquerda para a direita}
VertDir      = 1;  {De baixo para cima}
UserCharSize = 0;  {Tamanho de caractere definido pelo usuário}
```

CONSTANTES PARA CLIPPING

```
ClipOn   = true;
ClipOff  = false;
```

CONSTANTES P/ BAR3D

```
TopOn    = true;
TopOff   = false;
```

PADRÕES PARA PREENCHIMENTO COM Get/ SetFillStyle

```
EmptyFill     = 0;   {preenche área na cor de fundo}
SolidFill     = 1;   {preenche área com cores sólidas}
LineFill      = 2;   {padrão ---}
LtSlashFill   = 3;   {padrão / / /}
SlashFill     = 4;   {padrão / / / com linhas grossas}
BkSlashFill   = 5;   {padrão \ \ \ com linhas grossas}
LtBkSlashFill = 6;   {padrão \ \ \}
HatchFill     = 7;   {hachura horizontal}
XHatchFill    = 8;   {hachura diagonal}
InterleaveFill = 9;  {padrão reticulado}
WideDotFill   = 10;  {padrão pontilhado muito espaçado}
CloseDotFill  = 11;  {padrão pontilhado pouco espaçado}
UserFill      = 12;  {padrão definido pelo usuário}
```

OPERADORES BITBLT PARA O PutImage

```
NormalPut  = 0;  {MOV}
CopyPut    = 0;  {MOV}
XORPut     = 1;  {XOR}
OrPut      = 2;  {OR}
AndPut     = 3;  {AND}
NotPut     = 4;  {NOT}
```

JUSTIFICAÇÃO HORIZONTAL E VERTICAL PARA SetTextJustify

```
LeftText   = 0;
CenterText = 1;
RightText  = 2;
```

```
BottomText   = 0;
TopText      = 2;
```

NOVOS PROCEDIMENTOS E FUNÇÕES GRÁFICAS

(Necessitam de **Uses Graph** no programa e drivers .BGI no diretório atual ou na memória.)

FUNÇÕES DE MANUTENÇÃO DE ERROS

GraphErrorMsg (CodigoErro: integer): String;

Retorna uma string contendo uma mensagem para o erro indicado por **CodigoErro.**

GraphResult: integer;

Retorna um código de erro para a última operação realizada.

PROCEDIMENTOS E FUNÇÕES PARA DETECÇÃO, INICIALIZAÇÃO E MODOS DE VÍDEO

CloseGraph;

Volta a tela ao modo anterior e retira do *heap* todo as variáveis dinâmicas usadas pelo modo gráfico.

DetectGraph (var DriverGraf, ModoGraf: integer);

Analisa qual placa de vídeo está sendo usada e recomenda o modo gráfico mais adequado.

GetAspectRatio (var Xasp, Yasp: word);

A razão de aspecto é a razão entre os pixels horizontais e verticais que produz uma imagem bem proporcionada na tela. Depois de chamar **GetAspectRatio**, divida **Xasp** por **Yasp** para determinar a razão de aspecto do seu equipamento.

GetDriverName: String;

Esta função retorna uma string com o nome do driver atualmente em uso.

GetGraphMode: integer;

Esta função retorna um valor inteiro que indica o modo gráfico atualmente em uso.

GetMaxMode: Word;

Retorna o número máximo do modo do driver gráfico em uso.

GetMaxX: integer;

Função que retorna o valor da coordenada horizontal do pixel no canto mais à direita.

GetMaxY: integer;

Função que retorna o valor da coordenada vertical do pixel no canto inferior.

GetModeName (Numero: Word): string;

Retorna uma string com o nome do modo indicado pelo parâmetro **Numero**.

GetModeRange (DriverGraf: integer; var LoMode, HiMode: integer);

Retorna os modos máximo e mínimo para um driver gráfico especificado.

GetPalette (var Pal: PaletteType);

Preenche os campos da variável registro **Pal** de tipo predefinido PaletteType com o tamanho e as cores atuais da palette.

GetPaletteSize: word;

Retorna o tamanho da palette.

GetX: integer;

Função que retorna a coordenada horizontal da posição atual, sempre relativa ao **viewport** em uso. Isto quer dizer que se **GetX** retornar 0, a posição será no lado esquerdo do **viewport** e não no lado esquerdo da tela.

GetY: integer;

Função que retorna a coordenada vertical da posição atual, sempre relativa ao **viewport** em uso. Isto quer dizer que se **GetY** retornar 0, a posição será no topo do **viewport** e não no topo da tela.

GraphDefaults;

Muda a posição atual para o topo esquerdo da tela (0,0) e inicializa todos os procedimentos gráficos (viewport, palette, cores, características das linhas, textos gráficos etc.).

InitGraph (var DriverGraf: integer;
var ModoGraf: integer;
DriverPath: String);

Inicializa o ambiente gráfico e entra em modo gráfico. Se o **DriverGraf** for **Detect** (0), o **InitGraph** vai determinar qual driver e qual modo devem ser usados. **DriverPath** indica qual o caminho (*path*) para o diretório onde os drivers gráficos se encontram. Deixa a posição atual no topo esquerdo da tela (0,0).

SetGraphMode (Modo: integer);

Ajusta o ambiente gráfico para o **Modo** gráfico escolhido. Deixa a posição atual no topo esquerdo da tela (0,0).

RestoreCrtMode;

Retorna ao modo da tela anterior ao modo gráfico.

ROTINAS DE TELA E ViewPort

ClearDevice;

Apaga a tela atual e deixa a posição atual no topo esquerdo do viewport (0,0).

ClearViewPort;

Limpa o viewport e assume as cores da palette(0) para a tela. Deixa a posição atual no topo esquerdo da tela (0,0).

GetViewSettings (var ViewPort: ViewPortType);

Retorna uma variável registro predefinida, contendo os dados do viewport atual.

SetActivePage (Pagina: word);

Direciona toda saída da tela para a página indicada. Para tornar a página visível, use o procedimento **SetVisualPage**.

SetViewPort (x1, y2, x2, y2: integer; Clip: boolean);

Define uma porção da tela, de x1, y1 (canto superior esquerdo) até x2, y2 (canto inferior direito). Se **Clip** for verdadeiro, o *clipping* estará ligado. Depois do **viewport** ligado, todas as coordenadas estarão relacionadas com o **viewport** e não com as coordenadas físicas da tela. Deixa a posição atual no topo esquerdo (0,0).

SetVisualPage (Pagina: word);

Troca a página gráfica visualizada na tela por aquela indicada na variável **pagina**.

ROTINAS DE MANUSEIO DE PIXELS

PutPixel (X, Y: integer; Cor: word);

Coloca um pixel na coordenada x, y. A cor é determinada pela variável **cor**.

GetPixel (X, Y: integer): word;

Esta função retorna o valor da cor do pixel na coordenada x, y.

ROTINAS PARA CRIAÇÃO DE LINHAS

GetLineSettings (var InfoLinha: LineSettingsType);

Retorna em **InfoLinha** (variável registro) três dados que determinam a aparência da linha desenhada.

Line (x1, y1, x2, y2: integer);

Desenha uma reta da coordenada x1, y1 até x2, y2.

LineTo (X, Y: integer);

Desenha uma linha da posição atual até x, y.

LineRel (Dx, Dy: integer);

Desenha uma linha da posição atual até a posição relativa definida por Dx, Dy.

MoveRel (Dx, Dy: integer);

Move a posição atual para uma nova posição deslocada Dx, Dy da original.

MoveTo (X, Y: integer);

Move a posição atual para a coordenada x, y.

SetLineStyle (EstiloLinha: word;
Padrao: word;
Espessura: word);

Define o estilo da linha, o padrão e a espessura a serem usados ao desenhar uma linha.

ROTINAS PARA CRIAÇÃO DE POLÍGONOS, FIGURAS E PREENCHIMENTO DE ÁREAS

Bar (x1, y1, x2, y2: integer);

Desenha um retângulo da coordenada x1, y1 (canto superior esquerdo) até a coordenada x2, y2 (canto inferior direito), preenchendo-o com o padrão (*pattern*) selecionado com **SetFillStyle**.

Bar3D (x1, y1, x2, y2: integer; Prof: word; Topo: boolean);

Desenha um retângulo da coordenada x1, y1 (canto superior esquerdo) até a coordenada x2, y2 (canto inferior direito), preenchendo-o com o padrão (*pattern*) selecionado com **SetFillStyle**. A variável **Prof** indicará a profundidade em perspectiva isométrica e **Topo** indicará se o topo do retângulo deve ser aberto ou fechado (use as constantes TopOn e TopOff).

DrawPoly (NumPontos: word; var PontosPolig);

Desenha uma poligonal que liga os pontos definidos na matriz **PontosPolig** (matriz tipo registro com campos para X e Y do tipo word). A quantidade de pontos é definida em **NumPontos**. Em caso de coordenadas erradas na matriz, **GraphResult** retornará o valor -6.

GetFillPattern (var PadraoFill: FillPatternType);

Retorna o último padrão usado por **SetFillPattern**.

GetFillSettings (var InfoFill: FillSettingsType);

Retorna um registro do tipo **FillSettingsType** contendo dois parâmetros **Pattern** e **Color**, que indicam o tipo de hachura e a cor atuais.

FillPoly (NumPontos: word; var PontosPolig);

Desenha uma poligonal que liga os pontos definidos na matriz **PontosPolig** (matriz tipo registro com campos para X e Y do tipo word), preenchendo-a com o padrão e a cor definidas por **SetFillPattern**, **SetFillStyle** e **SetColor**. A linha que limita o polígono é desenhada de acordo com o estilo de linha e cor atuais e deve estar perfeitamente fechada. A quantidade de pontos no polígono é definida em **NumPontos**. Em caso de coordenadas erradas na matriz, **GraphResult** retornará o valor -6.

FloodFill (X, Y: integer; Border: word);

Preenche a área fechada pela linha de cor **Borda**. Esta área deve conter o ponto de coordenada x, y.

Rectangle (x1, y1, x2, y2: integer);

Desenha um retângulo da coordenada x1, y1 (canto superior esquerdo) até a coordenada x2, y2 (canto inferior direito), usando o estilo de linha e cor atuais.

SetFillPattern (Padrao: FillPatternType; Cor: word);

Define um padrão e uma cor a serem usados em preenchimento de áreas gráficas. **Padrao** é uma variável do tipo predefinido FillPatternType (uma matriz de bytes com 8 elementos) que gera um padrão de 8x8 pixels, sendo um pixel para cada bit do byte.

SetFillStyle (Padrao: word; Cor: word);

Escolhe uma cor e um padrão de preenchimento (12 predefinidos indo de 0 a 11).

SetGraphBufSize (TamBuf: word);

Permite mudar o tamanho do buffer (**TamBuf**) reservado para preenchimento de áreas.

ROTINAS DE ARCOS, CÍRCULOS E OUTRAS CURVAS

Arc (X, Y: integer; AngInic, AngFinal, Raio: word);

Desenha um arco com centro em x, y, de raio especificado em **Raio**. O arco começa em **AngInic** e termina em **AngFinal**, sendo desenhado sempre no sentido anti-horário (ângulo 0 equivale a leste ou 3:00 horas).

Circle (X, Y: integer; Raio: word);

Desenha um círculo de centro em x, y e raio especificado.

Ellipse (X, Y: integer;
AngInic, AngFinal: word;
XRaio, YRaio: word);

Produz uma elipse de centro em x, y. O desenho começa em **AngInic** e termina em **AngFinal** e a elipse torna-se mais alongada quanto for a diferença entre **XRaio** e **YRaio**. Além disto, é desenhada sempre no sentido anti-horário (ângulo 0 equivale a leste ou 3:00 horas).

GetArcCoords (var CoordArco: ArcCoordsType);

Retorna uma variável registro, contendo dados sobre o último arco criado pelo procedimento **Arc**, dados estes que podem ser usados para criar uma "fatia" em gráficos de setores (tipo torta), ao ligar linhas do centro do arco para as suas extremidades.

PieSlice (X, Y: integer; AngInic, AngFinal, Raio: word);

Desenha uma "fatia" para um gráfico de setores, com centro em x, y e raio determinado em **Raio**. O desenho começa em **AngInic** e termina em **AngFinal**, sempre em sentido anti-horário (ângulo 0 equivale a leste ou 3:00 horas). A "fatia" será preenchida com o padrão e a cor definidas em **SetFillStyle** e **SetFillPattern**.

Sector (X, Y: integer;
AngInic, AngFinal: word;
XRaio, YRaio: word);

Produz um setor de elipse de centro em x, y. O desenho começa em **AngInic** e termina em **AngFinal**. **XRaio** e **YRaio** são os raios da elipse. Além disto, o setor é desenhado sempre no sentido anti-horário (ângulo 0 equivale a leste ou 3:00 horas) e será colorido e preenchido de acordo com **SetFillPattern** e **SetFillStyle**.

ROTINAS PARA PALETTES E CORES

GetBkColor: word;

Esta função retorna um número que indica a cor de fundo atual.

GetColor: word;

Esta função retorna um número que indica a cor atual.

GetMaxColor: word;

Esta função retorna o número máximo da cor que pode ser passada como parâmetro em **SetColor**.

GetPalette (var Palette: PaletteType);

Retorna os valores atuais da palette numa variável tipo registro, predefinida com dois campos: **Size**, que indica o número de cores na palette e **Colors**, que contém as cores da palette.

SetAllPalette (var Palette);

Troca os dados da palette atual por aqueles contidos na variável registro predefinida **Palette**.

SetBkColor (Cor: word);

Escolhe a cor de fundo indicada por **Cor**. SetBkColor (0) muda a cor de fundo para preto.

SetColor (Cor: word);

Deixa a cor para desenho valendo **Cor**.

SetPalette (NumCor: word; Cor: shortint);

Altera a cor de número **NumCor** para a definida em **Cor**, permitindo alterar o padrão da palette.

SetRGBPalette (Num_cor, Verm, Verd, Azul: Integer);

Permite alterar a palette usada pelos drivers VGA e IBM-8514.

ROTINAS DE MANUSEIO DE BLOCOS DE PIXELS

GetImage (x1, y1, x2, y2: integer; var Figura);

Captura a imagem contida em x1, y1 (canto superior esquerdo) e x2, y2 (canto inferior direito) no buffer **Figura**. Este buffer tem de ser pelo menos do tamanho definido "mais quatro". Use a função **ImageSize** para saber o tamanho necessário.

ImageSize (x1, y1, x2, y2: integer): word;

Esta função retorna a quantidade de memória necessária para armazenar a imagem definida em x1, y1 e x2, y2.

PutImage (X, Y: integer; var Figura; BitBlt: word);

Transfere a imagem contida no buffer **Figura** para a tela na coordenada x, y, sem fazer "clipping" (ou seja: se a imagem sair fora do limite, não será transferida). **BitBlt** indica como a imagem deve ser transferida de acordo com as constantes predefinidas NormalPut, CopyPut, XORPut, AndPut e NotPut.

ROTINAS PARA TEXTOS NO MODO GRÁFICO

GetTextSettings (var InfoText: TextSettingsType);

Retorna os dados a respeito do estilo do tipo de letra atual. Os valores são retornados dentro de uma variável registro de tipo predefinido e dizem respeito a tipo escolhido, direção, tamanho e justificação horizontal ou vertical.

OutText (StringTexto: string);

Escreve a string contida em **StringTexto** na posição atual da tela gráfica.

OutTextXY (X, Y: integer; StringTexto: string);

Escreve a string contida em **StringTexto** na posição x, y da tela gráfica. Não atualiza a posição atual e não faz "clipping" com caracteres gráficos default.

SetTextJustify (Horiz, Vert: word);

Determina a maneira como o texto será escrito em relação à posição atual. Existem constantes predefinidas para valores horizontais (LeftText, CenterText e RightText) e verticais (BottomText, CenterText e TopText).

SetTextStyle (Font, Direcao: word; TamCarac: word);

Define a maneira como o texto será mostrado na tela. Existem cinco tipos de letras disponíveis (0 a 4, sendo que do 1 ao 4 ficam armazenados em disco). Existem constantes predefinidas para serem usadas como valor para **Font** (DefaultFont, TriplexFont, SmallFont, SansSerifFont e GothicFont), outras para **Direcao** (HorizDir e VertDir) e também para o tamanho do caractere dado por **TamCarac** (NormSize).

SetUserCharSize (MultX, DivX, MultY, DivY: byte);

Permite alterar o comprimento ou a altura para alguns tipos de letras.

SetWriteMode (Modo: Integer);

Ajusta o modo de escrita de acordo com o **Modo** indicado: CopyPut (valor zero) ou XORPut (valor 1);

TextHeight (StringTexto: string): word;

Esta função retorna a altura em pixels da string contida em **StringTexto**. A altura depende da fonte em uso.

TextWidth (StringTexto: string): word;

Esta função retorna o comprimento em pixels da string contida em **StringTexto** (depende da fonte em uso).

FUNÇÕES ESPECIAIS

RegisterBGIfont (font: pointer): integer;

Esta função retorna um número inteiro que pode indicar se houve um erro (número negativo) ou retorna o número do tipo de letra atual.

RegisterBGIdriver (driver: pointer): integer;

Esta função retorna um número inteiro que pode indicar se houve um erro (número negativo) ou retorna o número do driver atual.

OVERLAYS

(Necessitam de **Uses Overlay** no programa.)

OvrClearBuf;

Limpa o buffer reservado para *overlays*, forçando o carregamento em memória das outras *overlays*.

OvrGetBuf: LongInt;

Esta função retorna o tamanho atual do buffer para *overlays*.

OvrInit (Nome_arq: String);

Inicializa o gerenciador de *overlays* do Turbo Pascal e abre o arquivo *overlay* indicado pelo parâmetro **Nome_arq.**

OvrInitEMS;

Permite carregar o arquivo *overlay* para a expansão de memória tipo EMS.

OvrSetBuf (Tam: LongInt);

Permite ajustar o tamanho do buffer de **overlays** para aquele indicado em **Tam**.

MENSAGENS DE ERRO DO TURBO 5.0

(As mensagens de erro variam a cada versão.)

ERROS NA COMPILAÇÃO

1 Acabou espaço na memória.
2 Identificador esperado.
3 Identificador desconhecido.
4 Identificador repetido (já existente).
5 Erro de sintaxe.
6 Erro na sintaxe da constante real.
7 Erro na sintaxe da constante inteira.
8 Constante tipo String excede a linha.
9 Muitos arquivos aninhados (um dentro do outro).
10 Fim de arquivo inesperado.
11 Linha muito longa.
12 Identificador de tipo esperado.
13 Muitos arquivos abertos.
14 Nome de arquivo inválido.
15 Arquivo não foi encontrado.
16 Disco cheio.
17 Diretiva de compilação inválida.
18 Muitos arquivos.
19 Tipo indefinido na definição de um ponteiro (pointer).
20 Identificador de variável esperado.
21 Erro no tipo.
22 Estrutura muito grande.
23 Tipo base de um conjunto está fora do limite.
24 Componentes do arquivo não podem ser outros arquivos.
25 Comprimento da String é inválido.

26 Tipos incompatíveis.
27 Faixa de valores inválidas para o tipo base.
28 Limite inferior maior que limite superior.
29 Tipo ordinal (escalar) esperado.
30 Constante inteira esperada.
31 Constante esperada.
32 Constante inteira ou real esperada.
33 Identificador de tipo esperado.
34 Tipo de resultado da função é inválido.
35 Identificador de Label esperado.
36 BEGIN esperado.
37 END esperado.
38 Expressão inteira esperada.
39 Expressão ordinal (escalar) esperada.
40 Expressão booleana esperada.
41 Tipo do operando não corresponde ao tipo do operador.
42 Erro na expressão.
43 Associação ilegal.
44 Identificador de campo esperado.
45 Arquivo-objeto muito grande.
46 Procedimento ou Função externa indefinida.
47 Registro inválido no arquivo-objeto.
48 Segmento de código muito grande.
49 Segmento de dados muito grande.
50 DO esperado.
51 Definição PUBLIC inválida.
52 Definição EXTRN inválida.
53 Muitas definições EXTRN.
54 OF esperado.
55 INTERFACE esperado.
56 Referência relocável inválida.
57 THEN esperado.
58 TO ou DOWNTO esperado.
59 Forward indefinido.
60 Muitos procedimentos.
61 Typecast inválido.
62 Divisão por zero.
63 Tipo de arquivo inválido.

64 Não é possível usar variáveis deste tipo em READ ou WRITE.
65 Variável ponteiro (*pointer*) esperada.
66 Variável String esperada.
67 Expressão String esperada.
68 Unit não foi encontrada.
69 Nome da Unit incompatível.
70 Versão incompatível da Unit.
71 Nome duplicado (repetido) da Unit.
72 Erro no formato do arquivo da Unit.
73 IMPLEMENTATION esperado.
74 Tipos incompatíveis entre o seletor e as constantes do CASE.
75 Variável registro esperada.
76 Constante fora do limite.
77 Variável arquivo esperada.
78 Expressão ponteiro (*pointer*) esperada.
79 Expressão inteira ou real esperada.
80 Label não está presente no bloco atual.
81 Label já foi definido.
82 Label indefinido na área anterior dos comandos.
83 Argumento @ inválido.
84 UNIT esperado.
85 "**;**" esperado.
86 "**:**" esperado.
87 "**,**" esperada.
88 "**(**" esperado.
89 "**)**" esperado.
90 "**=**" esperado.
91 "**:=**" esperado.
92 "**[**" ou "**(.**" esperado.
93 "**]**" ou "**.)**" esperado.
94 "**.**" esperado.
95 "**..**" esperado.
96 Variáveis demais.
97 Variável de controle do FOR é inválida.
98 Variável inteira esperada.
99 Arquivos não são permitidos aqui.
100 Comprimento da String é incompatível.
101 Ordenação inválida dos campos.

102 Constante String esperada.
103 Variável inteira ou real esperada.
104 Variável ordinal (escalar) esperada.
105 Erro no INLINE.
106 Expressão do tipo caractere esperada.
107 Itens de relocação demais.
108 Não há memória suficiente para executar o programa.
109 Não consegue achar arquivo EXE.
110 Não pode executar uma Unit.
111 Compilação abortada.
112 Constante do CASE fora do limite.
113 Erro no comando.
114 Não pode chamar um procedimento Interrupt.
115 O co-processador 80x87 é necessário para compilar isto.
116 Precisa estar no modo 80x87 para compilar isto.
117 Endereço destino não encontrado.
118 Inclusão de arquivos não é permitida aqui.
119 Erro no formato do arquivo TPM.
120 NIL esperado.
121 Qualificador inválido.
122 Referência inválida à variável.
123 Símbolos demais.
124 Área dos comandos muito grande.
125 Módulo não tem informação para debug (depuração).
126 Arquivos devem ser parâmetros VAR (passagem de parâmetros por referência).
127 Símbolos condicionais em excesso.
128 Diretiva condicional em lugar errado.
129 Diretiva ENDIF faltando.
130 Erro nas definições condicionais iniciais.
131 Cabeçalho incompatível com definição anterior.
132 Erro crítico no disco.
133 Não é possível avaliar esta expressão.
134 Expressão terminada incorretamente.
135 Especificador de formato inválido.
136 Referência indireta inválida.
137 Variáveis estruturadas não são permitidas aqui.
138 Avaliação não é possível sem a Unit System.
139 Não é permitido acessar este símbolo.

140 Operação de ponto flutuante inválida.
141 Não é possível compilar *overlays* na memória.
142 Variável procedure ou função esperada.
143 Referência inválida a procedure ou função.
144 Não é possível tornar esta unit num *overlay*.

ERROS NA EXECUÇÃO (RUN-TIME ERRORS)

— erros de 1 a 99 equivalem aos códigos de erro do DOS;
— erros de 100 a 149 equivalem aos erros de E/S (I/O);
— erros de 150 a 199 são erros críticos;
— erros de 200 a 255 são erros fatais.

2 Arquivo não encontrado.
3 Path (caminho) não encontrado.
4 Muitos arquivos abertos.
5 Negado acesso ao arquivo.
6 Handle inválido para o arquivo.
12 Código inválido de acesso ao arquivo.
15 Número inválido do drive.
16 Não pode remover o diretório atual.
17 Não pode trocar nomes entre discos.
100 Erro na leitura do disco.
101 Erro na gravação em disco.
102 Arquivo não associado.
103 Arquivo fechado.
104 Arquivo fechado para entrada.
105 Arquivo fechado para saída.
106 Formato numérico inválido.
150 Disco protegido contra gravação.
151 Unit desconhecida.
152 Drive não está pronto.
153 Comando desconhecido.
154 Erro CRC nos dados.
156 Erro na procura (*seek*) em disco.
157 Tipo de meio desconhecido.
158 Setor não encontrado.

159 Impressora sem papel.
160 Falha no dispositivo de impressão.
161 Falha no dispositivo de leitura.
162 Falha do Hardware.
200 Divisão por zero.
201 Erro na checagem de faixa.
202 Erro de *overflow* no Stack.
203 Erro de *overflow* no Heap.
204 Operação de *pointer* (ponteiro) inválida.
205 *Overflow* de ponto flutuante.
206 *Underflow* na operação de ponto flutuante.
207 Operação de ponto flutuante inválida.
208 Gerenciador de *overlays* não foi instalado.
209 Erro na leitura do arquivo *overlay*.

OUTROS LIVROS NA ÁREA

Carroll – Programação em Turbo Pascal

Collins – Programação Estruturada com Estudos de Casos em Pascal

Jamsa – Turbo Pascal 4 – Guia de Referência Básica

Renzetti – Turbo Pascal – Comandos Básicos – Guia do Operador

Wood – Turbo Pascal – Guia do Usuário

0-07-450396-0