Byte / McGraw-Hill

dBASE II Completo - Total

GUIA DO USUÁRIO

- CONCEITOS TÉCNICOS
- MODO INTERATIVO
- PROGRAMAÇÃO
- EXEMPLOS DE PROGRAMAS
- CONCEITOS AVANÇADOS
- ESTILO DE PROGRAMAÇÃO

McGraw-Hill

LAN BARNES
MICROTREND INC.

dBASE II

Completo - Total

GUIA DO USUÁRIO

BR 040 - Trevo de Nova Lima
Loja NL 34 - 225-90 8

dBASE II
Completo - Total
GUIA DO USUÁRIO

Lan Barnes
Microtend, Inc.

Tradução
Denise Taboas de Souza

Revisão Técnica
José Cláudio Boccia
Analista Consultor de Sistemas

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
Introducing dBASE II

Copyright © 1985 by McGraw-Hill, Inc.
Copyright © 1986 da Editora McGraw-Hill, Ltda.

Todos os direitos para a língua portuguesa reservados pela Editora McGraw-Hill, Ltda.

Nenhuma parte desta publicação poderá ser reproduzida, guardada pelo sistema "retrieval" ou transmitida de qualquer modo ou por qualquer outro meio, seja este eletrônico, mecânico, de fotocópia, de gravação, ou outros, sem prévia autorização, por escrito, da Editora.

Editor Chefe: Milton Mira de Assumpção Filho

Editor: Denise Taboas de Souza

Coordenadora de Revisão: Daisy Pereira Daniel

Supervisor de Produção: Edson Sant'Anna

Capa: *Layout:* Cyro Giodano
Arte final: Ademir Aparecido Alves

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

Barnes, Lan.
B241d dBASE II (TM): completo-total: guia do usuário / Lan Barnes; tradução Denise Taboas de Souza; revisão técnica João Cláudio Boccia. – São Paulo: MacGrw-Hill, 1986.

1. dBASE II (Programa de computador) I. Título.

86-1083 CDD-001.6425

Índices para catálogo sistemático:

1. dBASE II: Computadores: Programas: Processamento de dados 001.6425

NOTA DOS EDITORES

As técnicas de utilização do dBASE II, por suas próprias características, podem ser divididas em vários grupos, o que facilita a abordagem do assunto. Neste livro, o autor procurou abranger todos os níveis, fazendo uma evolução clara e detalhada, conduzindo o leitor, iniciante ou não, a um aprendizado natural da linguagem.

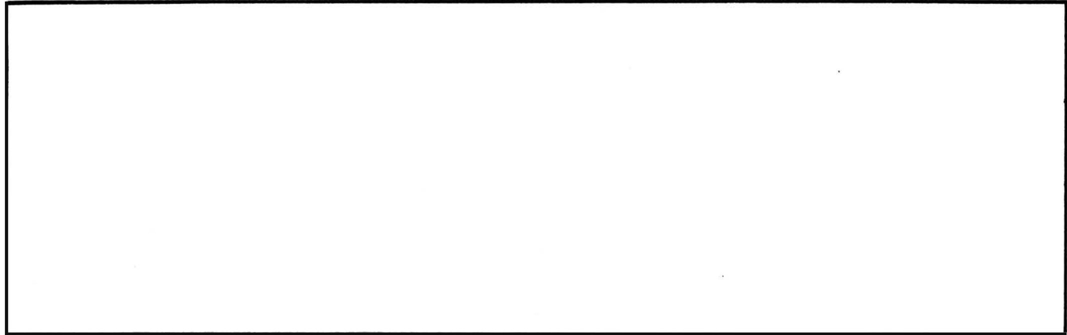
Os comandos interativos são amplamente discutidos, de tal forma a se constituir numa rica fonte de ensinamentos aos principiantes na linguagem, do mesmo modo conseguindo tirar as eventuais dúvidas daqueles que já a conhecem.

Os capítulos que tratam da programação, começam com exemplos de montagens de estruturas simples, passam por discussão sobre estilos de montagens de programas e terminam, com abordagens claras sobre a utilização de comandos de maior grau de complexidade.

Discussões detalhadas sobre o desenho de bancos de dados e técnicas de montagens de sistemas fazem parte do texto, de tal forma a permitir que este livro seja considerado Completo e Total.

dBASE II - Guia do Usuário representa mais uma contribuição bibliográfica a este poderoso e tradicional software, e com certeza enriquecerá o conhecimento técnico e a capacidade profissional de seus Usuários.

SUMÁRIO



Introdução IX

Capítulo 1
Uma Visão Geral Sobre Bancos de Dados 1

Capítulo 2
Conceitos de Banco de Dados 10

Capítulo 3
O Modo Interativo (I) 29

Capítulo 4
O Modo Interativo (II) 80

Capítulo 5
As Funções e Expressões do dBase 116

Capítulo 6
O Controle das Estruturas 142

Capítulo 7
Programação de Entrada/Saída 182

Capítulo 8
Relacionando Arquivos de Dados 206

Capítulo 9
Considerações Sobre o Desenho de Banco de Dados 241

Capítulo 10
A Depuração 262

Capítulo 11
Comandos e Técnicas Avançadas 282

Apêndice A
Um Resumo dos Comandos dBase 299

Apêndice B
Funções do Sistema 302

Apêndice C
Códigos de Movimentação do Cursor Quando em Tela-Cheia 304

Apêndice D
Mensagens de Erro 306

Apêndice E
Tabela de Conversão de Decimal para ASCII 309

Apêndice F
Limites e Limitações do dBASE 310

Índice Analítico 311

INTRODUÇÃO

Nunca vivemos sem dados e bancos de dados. Os arqueólogos dizem que cuneiformes e chapas de barro, hieróglifos em papiros e ruínas em escavações eram todos para registrar quantos itens estavam estocados, quantos vinhos existiam na adega e quem devia o que a quem.

É consolador, embora frustrante, ver que as pessoas há 3.000 anos viviam de maneira tão convencional quanto a nossa. A civilização difunde-se através de registros dos acontecimentos. Cada um de nós deixa registros oficiais para anos que virão – certidões de nascimento, boletins escolares transcritos, registros militares, históricos de empregos, contas bancárias, referências de crédito e seguro social; a lista é infinita. Até recentemente, todos estes registros eram manipulados pelas secretárias, empregados e burocratas, ou por muitos computadores que de tão caros não podiam ser usados por qualquer pessoa, a não ser pelo governo ou pelas grandes potências do mundo dos negócios.

Hoje, uma verdadeira revolução tecnológica está acontecendo. Com a criação dos microcomputadores baratos ou os computadores "pessoais", tornou-se possível para quase todas as pessoas terem acesso à manipulação de dados com as poderosas ferramentas eletrônicas – donas-de-casa, executivos, estudantes, pessoas que possuem *hobbies* e os pequenos comerciantes. Os computadores avançaram tão rapidamente que, hoje, os donos dos computadores pessoais possuem mais capacidade e velocidade utilizável em suas próprias casas do que a maioria dos governos possuía há 20 anos.

Porém, a revolução dos microcomputadores provavelmente confundiu os usuários em relação aos dados, sua definição e o que ele possui para nos auxiliar. Em nenhuma outra área isto é mais evidente do que em gerenciamento de banco de dados. A maioria dos comerciantes e gerentes possui um bom senso intuitivo de qual dado é usado em seu escritório, como ele é projetado e quais decisões são tomadas a partir dele. Nosso consolo em relação aos dados vão além da expectativa quando começamos a manipular os sistemas de banco de dados do computador básico. Existem muitas razões para isto, mas duas são particularmente notórias.

Em primeiro lugar, as colocações sobre bancos de dados em livros de computação são geralmente feitas em termos abstratos. Em segundo, os bancos de dados computadorizados não usam registros em papel no processamento de dados.

Onde antigamente manipulávamos uma conta de empregado com contas ou créditos e débitos de clientes em um livro razão, agora "gerenciamos um dado comum". Ainda pior, os arquivos de aço já não existem mais. Para responder perguntas diárias que surgem em cada escritório, os empregados têm de usar a tela e o teclado assim como para obter uma organização de dados que não podem ser folheados com olhos e dedos. Não é de surpreender que, hoje em dia, muitas pessoas culpam os computadores por confusões causadas.

– "Desculpe, mas nosso computador pifou", ou – "Seu pagamento deve estar perdido no computador".

Com a chegada do dBase II e outros sistemas gerenciadores de bancos de dados para microcomputadores, provavelmente mudará esta situação para melhor. Cada vez mais, pessoas comuns projetam e criam seus próprios sistemas de bancos de dados. A conscientização de nossa sociedade sobre o que são dados e quais são seus valores e riscos crescerá. E como estamos alcançando um estágio mais sofisticado, provavelmente também nos conscientizaremos de não mais responsabilizarmos a ineficiência dos computadores.

O dBASE II COMO LINGUAGEM DE PROGRAMAÇÃO

O dBase II foi criado pela Ashton-Tate como sendo uma linguagem de banco de dados relacional, e esta afirmação é verdadeira até certo ponto. Chamar o dBase de uma linguagem de programação e nada mais é ter uma visão restrita, injusta tanto para o dBase como para seu novo usuário.

dBase II pode ser usado em vários níveis diferentes, para realizar coisas qualitativamente diferentes. Por exemplo, quando um usuário tem um problema com relação aos dados ou uma parte dos dados para gerenciar, dBase pode ser usado como um programa "processador de dados" para classificar e manipular os dados e fazer relatórios específicos. Quando usado desta forma, dBase é um programa de aplicação. Possui a mesma relação com os dados do usuário como um bom programa processador de texto possui com o texto do usuário – o usuário fornece o cérebro, seguindo "um por um" seus dados, e dBase é apenas um instrumento para realizar o trabalho. Os usuários de dBase distinguem isto da programação, chamando de *nível de comando em dBase.*

PARA QUE SERVE ESTE LIVRO

Este livro foi escrito para ensinar os novos usuários como usar o dBase II como uma linguagem de programação e como uma programação de aplicação em nível de comando. Os assuntos foram organizados para que o leitor em qualquer nível e com experiência em dBase possa utilizar o livro de maneira apropriada.

Os Capítulos 1 e 2 trazem uma visão geral sobre a terminologia e a organização dos bancos de dados. Estes capítulos são curtos e de forma alguma substituem outros livros de compreensão

sobre a teoria e o desenho de bancos de dados, mas ajudarão os novos usuários a iniciarem estas aplicações. Eles introduzem suficientemente o assunto para tornar o restante do livro compreensível.

Nos Capítulos 3, 4 e 5 apresentamos exercícios em dBase, e seria mais interessante acompanhar sua leitura fazendo rodar o dBase em um computador. Nestes capítulos, os 40 ou mais comandos mais usados normalmente em programas dBase são descritos e demonstrados. Os capítulos do meio estão divididos em "operações", exercícios que não levarão mais de uma ou duas horas para serem efetuados.

Cada um dos Capítulos 6, 7 e 8 contém um tópico específico central da programação dBase – os controles das estruturas, o programa I/O (entrada e saída) e o processo de relacionar múltiplos arquivos em um só banco de dados. Talvez o programador experiente em dBase queira ir direto a estes capítulos e um outro em qualquer nível poderá revê-los várias vazes. Finalmente, os últimos Capítulos – 9, 10 e 11 – referem-se a tópicos avançados, determinados, como o projeto do programa global, depuração e técnicas avançadas.

CAPÍTULO

1

Uma Visão Geral Sobre Bancos de Dados

Este capítulo foi escrito a fim de fornecer uma visão geral do que é um banco de dados e como trabalhar com o dBase II em um microcomputador. Devemos prevenir que os conceitos e as definições aqui apresentados foram simplificados. Nosso objetivo é apresentar conceitos que permitam o fácil entendimento dos bancos de dados do dBase II, e não ensinar um curso precisamente correto no desenho ou teoria de bancos de dados.

O QUE É UM BANCO DE DADOS?

Para nossas finalidades, dados podem ser definidos como dois ou mais itens de uma informação efetiva que possui uma relação definível entre si. Vamos considerar um exemplo. Imagine uma tela de um computador (ou um pedaço de papel) com o número "32767" sem classificação. Este número pode ou não ter um significado útil para nós, mas sem classificação e sem contexto é impossível dizer. É um número de um endereço? Uma dívida? Um número de um telefone mexicano? No caso de ser qualquer uma de nossas suposições, a quem ou a que ele pertence? Por si só o número 32767 não é um item de dado prático.

Agora consideremos esta matriz de dois itens:

CÓDIGO POSTAL 32767

O número agora possui uma classificação. Os dois itens de fato descritos, um número e sua classificação, possuem algum significado que está implícito em sua descrição. O código de área postal 32767 significa no mundo real, Flórida, Lake Paisley. Neste exemplo corriqueiro, a matriz do número e da classificação constitui um banco de dados mínimo que contém um fato significativo.

VELOCIDADE DE RECUPERAÇÃO

Geralmente, os dados que são relativamente constantes, como códigos postais e números de telefones, são compilados em listagens, dicionários ou outros livros de consulta, e normalmente não pensamos em tais livros como bancos de dados. Os livros de consulta representam um meio excelente para armazenar este tipo de dado. Na verdade, colocar um dado como este dentro de um computador é uma escolha pobre em relação ao papel na maioria dos casos. Para nós, é impraticável registrar todos os códigos postais em um computador para obter um acesso mais rápido, desde que precisamos procurar menos que dez novos códigos postais por ano, e cada vez que o fazemos, devemos ligar o computador, acionar o programa de códigos postais, digitar o código postal para pesquisar e assim por diante. Para os que trabalham com códigos postais, porém, a possibilidade de encontrar um código postal imediatamente é muito útil, e eles possuem os computadores e programas para encontrar os códigos e suas cidades rapidamente. Este é um exemplo de um caso onde a velocidade da recuperação justifica o uso de um banco de dados computadorizado mais do que um livro de consulta.

OPERAÇÕES AUTOMÁTICAS DE DADOS

Um sistema gerenciador de banco de dados faz mais do que armazenar e recuperar os dados rapidamente. Um programa apropriadamente desenhado armazena os dados de maneira conveniente ao usuário, depois opera de maneira predeterminada e relata-os de uma forma mais significativa. Isto pode ser ilustrado considerando o fluxo de dados no comércio de vendas de um único proprietário – uma loja de artigos esportivos, por exemplo. O proprietário levaria o dia todo atendendo clientes, recolhendo recibos – pagamento à vista ou a prazo – e checando itens conforme saem do estoque. Todos os recibos e as trocas do estoque teriam de ser enviadas na mesma remessa no final do dia, com as reposições do estoque remarcadas, bem como a situação do caixa.

Ao pagar seus empregados, o comerciante teria de calcular suas vendas brutas para a comissão, retenção e dedução das taxas federais e estaduais. Se possuir clientes que debitem em conta, o comerciante precisará enviar suas contas a receber, contas gerais e, por último, fazer projeções rápidas do fluxo de caixa. Em todas estas operações, o proprietário da loja seria tão mecânico e metódico quanto possível, para minimizar erros e facilitar as coisas.

Se o dono da loja colocasse seus registros dentro de um banco de dados, isto mudaria a natureza de sua contabilidade. Ele até seria responsável pela entrada de dados e a geração de relatórios, mas os exercícios das operações – somas e subtrações das quantias em cruzados (procurar na página adequada do cliente no livro razão e diminuição estoque) – poderiam ser feitos automaticamente.

AGRUPAMENTOS E VISÕES DE DADOS

Agrupar dados significa armazenar grandes partes de dados relacionados em um ou mais arquivos para posterior recuperação. Os dois princípios do agrupamento de dados são:

1) Os dados não devem ser duplicados dentro de um agrupamento.

2) Os dados que estão relacionados entre si pertencem ao mesmo agrupamento.

Para ilustrarmos o primeiro princípio, consideremos um escritório de distribuição que serve centenas de clientes. Quando as entradas de dados de clientes são numerosas, a maioria dos comerciantes designa-lhes números e centraliza a função de manter os nomes e os endereços dos clientes em um arquivo principal. A inconveniência de buscar o registro pelo nome do cliente é justificada pela incidência de enganos de transcrição (grafia) do nome. Além disso, arquivos de dados agrupados terão o nome e o endereço do registrado uma só vez. Em um banco dados bem desenhado, nenhum item dos dados seria repetido porque permitiríamos a possibilidade de um erro embaraçoso, tal como uma carta personalizada enviada para "Ellen Campbell", cujo endereço iniciasse com 'Querida Allen".

É difícil justificar o princípio de que todos os dados relacionados pertencem a apenas um banco de dados, ponto que é sempre discutido ao desenhá-lo. A criação de um programa de estoque, usado mais tarde, neste livro, como um programa de aprendizagem, é um exemplo. Quando foi desenhado, alguns dos engenheiros de montagem fizeram objeções em colocar o custo dos itens em cada registro do estoque. Eles argumentaram que um item não muda a sua função, seja qual for o seu preço e o dado extra representava algo desagradável para as pessoas que fossem tratá-lo.

Os responsáveis pelo estoque, por outro lado, argumentaram que a avaliação dos custos era muito simples, desde que ignoravam os custos das montagens de terceiros, cujo valor era somado (mão-de-obra primária) às montagens domésticas. De qualquer maneira, os custos de cada unidade durante o ano seriam colocados no sistema que foi desenhado pelos programadores e retinham uma validade aproximada para distribuir as causas dos custos excedentes e os efeitos dos pedidos rápidos.

O ponto fundamental é: a menos que os custos dos itens tenham sido desenhados dentro do sistema desde o início, nenhum tipo de pergunta sobre os custos poderia jamais ser respondida pelo banco de dados. Colocando de uma outra forma, não podemos nunca conseguir dados de fora de um banco de dados, a menos que alguém os coloque primeiro. Assim, se uma informação relaciona-se com o banco de dados e achamos que podemos fazer perguntas sobre aquela informação algum dia, ela provavelmente deverá ter um lugar previamente reservado dentro do banco de dados.

As Visões do Usuário de uma União de Dados

Devido ao fato de que dados relacionais podem ser organizados de várias maneiras, um único banco de dados pode ser visto de formas diferentes, dependendo da relação de fatos desenhados a partir dos dados disponíveis. O conjunto de fatos que um usuário determinado vê em um banco de dados e a maneira como os fatos estão relacionados constituem a "visão do usuário". Podemos ilustrar o conceito das visões do usuário voltando ao exemplo da loja de vendas em atacado de artigos esportivos, assumindo que o dono colocou seu estoque, contas e transações em um banco de dados relacional. Um programa de banco de dados na operação de vendas em atacado aparecerá como sendo vários programas diferentes, dependendo de quem está olhando

para ele. Tipicamente, o registro fundamental será a entrada da venda realizada por um empregado no momento do pedido. Para o empregado, o sistema de banco de dados será a caixa registradora, indicando o nome do cliente, a mercadoria pelo número de estoque, a data e a forma de pagamento. Como resposta, o computador indicará ao empregado o preço, calculará a taxa, afixará qualquer desconto e possivelmente verificará o arquivo de crédito do cliente para ver se possui um histórico de cheques sem fundos.

Os dados da venda possuiriam aparências diferentes quando fossem utilizados para outros tipos de funções. Para o contador, um compêndio das vendas do mês, taxas recolhidas, comissões dos empregados e algo mais, apareceria como se fizesse parte de um programa de contabilidade. Da mesma maneira, o empregado do estoque veria o banco de dados como um programa de inventário, desde que as mercadorias fossem enviadas ao estoque automaticamente, após cada compra.

De fato, uma das propriedades do software de banco de dados é a habilidade de realocar mais programas de computador especializados em um sistema generalizado que pode ser adaptado às mudanças de necessidade do usuário. Os métodos para escrever estes tipos de programas usando o dBase II serão discutidos nos Capítulos 8, 9 e 10. O ponto é que uma vez que parte dos dados tenham sido reunidos de forma recuperável, sua utilidade é limitada apenas pela natureza dos dados e a imaginação da pessoa ao fazer perguntas ao banco de dados.

O VALOR DA INFORMAÇÃO

O valor da informação aumenta conforme ela é processada. No exemplo da loja de artigos esportivos, os recibos semanais continham todas as informações para preencher os registros financeiros e de estoque da loja, mas os registros afixados eram obviamente mais valiosos apenas pelo valor da somatória de trabalho. O valor dos dados é sempre muito subjetivo e informações aparentemente desprezíveis representam uma mina de ouro. A loja de artigos esportivos poderia estar perdendo tempo se armazenasse o signo astrológico de cada cliente, ao passo que o esporte favorito seria algo muito útil para se ter em um arquivo. Para uma livraria especializada em ciências ocultas, o valor dos dados seria o inverso.

Neste contexto, a maioria dos livros sobre processamento de dados computadorizados recomendam que seus leitores façam backups de seus dados freqüentemente. O mesmo recomendamos aqui. Um disco flexível de 5 1/4 polegadas custa entre Cz$ 50,00 e Cz$ 60,00 — menos, se comprarmos em quantidade. Avalie este custo em relação ao custo estipulado de recriar seus registros após a perda acidental ou destruição de uma primeira versão de seus arquivos de dados. Então considere as incertezas envolvidas, e decida sozinho se o fato de obter cópias dos dados faz sentido. O fato de copiar, porém, significa apenas um aspecto de proteção ao valor de seus dados. Outras práticas prudentes seriam: incluir uma parte fixa na rotina do computador e os programas de banco de dados deveriam ter características de segurança construídas por dentro. Estas práticas e características incluem:

1. Reter todos os documentos-fonte de maneira organizada por um período razoável depois da entrada no computador.
2. Marcar os documentos-fonte quando são operados para prevenir entradas duplicadas.

3. Desenhar programas que gerem automaticamente relatórios periódicos imprimindo todas as mudanças processadas nos bancos de dados.
4. Investir em sistema apropriado para copiar discos rígidos.
5. Instituir procedimentos disciplinares para rotular os discos flexíveis com etiquetas que mostrem seus conteúdos atuais.

CLASSIFICAR E INDEXAR

Os dados aleatórios tornam-se mais valiosos quando ordenados. Consideremos uma empresa de encomendas por correio que possui uma lista de nomes e endereços de clientes em potencial. Enquanto a lista tivesse valor, seria mais interessante que os clientes que fizeram encomendas mais recentemente fossem separados das pessoas que apenas se interessassem em obter os catálogos. Além disto, esta lista torna-se ainda mais valiosa para a casa de encomendas se for classificada pelo código postal, já que o correio necessita empacotar as quantidades pelo CEP para calcular suas taxas.

dBase ordena os dados através de dois mecanismos: **sorts** e **indexes**. Quando falamos em classificar arquivos diariamente, geralmente queremos dizer pegar um conjunto de objetivos e reorganizá-los em uma seqüência diferente. Para fazermos uma classificação alfabética de 25 cartões de arquivo, por exemplo, espalharíamos os 25 cartões em uma mesa, separaríamos os As, depois os Bs e assim por diante, até que estivessem em nova ordem. Teríamos 25 cartões no início e continuaríamos com os mesmos 25 no final do processo.

Porém, quando um computador classifica, faz conceitualmente algo diferente. Primeiro abre o arquivo de entrada de 25 registros. Em um outro lugar do disco (ou qualquer dispositivo de armazenamento em massa) ele abre um novo arquivo para a saída classificada. Então, lê do arquivo de entrada para determinar a ordenação dos registros e escreve os 25 registros no arquivo de saída na ordem correta. O arquivo original não é alterado de forma alguma, mas é criado um arquivo classificado e completamente novo, contendo os mesmos registros em uma ordem diferente.

Isto é muito simples (existem livros inteiros sobre classificação). A distinção definitiva de uma classificação em dBase é que esta cria um novo arquivo do mesmo tamanho do original, mas com um outro nome e uma nova ordem.

Como um todo, indexar em dBase é diferente. Como as classificações, as indexações oferecem ordem aos registros em um banco de dados, mas através de um mecanismo diferente. Uma boa analogia para o entendimento da indexação pode ser vista em uma biblioteca de um colégio. As estantes de uma grande biblioteca podem conter milhares de livros, e mesmo que estes livros tenham sido classificados duas vezes (pelo assunto e pelo autor) em relação ao assunto, representaria um objetivo formidável localizar um livro determinado iniciando da primeira prateleira de cima para baixo em cada estante e assim seqüencialmente. É muito mais prático iniciar a pesquisa nos dois índices que os bibliotecários mantêm na recepção: os cartões catalogados pelo autor, título e assunto. Nestes índices, cada entrada ou cartão mantém uma chave de pesquisa para cada livro individualmente na forma de listagens por autor ou assunto, e associa aquela chave a um "indicador" ao livro (como em uma biblioteca modelo ou número

decimal de Dewey). Com os números dos livros e depois com um mapa intermediário da biblioteca que mostre onde estão várias categorias de números, podemos encontrar o livro diretamente.

A indexação em dBase funciona da mesma forma. Falando tecnicamente, o dBase é um banco de dados com um **Index Sequencial Access Method (ISAM)*** (veja a Figura 1-1), isto significa que os registros em dBase são armazenados seqüencialmente quando entram. O acesso rápido a estes registros depende de uma ou mais indexação que é construída e mantida em dia.

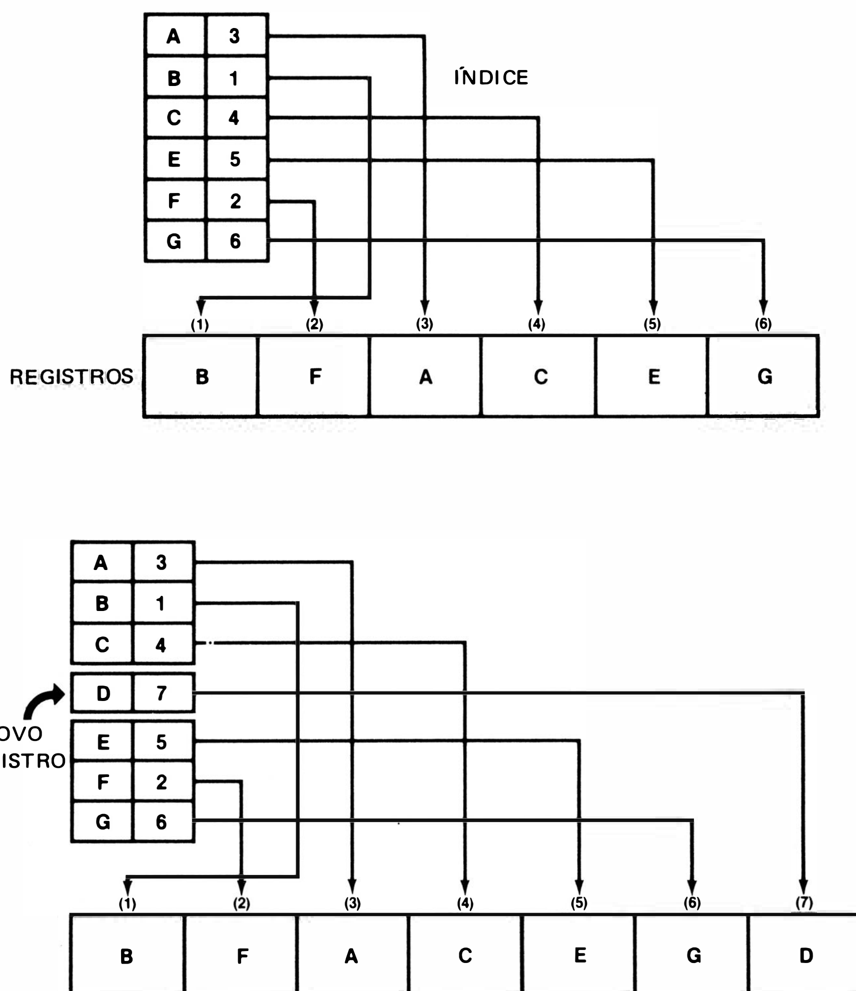


Figura 1-1 Um Esquema de ISAM (Método de Acesso Seqüencial Indexado).

* N.T. – Método de Acesso Seqüencial Indexado.

Se permitirmos algumas liberdades à analogia da biblioteca, o sistema ISAM poderá ser descrito em termos de um bibliotecário sem muita imaginação em classificar livros, mas com uma infinita paciência e velocidade em catalogá-los e recuperá-los. Quando nosso bibliotecário receber o primeiro livro da biblioteca, ele o colocará em posição 1, em suas prateleiras, sem se preocupar com o assunto. Quando entrarem mais livros, ele os colocará seqüencialmente perto, preenchendo sua biblioteca da esquerda para a direita e do início ao fim.

A única graça que poderia nos salvar desta situação caótica seria se o nosso sábio bibliotecário transcrevesse cada livro novo em quantos arquivos de cartões diferentes definíssemos para ele. Poderíamos indexar por autor, título, assunto, número da biblioteca modelo e, se quiséssemos, pelo editor. Então, necessitando de um livro, diremos simplesmente ao bibliotecário um fato relacionado a ele, tal como quem escreveu, e ele trará o livro certo como em um toque de mágica.

Em dBase é preferível indexar do que classificar, quando queremos tirar vantagem de um banco de dados. Dizer que a indexação "coloca os registros em ordem" é um fato. Mesmo que os registros tenham sido colocados em um banco de dados aleatoriamente (seqüencialmente), o dBase executa com rapidez a tarefa de indexá-los e fazer com que eles pareçam ter sido classificados.

Nosso fiel e super-rápido bibliotecário poderia demonstrar-nos este processo. Imaginemos estar diante dos diferentes catálogos, folheando os cartões um por um, enquanto o bibliotecário encontraria o livro mencionado em cada cartão em um microssegundo. Quando procuramos através dos autores (Andrade, Assunção, Aguiar), vemos uma biblioteca; procurando pelos assuntos catalogados (ácido, aerosol, álcool), vemos uma outra ...

BANCO DE DADOS RELACIONAIS *VERSUS* HIERÁRQUICOS

Obviamente os bancos de dados podem ser desenhados de inúmeras maneiras diferentes. Os dois conceitos de desenhos mais freqüentemente mencionados em livros de microcomputadores são os bancos de dados **relacionais** e os **hierárquicos**. Nos dois casos, o adjetivo usado para descrever o desenho do banco de dados refere-se à organização da inter-relação entre os arquivos de dados no banco de dados.

O Desenho Hierárquico

A característica que distingue um banco de dados hierárquico é a habilidade de que cada arquivo listado em um diretório contém um outro diretório para si próprio. Uma analogia com arquivos de papéis fará com que fique mais claro. Vamos iniciar imaginando um grande arquivo de pastas com o rótulo de "Nações Ocidentais". Quando fôssemos abri-lo encontraríamos uma coleção de pastas de arquivos menores com os rótulos tais como "Inglaterra", "Mônaco", "Canadá" e "Estados Unidos". Se abríssemos um destes arquivos de pastas pequenas como Estados Unidos, poderíamos descobrir que este possui 50 subpastas, rotuladas com "Alaska", "Ohio", "Rhode Island" e assim por diante.

Os bancos de dados hierárquicos são organizados em linhas parecidas, com cada item em um arquivo de dados particular possuindo ao menos a capacidade de representar um arquivo de dados próprio. (Veja a Figura 1-2.) Tais sistemas podem ser extremamente poderosos e flexíveis, já que os usuários podem concentrar suas atenções em qualquer tarefa, sem que antes tenham de considerar em que nível as informações estão armazenadas no banco de dados.

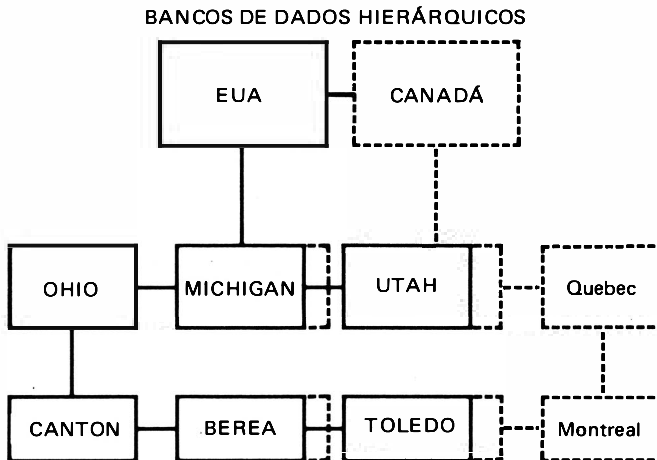


Figura 1-2 Nos bancos de dados hierárquicos, cada subdivisão de um registro pode também possuir subdivisões.

Se estivéssemos fazendo perguntas ao nosso banco de dados imaginário, por exemplo, usaríamos o mesmo procedimento seja em qualquer um dos arquivos, "Nações Ocidentais" ou "Ohio", onde poderíamos encontrar listagens para todos os municípios – "Lorain", "Sandusky" e assim por diante.

Por um lado negativo, os bancos de dados hierárquicos podem ser muito difíceis, até para os usuários experientes compreenderem seu funcionamento. Se estivéssemos no diretório de Nações Ocidentais, por exemplo, poderíamos ter uma boa idéia do que encontraríamos nos arquivos Estados Unidos e Canadá, mas o que encontraríamos em Mônaco? Como descobriríamos qual arquivo de nação abrir para obtermos informações sobre "Ticino" se não soubéssemos que é um distrito da Suíça? As primeiras experiências com os bancos de dados hierárquicos é algo como se estivéssemos perdidos em uma floresta à noite.

Desenho Relacional

O princípio de organização que envolve o desenho relacional em bancos de dados é que os arquivos de dados que possuem qualquer relação entre si geralmente devem suas associações a um fator primário, chamado "ligação".

Consideremos um arquivo de uma escola primária. Em algum lugar haverá um arquivo contendo uma lista diferente de cada classe organizada pelo professor e sua classe. Em outro, cada registro do aluno com seu endereço e o nome dos pais. Se por alguma razão a administração solicitar uma lista dos nomes e endereços de todos os pais dos alunos do quinto ano, alguém, geralmente uma secretária com trabalho excessivo, terá de iniciar com as listas do quinto ano e pesquisar o arquivo dos alunos um por um para fazer a lista. Neste caso, os bancos de dados CLASSE:LISTAS E ALUNOS:REGISTROS foram unidos através do campo ALUNO:NOME, presente em ambos.

O desenho dos bancos de dados relacionais aproximam-se da maneira em que a maioria das pessoas pensam de seus dados e é geralmente mais fácil de entender. Estamos acostumados a procurar pedidos pelo número do cliente ou correspondência arquivada pelo nome do remetente.

Um desenho relacional é sempre mais flexível quando um novo arquivo de dados é necessário, ele pode ser criado "do lado de fora" sem afetar os dados ou as relações entre os dados já estabelecidos no banco de dados. No exemplo da escola primária, se a lista dos pais e endereços por classe era importante o suficiente para estabelecer a criação de um novo arquivo, então ele poderia ser construído sem a necessidade de redesenhar nenhum dos arquivos já estabelecidos.

CAPÍTULO

2

Conceitos de Banco de Dados

Neste capítulo trataremos dos conceitos do gerenciamento de bancos de dados computadorizados, com maiores detalhes e, novamente, enfatizando como o dBase II funciona em um banco de dados de um microcomputador. Usaremos também alguns exemplos para ajudar a entender o trabalho interno do sistema que será importante quando formos escrever programas.

A TERMINOLOGIA DOS BANCOS DE DADOS

Existem três termos no gerenciamento de bancos de dados que ocasionalmente confundem-se, mas que possuem significados específicos e precisos: campo, registro e arquivo. Já que as definições formais podem gerar confusões, definiremos estes termos considerando um banco de dados contido em um pacote de cartões de arquivo de 3 por 5 polegadas, como ilustrado na Figura 2-1.

Se mantivermos este banco de dados propositadamente simples, teremos os cartões de arquivo sem nada a não ser os espaços em branco para preenchermos o nome e cidade. Se fôssemos preencher estes cartões com os nomes e cidades de nossos familiares, pessoas do clube, time de futebol ou outro grupo qualquer, teríamos um pequeno banco de dados.

Arquivos

Em nosso exemplo, o termo **arquivo** é usado para expressar a totalidade de um grupo de cartões. Assim, se tivéssemos listado o time de futebol e as pessoas do clube, teríamos dois arquivos.

Podemos notar que os arquivos podem ser grandes ou pequenos – um ou até nenhum cartão – e que eles podem estar subdivididos em arquivos menores mesmo que possuam uma linha lógica ou arbitrária. O time de futebol poderia ser dividido entre aqueles que nasceram na capital e no resto do estado e as pessoas do clube, entre os que praticam esportes internos e aqueles que apenas participam socialmente.

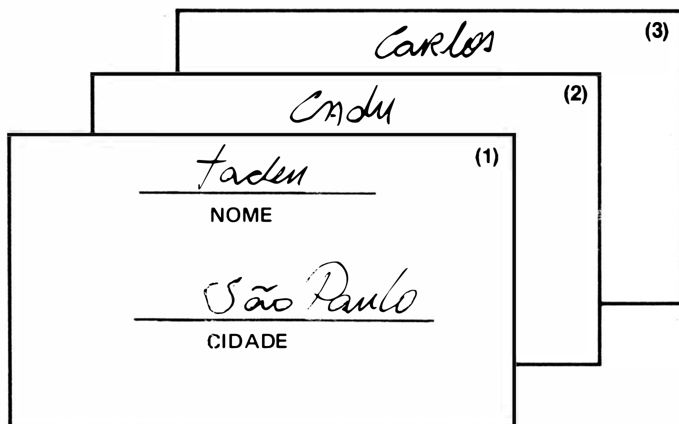


Figura 2-1 A terminologia do banco de dados ilustrada em cartões de arquivo de 3 por 5 polegadas.

Registros

Em nosso exemplo, cada cartão de arquivo individual é um **registro**. O registro #1, o primeiro, refere-se a Tadeu de São Paulo e o segundo, atrás deste, (#2) mostra o nome Cadu. Não podemos vê-lo, mas ele é de Campinas.

Existem alguns pontos importantes sobre os registros. O primeiro é que todos os dados em um registro fazem parte dele e a associação entre eles é completa. Como na ilustração, na Figura 2-1, Tadeu é de São Paulo e Cadu é de Campinas e em nenhuma manipulação em nosso banco de dados nunca ocorrerá a troca de cidades. No caso disto acontecer, teremos um problema: o nosso sistema gerenciador de banco de dados estaria inutilizado.

O segundo ponto é que cada registro em um banco de dados possui a mesma habilidade para guardar um dado – nem mais nem menos. Considerando o registro #3 na Figura 2-1 para Carlos, imaginemos que não foi escrito nenhuma cidade em seu cartão. Provavelmente ela deve ter mudado ou quem preencheu seu cartão não possuia a informação. Isto acontece.

Mas mesmo assim, o cartão do Carlos possui espaço para preenchermos a cidade. Não economizamos nenhum espaço no arquivo deixando aquela parte do cartão em branco – na verdade, deixamos um buraco no nosso banco de dados, e até que não coloquemos os registros em ordem alfabética por cidade, devemos decidir sobre a falta da cidade do Carlos.

Campos

O termo **campo** é reservado para indicar um pedaço específico de dado dentro de um registro. Em nosso exemplo, cada registro possui dois campos, um nome e uma cidade. Os campos possuem a mesma classificação de identificação distinta dentro dos registros que estes possuem dentro dos arquivos. Não haverá nenhuma troca de cidades com nomes. Se um campo foi

definido em um registro, então ele deve ser considerado como uma extensão do banco de dados, aparecendo em cada registro apenas uma vez. Tadeu possui apenas uma cidade, o mesmo acontece com Cadu e Carlos (já que uma cidade não preenchida é uma entrada válida).

TIPOS DE DADOS E A DIGITAÇÃO

Para registrar, manipular e recuperar dados, um computador necessita que convertamos este processo em uma representação interna e eletrônica. Na maioria das vezes, o usuário não necessita preocupar-se com isto. Digitamos coisas na máquina e pressionamos a tecla "enter" ou "return". A máquina cuida de toda a operação eletrônica.

Porém, os programadores devem entender da digitação de dados e como escrever programas, estarem alertos de que tipo de dado a máquina estará armazenando. Algumas linguagens de programação possuem mais de seis tipos diferentes de dados e outras como Pascal permitem ao programador definir novos tipos de dados para objetivos especiais.

Felizmente, o dBase II é simples e possui somente três tipos de dados: caractere, numérico, e lógico. Estes são os únicos tipos de dados que podem ser armazenados em um arquivo de dados dBase. Pelo fato de ser essencial um bom entendimento destes três tipos para a programação, cada um será discutido de forma extensa.

Dados Tipo Caractere

Em termos técnicos, o dado de caractere em dBase consiste em todos os caracteres ASCII excluindo os caracteres de controle, e é armazenado em série. Estas séries são classificadas, indexadas e até comparadas na seqüência de intercalação do ASCII standard.

O dado de caractere geralmente necessita ser marcado por delimitadores, ou " ", ' ' ou [ ]. Vamos reformular isto de maneira não técnica. Quando dizemos que os dados do dBase serão do tipo de caractere, ele aceitará letras maiúsculas e minúsculas, os dez dígitos, todos os sinais de pontuação e o espaço. Em um total de 95 caracteres diferentes e cada um é único, não possuindo significados intrínsecos. Lembre-se de que estamos falando sobre dados armazenados como caracteres. Em um *programa* dBase, a seqüência de caracteres:

```
? (2 * 6)/4
```

teria um significado muito específico e faria com que o dBase imprimisse o numeral 3. Neste caso, o ponto de interrogação, os parênteses, o asterisco, a barra e os dígitos, todos possuem significados específicos de valor, operação e agrupamento. Porém se estes mesmos caracteres estivessem *armazenados em um banco de dados*, eles perderiam o significado matemático e seriam simplesmente dados do tipo caractere.

Além disto, como todos os programas de computação, o dBase não possui nenhuma maneira de fazer julgamentos qualitativos sobre as relações entre caracteres diferentes. Onde quer que vejamos "A" e "a" como sendo a mesma letra, para o dBase, elas são caracteres únicos

com a mesma semelhança entre "W" e "#". Uma das experiências mais frustrantes que o usuário de bancos de dados pode ter é passar horas pesquisando por um cliente com o nome de "Silva", quando o banco de dados possui o sobrenome armazenado como "SILVA". As séries de caracteres *não* são as mesmas para o dBase, e é da responsabilidade do programador lembrar deste fato, bem como antecipar e prevenir problemas que este pode causar.

Embora o dBase não possa fazer distinções de julgamentos entre os caracteres, ele possui uma ordem quantitativa para eles, do menor para o maior, chamada "seqüência de intercalação" (veja a Tabela 2-1). Como os computadores, o dBase usa o código ASCII (American National Standard Code for Information Interchange)* para armazenar dados tipo caractere (veja o Apêndice E). O valor dos caracteres em ASCII de 32 (espaço) até 126 (til ou "~") fornece a seqüência que o dBase usará para classificar aqueles caracteres, em uma classificação em ordem alfabética extensiva. A habilidade do dBase em classificar os dados tipo caractere usando esta seqüência de intercalação e de comparar duas séries de caracteres com outras para determinar qual é "maior", é uma das características mais poderosas do programa. Os dados podem ser definidos como sendo tipo caractere quando é criado um arquivo de banco de dados (Capítulo 4), ou dentro de um programa. Quando uma variável aparece em um programa pela primeira vez, o dBase não a considera como uma série de caracteres, a menos que apareça com aspas (duplas ou simples) ou colchetes. Estes delimitadores devem ser iguais no início e final da série, como em:

```
'ESTOU COM SEDE'
"VOU TOMAR UM COPO D'ÁGUA"                (*Apóstrofe incluso*)
[UM COPO D'ÁGUA MARCA "DOYALIN"]          (*Apóstrofe e aspas inclusas*)
```

No segundo exemplo, as aspas simples são inapropriadas como delimitadores por causa do apóstrofe dentro da série. As aspas duplas, como mostramos, ou os colchetes, são necessários para delimitar a série. Da mesma forma, o terceiro exemplo contém aspas simples e duplas, então necessitamos dos colchetes como delimitadores.

Quando o dado do tipo caractere é armazenado em um arquivo de banco de dados dBase em disco, ele é registrado como caractere ASCII, ajustando-se à esquerda no campo de dados.

Dados Numéricos

O único tipo de dado numérico do dBase é o "número decimal com sinal, de precisão simples e ponto flutuante". As operações aritméticas limitam-se em adição, subtração, multiplicação e divisão, usando os símbolos das operações +, –, * e /. Os campos de dados definidos como numéricos podem aceitar somente dígitos, um comando de mais ou menos e um ponto como um decimal. O dado numérico é armazenado em arquivos de bancos de dados em disco como dígitos de ASCII. Exemplificando em termos não técnicos, o dBase, como todos os programas de computação, deve ser avisado quando um usuário pretende considerar um dado específico

* N.T. – Código-Padrão Americano para Intercâmbio de Informações.

Tabela 2-1. Seqüência de Intercalação do dBase II

Caractere	Valor Ordinal	Caractere	Valor Ordinal	Caractere	Valor Ordinal
Space	32	@	64	`	96
!	33	A	65	a	97
"	34	B	66	b	98
#	35	C	67	c	99
$	36	D	68	d	100
%	37	E	69	e	101
&	38	F	70	f	102
'	39	G	71	g	103
(	40	H	72	h	104
)	41	I	73	i	105
*	42	J	74	j	106
+	43	K	75	k	107
,	44	L	76	l	108
–	45	M	77	m	109
.	46	N	78	n	110
/	47	O	79	o	111
0	48	P	80	p	112
1	49	Q	81	q	113
2	50	R	82	r	114
3	51	S	83	s	115
4	52	T	84	t	116
5	53	U	85	u	117
6	54	V	86	v	118
7	55	W	87	w	119
8	56	X	88	x	120
9	57	Y	89	y	121
:	58	Z	90	z	122
;	59	[	91	{	123
<	60	\	92	|	124
=	61	]	93	}	125
>	62	^	94	~	126
?	63	_	95		

como um número. O termo "considerar como um número" possui um significado crítico aqui. Para nós, um número de telefone tal como 2255959 é muito mais significativo do que $29.95 ou 55 km.

Esta distinção entre números como: datas, números de telefone, endereços, códigos postais e licenças de automóvel, que são na verdade dados de caractere e números representativos de medida ou quantidades, devem ser colocados claramente para o dBase. Se efetuarmos qualquer tipo de manipulação matemática no número, o computador tem de traduzir o número em uma representação interna apropriada para operar aritmeticamente. Diferente de outras linguagens de programação, o dBase pode aceitar e processar apenas números decimais. Quando os programadores ou os usuários de programas colocam um número em um arquivo de banco de dados ou programa e quando o dBase responde com um número, a única forma de aceitá-lo é

a antiga: contar nos dedos com base dez. Os programadores mais avançados podem sentir uma tristeza passageira ao verificar que os menores alcances ao octal, hexadecimal e outros "als" são negados a eles, mas para nós isto é um alívio.

Uma vez que um número entrou no computador, o dBase coloca-os em uma forma codificada chamada **ponto flutuante**, que é apropriada para uma aritmética fácil. O sistema de codificação de ponto flutuante é muito parecido com a notação científica que a maioria de nós já aprendeu na escola, em que um número é representado por uma quantidade decimal (chamada "mantissa") multiplicada pela potência de dez. Isto traz a vantagem de permitir expressar os números grandes e pequenos facilmente e somá-los ou multiplicá-los sem problemas. Por exemplo:

Número grande: 6.023×10^{23}
Número pequeno: 6.624×10^{-27}

Esta escolha de representação numérica é completamente flexível e útil. Permite números nos arquivos dBase e programas situarem-se entre mais ou menos 1.8×10^{63}, para números grandes e mais ou menos 1×10^{63} para números pequenos com precisão de dez dígitos.

Porém, os números de ponto flutuante possuem uma desvantagem, que os usuários devem estar cientes – às vezes, o ponto flutuante aritmético não fornece a resposta "correta". Consideremos o seguinte comando dBase:

```
? (7/9)*9
```

Aqui pedimos ao dBase para imprimir o resultado de um cálculo "sete dividido por nove, e então multiplicar por nove". Todos sabemos que as operações são comunicáveis (isto é, a ordem em que são executadas não podem afetar o resultado) e quando dividirmos por nove e então multiplicarmos por nove, de fato não teríamos feito nada. Por que então, quando pressionamos a tecla return o dBase diz que a resposta é seis (o que ele faz?). A razão concentra-se na representação interna do ponto flutuante. O dBase separa cada número da expressão, e pára a codificação de ponto flutuante com um valor aproximado.

Esta aproximação é bem razoável, com precisão de dez dígitos, mas já existe uma mudança da base decimal para a binária (os computadores não nasceram com dez dedos) e desde que o computador possui uma área limitada na qual grava números com precisão ilimitada, a exatidão é perdida.

Para completar, a maneira com que colocamos a expressão, diz implicitamente ao dBase que não estávamos interessados em nenhum dígito que poderia aparecer depois do ponto decimal.

Como uma segunda experiência, se digitarmos:

```
? (7.000000/9) *9
```

O dBase voltaria com 6.999999, que é mais perto do que temos em mente.

Enquanto isto é aceitável em termos científicos, os comerciantes esperam que os cruzados e os centavos se apresentem corretamente. Mas como os programadores veteranos em BASIC nos dirão, o ponto flutuante fornecerá a resposta correta se cuidarmos dele com carinho. Falando mais claro, neste tipo de matemática a resposta seria arredonde antes do truncamento. Por exemplo, uma expressão que forneça o preço final como custo mais taxas de venda pode ser:

```
(preço final) = ((custo) * (1 + taxas de venda)) + 0.009
```

Neste caso, somando 0.009 ao resultado antes de truncar a resposta para as duas casas decimais assegura que o preço inclui o centavo final.

Quando o dBase grava dados numéricos em um arquivo de banco de dados em disco, ele traduz os números para a representação ASCII de seu decimal equivalente. Em outras palavras, os números são armazenados em disco como caracteres, os quais são escritos e lidos do disco através da mesma tradução usada para a entrada e saída do teclado ou tela.

Dados Lógicos Booleanos

Valores lógicos ou **booleanos**, nomeados por George Boole, matemático inglês dos anos de 1800, são aqueles dados que representam uma ou duas condições exclusivas. Um exemplo pode ser de uma conta em um banco de dados de contabilidade, que possui um lugar para marcar "PAGO". Na verdade, as contas são pagas ou não. Não importa quantas vezes dizemos aos credores que a conta foi paga pela metade, eles (e seus computadores) continuam insistindo que uma conta paga pela metade não está realmente PAGA. Da mesma forma, um escritório de ex-estudantes de uma universidade poderia colocar um sinal em seus registros em um lugar chamado "FORMADO". Não importa qual o "status" de qualquer estudante, se subiu na vida ou não, se está vivo ou morto, cada aluno possui a classificação de formado ou não formado.

Os dados lógicos são de várias maneiras os mais fáceis de trabalhar. Por si só, como um tipo de dado, o dado de valor lógico seria muito limitado. Aplicações reais sempre pedem para encontrar aquelas contas que foram pagas pela metade ou os alunos que quase se formaram. Entretanto a habilidade em armazenar e testar valores lógicos oferece ao programador em dBase um instrumento poderoso para a classificação rápida dos registros em dois grupos. Mais tarde veremos os valores lógicos sozinhos como dispositivo de controle interno dentro dos programas dBase.

Os valores lógicos ou booleanos, dentro dos arquivos de bancos de dados dBase representam um único caractere com valor ASCII para "T", "t", "Y" e "y" para "verdadeiro" (ou "sim") e "F", "f", "N" e "n" para "falso" (ou "não"). Quando os valores lógicos são trazidos para a tela, o dBase apresenta "T" ou "f" com pontos como .T. ou .F., para distinguir o verdadeiro e falso dos caracteres "T" e "F". Os programadores não devem usar estes pontos em programas. Nos arquivos de banco de dados, o dBase registra dados lógicos armazenando o caractere ASCII digitado – T, t, Y, y, F, f, N, ou n. A sintaxe do dBase para comandos lógicos será discutida extensamente no Capítulo 3.

Intercâmbio de Tipos de Dados

Enquanto o objetivo dos programadores é selecionar tipos de dados certos para cada item de dado quando um programa é desenhado, às vezes é necessário converter um tipo de dado para outro dentro de um programa.

Suponhamos que um dentista queira, por exemplo, colocar seus clientes em ciclos diferentes para seus *check-ups*, alguns de quatro em quatro meses, alguns de seis em seis e outros anualmente. Existiriam várias maneiras para fazermos isto em um programa dBase, mas um bom sistema poderia ser armazenar o intervalo das rotações como um número (4, 6 ou 0 para anualmente), e armazenar a data como uma série no formato "mm/dd/aa".

Quando a visita de um novo paciente entrasse no computador, o programa dBase poderia converter os dois primeiros caracteres da data em um número, somar o fator de rotação, subtrair 12 se a resposta fosse maior que 12, e então converter aquele número em uma série de caractere para o mês da próxima visita. Por exemplo:

VISITAS: DATA	"09/15/85"	(Série de caractere)
"09" →	9	(Conversão numérica)
9 + 6 →	15	(Somar seis meses no fator rotação)
15 – 12 →	3	(Número do mês da próxima visita)
3 →	"03"	(Converter para série de caractere)
NOVA VISITA	"03/15/86"	(Data da nova visita calculada)

Enquanto estas conversões e outras funções relatadas são discutidas em detalhes no Capítulo 5, mencionamos aqui que o dBase permite conversões de dados para todos os seus tipos de dados. Portanto, já que o dado não é fixado em nenhum tipo específico para sempre, o programador é livre para designar o dado para o tipo mais conveniente à sua aplicação. Existem três regras que fazem com que a seleção do tipo de dado fique mais fácil:

1. Use o tipo caractere como default para os programas dBase. O dado tipo caractere é o mais fácil para pesquisar e manipular.
2. Selecione o tipo numérico para o dado se (e somente se) for fazer cálculos aritméticos com ele.
3. Apesar de campos lógicos serem muito úteis para controles internos, uma vez assim definidos eles não servirão para mais nada. Antes de selecionar um campo lógico, considere-o como um campo de um caractere. E assim terá:

```
CRÉDITO (S/N) : :
```

considerar:

```
CRÉDITO (0-5) : :
```

O primeiro exemplo permitiria ao usuário colocar se o cliente possui crédito ou não. O esquema de baixo permite ao usuário entrar com o crédito avaliado como um número de 0 ("de forma alguma") a 5 ("cliente especial").

Porém, se um campo lógico tem sentido – o conteúdo representa um dos dois casos mutuamente exclusivos com uma verdade para cada registro – então utilize-o sem susto.

DESENHO DO REGISTRO

Uma pequena olhada no modelo do cartão de arquivo poderia deixar claro como o desenho do registro é o elemento mais importante sob o qual o banco de dados funcionará. Os registros podem sempre ser adicionados ou eliminados do arquivo em uso, com o número de registros variando de zero ao limite máximo do dBase. Além disso, quando um banco de dados é criado, muitos campos podem ser nomeados arbitrariamente e definimos dentro dos registros como o programa permite. O dBase II permite 65.535 registros por arquivo e 32 campos por registro (veja o Apêndice A), o que é bastante para a maioria das aplicações.

Um bom desenho de registro evitará possíveis erros, mas um registro pobremente definido será uma fonte de constantes aborrecimentos até que o programador finalmente desista com ódio e modifique a estrutura do registro. Um desenho ideal de registro será discutido de maneira mais extensa no Capítulo 9; porém, é instrutivo examinar um exemplo do tipo de confusão que um registro mal desenhado pode causar a um programador.

Vamos considerar hipoteticamente o escritório de ex-estudantes de uma universidade, que transferiu seus velhos registros para um arquivo dBase (Figura 2-2). O programador, acreditando que o programa poderia aproximar-se do antigo sistema o mais perto possível (geralmente uma boa prática de programação), define um campo no registro chamado "NOME" com 30 caracteres de extensão. O resto do registro é definido com campos para endereços, o nome do cônjuge, contribuições anuais (se fizeram ou não e quanto) e todos os outros detalhes que a associação usa para preencher suas intermináveis correspondências.

Depois dos funcionários do escritório entrarem com os registros de milhares ex-estudantes em seu banco de dados, os problemas iniciaram. Em primeiro lugar, eles entraram com todos os sobrenomes primeiro, porque sempre trabalharam desta forma e também porque tipicamente esta é a maneira mais comum de classificarmos nomes – alfabeticamente pelo sobrenome*. Este processo seria ótimo, entretanto, envolveria um tempo adicional muito grande com a exaustiva tarefa de redigitar tudo com o nome na frente, neste caso o preenchimento automático do computador seria um registro com o G de "Geraldo C. Simões Jr".

Qualquer um pode cometer erros ao entrar com um registro, embora seja prudente desenhar rotinas de consistência no programa, conduzindo o usuário para o dado correto. Porém, uma falha mais séria no desenho emergirá quando usarem a correspondência para solicitar dinheiro

* N.T. – Sobrenomes antes dos nomes são comuns nos EUA.

à sociedade. Dinheiro é um assunto sensível e quando Geraldo Simões sentir-se constrangido ao receber uma carta da associação endereçada para "Simões, Geraldo C. Jr.", e se a carta começar com:

Caro Simões, Geraldo C. Jr.,

---

a resposta do Geraldo poderia ser bem mal-criada. Por favor, não pensem que isto é fantasioso. Já vimos grandes segmentos de programas dBase desenhados com a finalidade de encontrar o segundo nome de um campo nome. Apesar de rotinas como estas poderem ser escritas, ela diminui a velocidade de um programa terrivelmente. Uma vez, perguntamos a um programador com um problema semelhante, porque ele não cortava o campo nome em dois ou três campos.

– Ah! ele respondeu – às vezes é difícil adivinhar o que vem pela frente quando desenhamos nossos registros.

Sem comentários ...

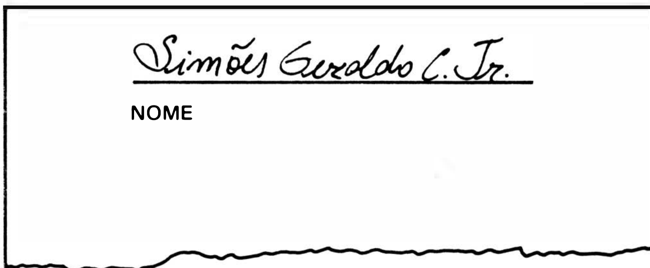


Figura 2-2 Um registro da associação dos ex-estudantes.

O MODELO DE GRADE

Apesar do modelo de cartão de arquivo de bancos de dados ser um bom exemplo de visualização de dados, algumas pessoas preferem uma outra alternativa para demonstrar seus campos, registros e arquivos. Esta alternativa é a grade ou tabela de colunas que é fornecida pela documentação do dBase. O modelo de grade possui algumas vantagens conceituais já que muitos dos comandos do dBase (notadamente LIST e REPORT FORM) mostram os dados dos arquivos em um formato de grades. Como ilustração, vamos considerar a lista de uma lavanderia mostrada na Tabela 2-2.

Tabela 2-2. A lista de uma lavanderia como a grade de dados.

Nº	Descrição	$ Custo	Goma (S/N)
3	Camisas	20,00	S
1	Calças	5,20	N
8	Gravatas	15,00	N

Em uma grade, os campos são as colunas verticais e os registros são as linhas horizontais. Os nomes dos campos estão no topo das colunas e todas as linhas juntas compreendem o arquivo. Perceba que o modelo de grade não é nada diferente do de cartão de arquivo. Os registros ainda são unidades indivisíveis – não queremos um "sim" para a goma associado com as gravatas ou o preço para passar as calças a ferro colocado na linha da camisa.

A imagem da grade de dados será valiosa mais tarde neste capítulo, mas se isto parecer confuso sobre a relação entre campos, registros e arquivos, esqueça por enquanto.

REGISTROS DE EXTENSÃO FIXA E VARIÁVEL

Obviamente, quando um programa armazena registros em um disco (ou lê registros de um disco), algumas convenções devem ser seguidas para que o programa possa comunicar-se com um campo e outro e um registro e outro. Enquanto uma grande variedade de convenções foi desenvolvida, muitas das quais são propriedades comerciais secretas usadas por programas específicos, dois métodos diferentes de marcar campos e registros, os dois simples e racionais, tornaram-se o fator padrão em microcomputação. Estes são os registros de **extensão fixa** e **delimitada** (ou **extensão variável**).

Registros Delimitados

Os registros delimitados são provavelmente mais fáceis de conceituar. Eles possuem este nome porque um delimitador, geralmente uma vírgula, é usada para marcar os limites entre os campos. O limite entre os registros também pode ser marcado por um delimitador, geralmente a tecla return ou o programa pode simplesmente manter a contagem dos campos em cada registro quando o número de campos que constitui um registro tenha sido alcançado.

Vamos observar um programa que usou registros delimitados para escrever o seguinte arquivo de nosso banco de dados de arquivo de cartões da Figura 2-1:

```
TADEU,SÃOPAULO<
Cadu,Campinas<
Carlos,<                (*Perceba o campo cidade vazio.*)
PRÓXIMONOME,PRÓXIMACIDADE<
```

Os arquivos delimitados possuem algumas vantagens. Sua melhor característica é que seus campos podem variar em extensão, assim os espaços desperdiçados não precisam ser armazenados no arquivo.

Por exemplo, se o registro de uma cidade fosse ATIBAIA, seria uma pena ter de aumentar o campo com espaços para a mesma extensão como em SÃO BERNARDO. De fato, se um campo específico em um registro está vazio, o programa pode apenas adicionar um delimitador extra para indicar seu lugar, como nestes fragmentos de três registros consecutivos contendo o endereço, o número do apartamento e a cidade:

. . . , 1234 R. Elma, apto 203, Marília, . . .
. . . , 200 Av. Cinco,,Guarujá, . . . (*Note as ",,"*)
. . . , 4321 Pr. Olga,apto 123,Florianópolis, . . .

A maioria dos programas que lêem ou escrevem registros em arquivos delimitados descarta qualquer carregamento ou espaço de pista dentro dos campos, que ajuda a preencher o dado.

Existem algumas desvantagens em registros delimitados. Assim devemos tomar cuidado ao incluir o delimitador dentro de um campo, para não confundi-lo com separador de campos. Se desejamos ter vírgulas dentro dos campos delimitados em um registro, ou temos de mudar o delimitador para uma outra marca como um sinal de número (#) ou uma barra invertida (\), o que transforma nossos arquivos em não padrões e pode atrasar o tempo real de processamento; ou deveremos encontrar alguma maneira para dizer ao programa que algumas vírgulas são para delimitar campos e outras são vírgulas mesmo.

A maneira de indicar que uma vírgula ou espaço deve ser considerado como um caractere é colocando aspas em volta dos conteúdos dos campos, o que diz ao programa para não tocar as vírgulas e espaços que podem estar dentro das aspas. Elas podem ser simples ou duplas, desde que se use o mesmo tipo para abrir e fechá-las.

Um exemplo de campo delimitado com aspas seria:

Roberto Abel, "R. Cury 675, Caixa 8692",Mauá, . . .

Sob as regras de registros delimitados, a Rua Cury e o número da caixa postal constituem um campo naquele registro e as aspas não aparecerão quando as etiquetas de endereços forem impressas.

Os *software* que usam registros delimitados são o BASIC, DataStar e a opção MailMerge do WordStar. O dBase pode ler e escrever arquivos delimitados, porém, ele não os utiliza.

Registros de Extensão Fixa

Os registros de extensão fixas são desenhados de forma que cada campo em um registro possui uma extensão fixa atribuída a ele, e o computador acessa os dados de cada campo, simplesmente contando qual a distância dos dados a partir do registro. Visualmente é a mesma figura como no modelo de grade na Tabela 2-2.

Já que não temos delimitadores, os registros de extensão fixas podem armazenar qualquer caractere ou dado numérico sem se preocupar com as regras especiais sobre quais caracteres são dados e quais são marcas de armazenamento. A única regra observada na maioria dos sistemas de extensão fixa é que o dado de caractere — letras, pontuação e dígitos — são ajustados à esquerda dentro dos campos, e os números são ajustados à direita.

Assim os campos:

```
| MILA     | 123 R ELMA   |     12.3 |
| JOÃO     | RUA PALMAS   |    – 223 |
```

onde as linhas verticais (|) representam as bordas dos campos, mostra dois campos de caracteres, para o nome e para o endereço e um campo numérico com um número arbitrário nele armazenado como dígitos, um ponto decimal e um sinal de mais ou menos.

As desvantagens dos campos de extensão fixas são mais evidentes do que as vantagens. Eles têm de ser definidos com cuidado em relação ao tamanho, já que um campo que possui poucos caracteres a mais do que ele necessita, pode desperdiçar espaço significativo quando multiplicado por milhares de registros. Por outro lado, se uma definição de campo é muito pequena, irá truncar entradas maiores:

```
| JOÃO     | ATIBAIA      |
| MILA     | SÃO BERNAR   |          (*São Bernardo, com certeza*)
```

Os registros de extensão fixa possuem uma vantagem a mais, porém, que faz com que eles sejam as estruturas de dados preferidas para muitos programadores – rotinas que pesquisam ou acessam aleatoriamente os registros de extensão fixa que tendem a funcionar com mais rapidez do que aqueles que têm a mesma função em arquivos de registro de extensão variável. Quando o computador quer o registro número 40, é muito mais fácil e rápido para ele chegar ao destino [40* (extensão do registro)] do que se tivesse de contar as vírgulas.

Os instrumentos de programação que usam os registros de extensão fixa incluem o dBase II, Pascal, FORTRAN e COBOL.

ÍNDICES BINÁRIOS E ÁRVORE-B

No Capítulo 1, falamos sobre a determinação de ordem em um banco de dados pela indexação, que com indicadores no arquivo de índice indica ao sistema onde o registro físico atual pode ser encontrado em um disco. Daquela vez, era conveniente manter a discussão e os diagramas de maneira abstrata, com setas genéricas para os indicadores e assim por diante. Aquela representação gráfica não está longe da real. Os índices físicos consistem em arquivos de disco, separados do arquivo de dados, que contém pequenos registros de índices associados a um número de registro no arquivo de dados.

Nesta seção falaremos em mais detalhes sobre a estrutura dos índices, para fins informativos – satisfazendo a curiosidade. Ela pode ser pulada no caso de não haver interesse. Lendo esta seção não será possível programar rotinas de índices (a não ser para programadores em linguagem assembly). O dBase II possui a capacidade de fazer indexações excelentes, mesmo que o operador não entenda o processo.

Um Índice Linear Simples

Na forma mais simples de um índice, o dado chave é tirado de um campo apropriado do arquivo de dados (acompanhando a anotação da posição do registro no arquivo), classifica-o e então grava-o em arquivo de índices em ordem classificada como mostramos na Tabela 2-3.

Tabela 2-3. Um índice linear simples.

Arquivo de Índices		Arquivo de Dados	
Alberto	3	1	Marcos
Clara	2	2	Clara
Marcos	1	3	Alberto
Norma	5	4	Zilda
Zilda	4	5	Norma

Um **índice linear** como este pode ser pesquisado rapidamente usando uma pesquisa "binária". O procedimento de uma pesquisa binária subdivide a lista em duas metades e vê se o item da pesquisa está no topo ou na base. O mesmo é feito com a metade da lista que o item está, e este processo continua até que o item da pesquisa seja encontrado ou seja estabelecido que o item não está no índice.

Com um índice linear, porém, um novo índice tem de ser montado todas as vezes que um novo registro é acrescentado a um arquivo o que é desagradável. Os índices lineares são a primeira escolha dos alunos que iniciam os cursos de programação, mas esta escolha não permanece constante depois.

Índice de Árvore Binária

As **árvores binárias** são estruturas de dados de grande potência para aplicações maiores. Em uma indexação construída em um modelo de árvore binária, cada entrada possui um registro do campo chave, a localização do registro e dois indicadores, um deles indicando o próximo item do índice, e outro indicando o último. Um item no índice, chamado **raiz**, é o indicador de entrada para cada pesquisa.

As árvores binárias ganharam este nome por causa do desenho da sua estrutura como mostramos na Figura 2-3. A pesquisa de uma árvore binária é bem parecida com a pesquisa binária de uma lista, exceto que as decisões são ou para a direita ou esquerda, em vez de qual metade da lista deve ser descartada. Porém, as árvores binárias possuem uma grande vantagem, quando vamos ver novas entradas. Vamos pegar um exemplo de indexar um "J" na Figura 2-3. Como com todas as indexações em computador, necessitamos de uma maneira para fazermos decisões comparativas com os valores relativos dos itens de dados, e como sempre, usaremos a seqüência de intercalação ASCII, na qual "A" é maior que "B", "B" é maior que "C" etc.

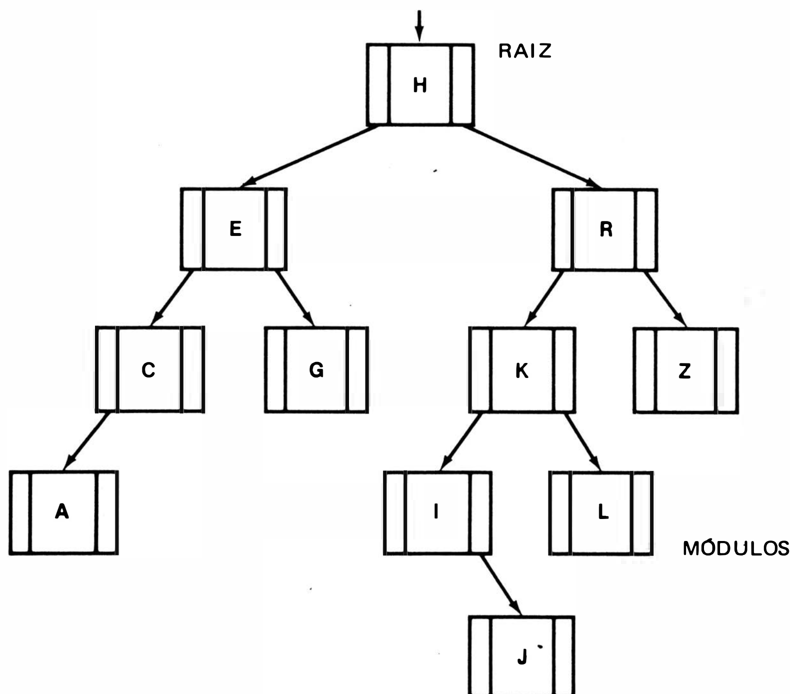


Figura 2-3 Um exemplo de árvore binária.

Quando entramos na raiz, vemos que (J > H), assim vamos para a direita. O próximo, (J < R) nos manda para a esquerda; outra para a esquerda (J < K), e nos encontramos no módulo que contém "I". Já que "I" não tem nenhum módulo à sua esquerda e desde que (J > I), simplesmente colocamos "J" ao lado direito de "I". Os índices de árvore binária trabalham bem em aplicações reais especialmente quando o índice é pequeno o suficiente para ser carregado inteiro na memória, de uma só vez.

Índices de Árvore-B

Os índices de árvore binária são bons, porém começam a perder velocidade nas pesquisas à medida que os seus arquivos vão crescendo. A diminuição da velocidade ocorre pelo fato de o programa de gravação dos índices que acrescenta novas entradas de índices seqüencialmente ao arquivo de disco, ao considerar cada nova entrada para encontrar seu lugar, precisa olhar para no mínimo um indicador de registro, e é mais provável que tenha de olhar muitos. Lembre-se de que estamos discutindo sobre um índice armazenado em um arquivo de disco, cuja finalidade é indicar às novas entradas de índice, na verdade registros, onde elas poderão ser encontradas no disco. Desde que as chaves foram entradas aleatoriamente com os indicadores indo e voltando internamente, o arquivo de índice deve ser lido várias vezes para que a rotina de pesquisa encontre os endereços corretos para os indicadores de cada módulo sucessivo.

O próximo passo lógico depois das árvores binárias foi o desenvolvimento do índice de **árvore-B**, assim chamado porque um dos primeiros desenhos das árvores-B foi feito por um homem chamado Bayer (observe a legenda). As árvores-B trabalham como as árvores binárias, exceto que cada módulo, ou ponto de bifurcação, pode conter várias entradas e indicadores, como mostramos na Figura 2-4.

É muito mais complicado escrever um programa que faça novas entradas ao índice da árvore-B, já que os módulos que foram preenchidos tinham de ser quebrados em módulos pequenos, e isto pode ter uma repercussão nos níveis mais altos da árvore, causando a necessidade de introdução de módulos mais altos. De qualquer forma, a grande novidade é que as pesquisas das árvores-B são muito rápidas. O dBase II usa um tipo de índice chamado **índice de árvore-B***, e pode encontrar e mostrar um registro indexado em um arquivo de 65.535 registros dentro de 2.5 segundos, até mesmo com drives de disco flexíveis com velocidade relativamente lenta. Isto é rápido.

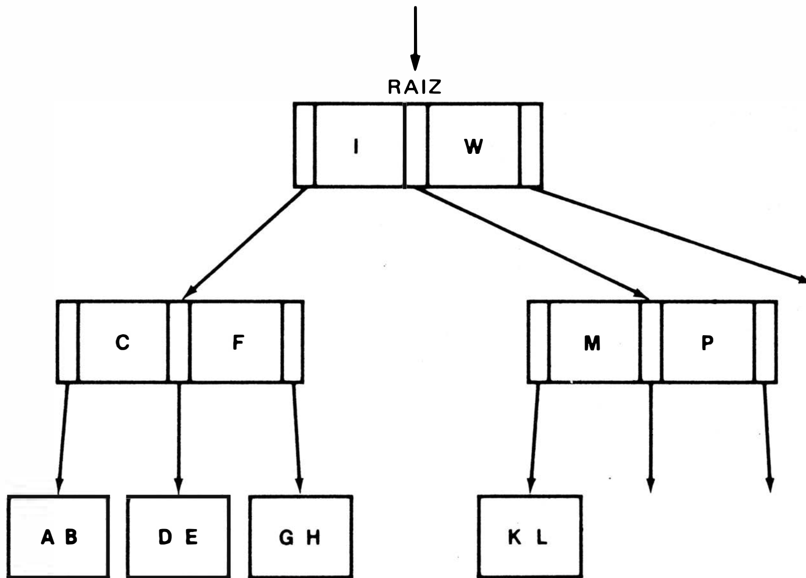


Figura 2-4 Um exemplo de uma árvore-B.

ANALISANDO UM ARQUIVO DE DADOS dBASE

O quadro que fizemos de um arquivo dBase até agora, é de um arquivo de dados feito de registros de extensão fixa que podem conter dados na forma de caracteres, números ou valores lógicos (verdadeiro/falso). Antes de iniciarmos o trabalho com o dBase a nível de comandos – isto é, fazer com que o programa funcione executando um comando de cada vez – vamos observar mais de perto a estrutura de um arquivo de banco de dados dBase.

Um arquivo dBase consiste em uma **estrutura** relativamente pequena (às vezes chamada **cabeçalho**) que contém um registro de nomes, extensões e tipos de dados de cada campo no registro e de uma seção de dados que contém os registros entrados pelo usuário. Enquanto a extensão da seção dos dados pode variar do registro zero até o número máximo (65.535 registros), cada arquivo de banco de dados deve ter uma estrutura. Esta estrutura pode ser considerada como o gabarito que o dBase carrega antes de trabalhar com os bancos de dados para que o programa saiba as regras do jogo que funciona para o conteúdo dos dados. Pela estrutura do dBase ser um arquivo normal de disco, ela pode ser copiada e movimentada da mesma maneira como qualquer outro arquivo de disco, usando PIP ou COPY; embora como veremos, o próprio dBase possui comandos amplos dentro do programa para copiar e manipular arquivos de banco de dados e suas estruturas.

Se fôssemos examinar a estrutura e os registros de um arquivo de dados dBase bem pequeno, poderíamos ver algo como na Figura 2-5.

```
NUMBER OF RECORDS:    00003
DATE OF LAST UPDATE:  00/00/00
FLD        NAME     TYPE   WIDTH   DEC
001      NOME          C     005
002      ENDEREÇO      C     015
003      CIDADE        C     015
004      ESTADO        C     002
005      CEP           C     005
006      VALOR         N     005     002
      **TOTAL**                 00049

00001      Mila   R. Elma   123 Sao Paulo   SP   92102   0,00   .T.
00002    * Jane   R. 22     22  Maceio      AL   10012  99,00   .F.
00003      Luiz   R. Oliva  85  Marilia     SP   55444   4,95   .F.
```

Figura 2-5. Estrutura e registros em típico arquivo .DBF.

Esta representação é um tanto idealizada – no arquivo verdadeiro de banco de dados, o cabeçalho não necessita de nenhum título como "FIELD", "WIDHT" ou "DATE OF LAST UPDATE". Porém, ela fornece uma boa imagem dos conceitos de uma estrutura de arquivo de banco de dados.

No topo do cabeçalho, o dBase armazena o número de registros gravados no arquivo de banco de dados e a data de sua última entrada. Esta data é a data do sistema, que é solicitada quando se carrega o CP/M (ou MS-DOS) e sempre está presente na memória do sistema operacional (embora não necessariamente correta).

A data do sistema e seu uso nos dois sistemas operacionais são discutidos com maiores detalhes no Capítulo 5.

Em seguida, a estrutura do registro é gravada armazenando os nomes dos campos na ordem que o usuário definiu. Cada tipo de dado do campo é gravado armazenando "C", "N" ou "L" para "caractere", "numérico" e "lógico". E finalmente a extensão do campo, e se ele for um número, suas casas decimais.

Na prática, o dBase reserva espaço suficiente no cabeçalho para definir 32 campos, não importa a quantidade especificada pelo usuário. Esta alocação de espaço fixo para a definição de campos é a razão fundamental que 32 campos por registro especifica o limite máximo em um arquivo de dados dBase.

Definidos os campos (e o cabeçalho), o dBase caminha para os dados. Nesta representação, cada um dos três registros possui um número de registro com 5 dígitos, porém, no arquivo físico de disco, nenhum dos números de registro são armazenados. O dBase anota a extensão do registro – a soma das extensões dos campos mais um byte de "controle" por registro. Em nosso exemplo, a extensão do registro é de 48 caracteres para os campos e o 49º para byte de controle. Os números de registro são muito importantes para a manipulação de dados no arquivo, mas eles tendem a tornar o dBase mais lento. Quando um usuário pede para que o dBase procure o registro número 3, o programa sabe que este registro iniciará no primeiro caractere depois do registro número 2 porque o dado é armazenado em registros de extensão fixa. Em nosso exemplo, sabemos que cada registro tem 49 caracteres. Portanto, ele conta $((2 \times 49) + 1)$ e está indicado o terceiro registro.

Na verdade, examinando o arquivo de disco de nosso banco de dados veríamos algo parecido com:

```
Mila    R. Elma, 123    Sao Paulo      S
P 92102  0,00 y*Jane    R. 22, 22      M
aceio        AL  10012   99,00  f Luiz  R
.Oliva, 85   Marilia    SP     55444
4,95 N
```

Muitas coisas são evidentes neste ponto. Primeiro, nos dois exemplos o dBase armazenou todos os dados de caracteres "ajustando à esquerda" dentro de seu campo ("justificando à esquerda"), e todos os dados numéricos como caracteres de dígitos "ajustando à direita". Isto é completamente aceitável para a maioria da intuição das pessoas. O campo lógico necessita um pouco mais de explicação. Na versão idealizada (Figura 2-5), o dBase armazenou o valor dos três registros como .T., F. e .F. Como mencionamos nos tipos de dados, os pontos antes e depois das letras T/F indicam que o dBase está dando a eles o valor verdadeiro/falso, diferente de um campo de um caractere de um byte contendo os valores "T" ou "F".

Para os programadores veteranos pode parecer surpresa, quando observarem que os valores gravados no arquivo de disco para os mesmos registros são "y", "f" e "N". O dBase trabalha diretamente e aceita oito tipos de letras para entrada de campos lógicos (TtFfYyNn) e grava realmente o que foi digitado. Nossa observação final é sobre o "byte" de "controle". Se formos

ver o registro 2 na Figura 2-5 – Maceió Jane, observaremos um asterisco (*) antes do primeiro campo. Este asterisco é uma marca de apagamento do dBase e o espaço para ele é fornecido em cada registro pelo byte e controle. Quando tratarmos da prática de comandos no Capítulo 3, veremos que marcar um registro de dados dBase para apagar, na verdade, não é retirá-lo do banco de dados. Ao contrário, o registro é marcado com asterisco e seus dados e valores são deixados de lado durante todas as manipulações e cálculos. Uma das grandes propriedades do dBase está nesta característica – os registros podem ser marcados para apagamento e depois chamados de novo.

PREVENDO O TAMANHO DE UM BANCO DE DADOS

Uma das perguntas mais importantes a fazer antes de desenhar um banco de dados ou até comprar um equipamento para montá-lo, é qual será o tamanho dos arquivos de dados. Obviamente em um ambiente de comércio pequeno onde os custos devem ser calculados contra o retorno, os ajustes podem ser feitos em determinadas peças de equipamento, mas não existe nenhuma maneira de ajustar o tamanho do banco de dados. Ou há lugar suficiente nos discos para nossos dados ou não há. O procedimento que os programas seguem para estimar a quantidade de armazenamento necessitada é simples. Eles criam ou no papel ou com o dBase, as estruturas para todos os arquivos do banco de dados. Calculam o total dos tamanhos dos campos em cada registro de arquivo de dados, somando um caractere para a marca de apagamento. Multiplicam cada tamanho pelo número previsto de registros para o período de ciclo do programa (ano fiscal, semestre, mês ou seja lá o que for) e conseguem estimar o tamanho daquele arquivo de dados. Depois somam todos os tamanhos dos arquivos de dados e dividem por 1000 (1024 se forem binários) e conseguem a quantidade de armazenamento aproximada necessária em "K", a abreviação usual para milhares de caracteres.

Os programadores experientes levam este processo mais adiante. Eles sabem que alguns cuidados a mais precisam ser tomados. Um espaço necessita ser colocado para arquivos de classificação (eles são do mesmo tamanho que os arquivos fonte, lembra-se?), temporários e índices extras. Além disso, os clientes gostam de ter certeza de poder expandir os seus programas. O que inicia como um pequeno programa de controle de caixa do escritório, pode crescer lateralmente até que manipule também o livro razão geral da firma, contas a receber e pagar e as tabelas de câmbio para o escritório em outro país. A carreira de programador pode ser construída ou destruída a partir destes eventos.

Assim cada programador possui um fator que varia conservadoramente de 1.5 até 10 pelo qual ele multiplica o espaço de disco previsto que ele necessita. O bom senso ajuda. Se pensamos que podemos fixar nossos dados em 3 megabytes (milhões de bytes), então 5 megabytes seriam o suficiente. Dois "megas" é bastante espaço extra. Por outro lado, se o cálculo final estimado for 150K (milhares de bytes), isso não quer dizer que com 250K nós vamos poder brincar à vontade. O índice é o mesmo, mas a experiência nos ensina que 100K de respiração no segundo exemplo é quase nada.

CAPÍTULO

3

O Modo Interativo (I)

Nos próximos três capítulos, os comandos dBase mais usados serão demonstrados no modo interativo. Os comandos estão agrupados logicamente e é seguida uma progressão natural, assim nenhum comando será introduzido sem que antes tenha sido criado um fundamento para ele. Na prática, alguns comandos como o CREATE podem ser usados muito facilmente e devem introduzir desde já, porém, alguns também possuem uma aplicação refinada (CREATE FROM) que pertence à seção das técnicas avançadas. Outros comandos como LIST possuem muitos **argumentos** adicionais – que se unem à linha de comando e que modificam sua ação – isto será discutido nestes três próximos capítulos. Nestes Capítulos 3, 4 e 5, acreditamos que o leitor já tenha adquirido um computador e a sua cópia do dBase. Estes capítulos foram divididos em "operações" do dBase, ou exercícios nos quais as séries dos comandos dBase são discutidas e descritas como amostras dos resultados.

Independentemente da clareza com que um comando é colocado neste livro, ele não vai conseguir funcionar sozinho. Experimentadores aprendem usar um computador mais rápido do que os leitores, e pessoas que experimentam com o livro ao lado aprendem mais rápido ainda. Estes capítulos foram desenhados como um livro de exercícios para estimular a prática.

ANTES DE INICIAR

Este livro não substitui a documentação do dBase nem o manual do computador. O dBase está sendo usado em uma variedade de versões, de 2.3 A, B, C e D através do 2.6, em dois sistemas operacionais, CP/M e MS-DOS (PC-DOS).

Portanto, especialmente se você for um principiante.

LEIA SUA DOCUMENTAÇÃO ANTES DE TESTAR SUA MÁQUINA OU DISCO dBASE

A maioria dos manuais possui uma seção chamada "Como Iniciar" que ao menos deve ser compreendida antes de iniciar este livro.

OS ARQUIVOS dBASE

Quando revirmos a seção em nossa documentação do dBase de como iniciar, provavelmente encontraremos uma lista dos arquivos apresentadas no disco original. Este disco contém o programa de instalação, uma demonstração do programa dBase (geralmente um programa de checagem de equilíbrio), e os próprios arquivos dBase.

Dependendo da versão que possuímos, estes arquivos dBase distribuem-se em números de dois a doze e todos devem estar no mesmo disco. Por exemplo, a versão 2.3 (CP/M) possui um arquivo de "programa raiz" chamado DBASE.COM e 12 overlays (módulos com nomes tais como DBSORT.OVR. A versão posterior (2.4) foi expedida com o nome da raiz DBASE-II.COM e apenas um módulo chamado DBASEOVR.COM. Nos dois casos o programa funciona da mesma forma.

CARREGANDO O dBASE

O dBase é carregado solicitando-se o nome do programa no prompt do sistema operacional. Assumindo que o dBase esteja no drive A, isto poderia ser feito digitando:

```
A>DBASE<cr>  ou  A>DBASE-II <cr>
```

qualquer um dos nomes está correto.

Durante sua operação, o dBase fará referências temporárias às partes do programa armazenadas nos módulos de superposição no drive do disco. Já que o dBase tem de saber onde encontrar as superposições, o programa inteiro – isto é, a raiz e todos os módulos – devem estar no disco cujo drive é indicado pelo prompt. Não podemos carregar o dBase digitando um comando como:

```
B>A:DBASE  <cr>  (*Não funcionará*)
```

O sistema de operação voltará ao drive A para o arquivo DBASE.COM, mas quando o primeiro módulo é chamado, o dBase o procurará no drive B.

Além disso, podemos causar um grave problema se decidirmos renomear qualquer dos módulos do dBase. Quando esta função for chamada, o dBase vai procurar pelo nome original. E não vai achar. É proibido renomear os módulos. Se, por outro lado, queremos renomear o arquivo raiz, isto é possível desde que não se mexa no tipo do arquivo (as últimas três letras depois do ponto) – .COM em CP/M e .CMD em MS-DOS.

Isto é muito mais uma questão de preferência. Por exemplo, podemos abreviar o nome DBASEII.COM para DB.COM e isso vai, além de tudo, facilitar a digitação na hora da carga. Agora podemos digitar:

```
A>DBASE  <cr>
```

e veremos uma mensagem protocolo da nova versão, seguida de um pedido para a entrada de uma nova data. Por enquanto, (até o Capítulo 5), omitiremos a entrada da data pressionando a tecla return, e o dBase nos indica que chegamos ao modo interativo, com um ponto (.).

A ESTRUTURA DOS COMANDOS dBASE

A maioria dos comandos dBase possuem a mesma **sintaxe**, ou estrutura – sentenças que consistem em um verbo no imperativo seguido de substantivos e frases. A sintaxe dos comandos é de fácil memorização, entretanto, ela é uma linguagem de computador e como tal, espera que os comandos sejam informados corretamente.

A sintaxe de comando dBase deve ser descrita como:

VERBO	ESCOPO	SUBSTANTIVO(s)	CONDIÇÃO(ções)

Um exemplo típico poderia ser:

.display	all	sobrenome	for cidade = ''Ilheus''
VERBO	ESCOPO	SUBSTANTIVO(s)	CONDIÇÃO(ções)

Se tentarmos este exemplo agora, vamos receber de volta um monte de mensagens de erro. O dBase não pode mostrar nenhum dado sem possuir um banco de dados em uso de onde possa retirá-lo.

Todas as partes de todos os comandos podem ser digitadas em letras maiúsculas ou minúsculas sem afetar o entendimento do dBase, exceto para variáveis de strings. No exemplo acima, o mesmo comando poderia ser encontrado com:

```
.DISPLAY ALL SOBRENOME FOR CIDADE = ''Ilheus''
```

O operador pode alterar as letras maiúsculas ou minúsculas nos comandos, sem alterar a função – "dIsPLAy", como um exemplo horrível – mas a diferença entre

```
cidade = "Ilheus"   e   cidade = "ILHEUS"
```

é profunda (veja no Capítulo 1 o tipo de dados caractere).

A ordem dos elementos de um comando dBase pode ser trocada, embora não arbitrariamente. As regras são:

1. O verbo deve iniciar no primeiro caractere da linha, sem espaços à esquerda.
2. Todas as palavras dBase – verbos, escopos, preposições e assim por diante – podem ser abreviadas às suas primeiras quatro letras, mas devem ser corretamente escritas. Todas as palavras do usuário – nomes de campo, nomes de variáveis de memória, nomes de arquivos de disco ou qualquer outra indicação definida pelo usuário – devem ser escritas completa e corretamente.
3. Todas as palavras em uma linha de comando devem ter pelo menos uma tabulação, espaço ou marca de pontuação não alfanumérica (outra que não seja os dois pontos) separando-as. Em nomes definidos pelo usuário, dois pontos (:) são considerados como letras (p.e., TAX:RATE e TAXRATE são nomes de variáveis diferentes e legais).
4. Uma linha de comando é terminada pela digitação da tecla *return*. Se um programador deseja quebrar um simples comando em duas linhas em um programa, a primeira linha pode ser terminada com ponto e vírgula (;), dizendo ao dBase para ignorar a tecla *return*.

```
{inicio de linha de comando } ; <cr>
{o resto de uma linha de comando } <cr>
```

Se as regras forem seguidas, o dBase aceita. Qualquer um destes comandos funcionará:

```
DISPLAY FOR CIDADE = "Ilheus" NOME ALL
            e
.DISPLAY ALL NOME FOR CIDADE = "Ilheus"
```

A única vantagem do segundo caso é que ele é mais claro. Mas que vantagem!

SOBRE AS OPERAÇÕES

O resto deste capítulo e os próximos dois estão divididos em sete "operações" ou seções do dBase. Cada seção foi destinada a demonstrar as funções de cinco a doze verbos dBase com seus vários **argumentos**, um termo emprestado do significado matemático "instruções ou indicações adicionais".

Quando cada comando for apresentado, seus **defaults** e argumentos aceitáveis serão explicados. Em computação, os defaults descrevem a maneira que um comando atuará se o usuário preferir deixar um argumento de lado. Por exemplo, no comando

```
.DISPLAY ALL NOME FOR CIDADE = "Ilheus"
```

o que poderia acontecer se o escopo "ALL" fosse retirado? E se o substantivo "NOME" fosse pulado? Nos dois casos as trocas possuem um efeito específico que seriam capazes de prevenir depois de cada discussão do DISPLAY e seu default.

Usaremos a convenção da linguagem Pascal para diferenciar os comentários da operação real do computador – entre parênteses com asteriscos, como em (*ISTO É UM COMENTÁRIO*). Esta é puramente uma convenção deste livro e NÃO FUNCIONARÁ EM dBASE!

Cada seção será construída com base na anterior, e como conseqüência, os exercícios necessitam de um objetivo comum para que sejam coesos. Nosso projeto, portanto, será construir e manipular um banco de dados para controlar os associados de um clube de computadores fictício (todos os clubes de computação são, de um grau ou de outro, fictícios).

SEÇÃO 1 – CRIAÇÃO, ENTRADA DE DADOS, VISUALIZAÇÃO DE DADOS E APAGAMENTO

A proposta da primeira seção é criar um banco de dados de sócios e inserir alguns registros nele. Depois, vamos manipular os dados algumas vezes e em seguida apagar e recuperar alguns registros. A ênfase aqui é para a demonstração da sintaxe correta do dBase no modo interativo. Os comandos que veremos são:

HELP	<Esc>
CREATE	DISPLAY
USE	DELETE
DISPLAY STRUCTURE	RECALL
APPEND	PACK
LIST	QUIT
Ctrl-S	

HELP

Nas versões 2.3 e anteriores não encontrávamos o comando HELP. Se não soubéssemos o que fazer em uma operação, manusearíamos a documentação para observar as sugestões ou até telefonaríamos para alguém que provavelmente saberia fornecer a resposta. Como resposta a várias perguntas, a Ashton-Tate colocou a facilidade do HELP na versão 2.4.

A sintaxe do HELP é:

```
.HELP      ou      .HELP <comando>
```

Os dois modelos do comando alcançam o disco default e abrem um arquivo de texto de descrições dos comandos dBase. HELP sem nenhum argumento fornece uma lista alfabética de todos os comandos dBase com descrições em uma linha. Quando HELP é seguido de um verbo de comando específico, como em "HELP LIST", é mostrado um parágrafo com uma página de informações sobre aquele comando.

Há uma surpresa, o texto para todas estas mensagens está num arquivo chamado "DBASEMSG.TXT", que deve estar presente no mesmo drive que dBase, e ocupa aproximadamente 53K bytes de espaço de disco. Pessoas que operam dBase em computadores com drives de disco pequenos, tais como o Osborne I ou Apple II-CP/M, não possuirão lugar para o arquivo HELP inteiro e terão de usar uma versão abreviada (que deve ser renomeada para DBASEMSG. TXT). Esta versão menor pode somente responder ao HELP sem argumento e mostrar a lista de comandos.

Esta é a única discussão do comando help que aparecerá neste livro. A facilidade do HELP é muito útil durante o período de aprendizagem de um novo usuário, mas os programas dBase raramente ou nunca usam HELP, visto que o programa deve ter prevenido cada possível problema.

Como exercício digite:

```
.HELP <cr>
```

e depois (se possuirmos a forma longa do DBASEMSG.TXT):

```
.HELP CREATE <cr>
```

O comando HELP combinado com o arquivo grande DBASEMSG.TXT também chamará as telas de informações para as seguintes palavras-chaves (como em ".HELP UTILITIES <cr>"):

UTILITIES	FULL-SCREEN	LIMITS	BACKUP
INSTALL	NEW	ERRORS	CP/M
EXAMPLES	FUNCTIONS	DBASE	HELP
RUNTIME			

O Diálogo de Correção de Comandos

Alguém digitou naquele exercício "HLEP"? Se digitou, já descobriu o **diálogo de correção de comando,** uma das características mais maravilhosas do dBase (embora às vezes frustrante). Aqui está como funciona: muitas vezes, cometeremos erros ao digitar os comandos. Alguns enganos não interessam ao dBase, tal como erros de ortografia em série ("Ilhous" em vez de "Ilheus"). Outros, como "HELP" acima confundem o dBase. Lógico que seria mais fácil para o programa mostrar o comando errado, para que pudéssemos ver o nosso erro e em seguida voltar ao prompt, mas neste caso teríamos de entrar novamente toda a linha de comando. Acontece que 50 por cento dos enganos são cometidos entre 10 da noite e 4 horas da manhã, não há garantia que a segunda tentativa seja melhor que a primeira.

Portanto, a menos que seja eliminada especificamente durante a instalação, o dBase oferece ao usuário a opção de corrigir a linha que o programa não pode entender. Aqui está uma operação simples com uma grafia incorreta de CREATE.

```
.CREAET SOCIOS                      (*Escrita errada*)
***UNKNOW COMMAND                   (*Resposta do dBase*)
CREAET SOCIOS
CORRECT AND RETRY (Y/N) ? Y         (*Digitamos Y para sim*)
CHANGE FROM: ET                     (*"ET" é o erro*)
                                    (*que queremos trocar*)
CHANGE TO    : TE                   (*para "TE".*)
CREATE SOCIOS                       (*O dBase mostra a nova versão*)
MORE CORRECTIONS (Y/N) ?N           (*e pergunta se desejamos mais*)
                                    (*correções*)
```

Depois que dissermos ao dBase que não há mais correções, a linha de comando editada será submetida a execução. Se estiver tudo correto, o dBase executará. Do contrário, a linha será solicitada novamente para as possíveis correções. Se, em qualquer pergunta CORRECT AND RETRY tivéssemos digitado "N", o dBase teria voltado ao prompt.

Mais de uma parte de uma linha de comando pode ser corrigida uma vez que ela foi submetida ao diálogo de correção. Vamos considerar os seguintes remendos:

```
. CREAET SOICOS            (*" SOICOS"?"*)
*** UNKNOWN COMMAND
CREAET SOICOS
CORRECT AND RETRY (Y/N)? Y
CHANGE FROM :ET
CHANGE TO   :TE
CREATE SOICOS
MORE CORRECTIONS (Y/N)? Y
```

```
CHANGE FROM :IC
CHANGE TO   :CI
CREATE SOCIOS
MORE CORRECTIONS (Y/N)? N
```

Saímos e estamos operando. Perceba neste exemplo que a razão sólida para o dBase devolver este comando, foi que ele não reconheceu o verbo ("CREAET") no comando. Foi um benefício acidental que permitiu corrigir a escrita do nome do nosso arquivo de banco de dados antes de submeter o comando ao dBase.

Se tivéssemos entrado:

```
.CREATE SOICOS  <cr>
```

então teríamos com um banco de dados chamado SOICOS.DBF que teria a necessidade de ser renomeado mais tarde.

CREATE

O comando CREATE em sua forma mais simples é usado para definir a estrutura dos registros de um novo banco de dados. Quase sempre o CREATE é usado interativamente – isto é, o dBase faz ao usuário uma série de perguntas e então forma um arquivo de banco de dados a partir das respostas.

O comando CREATE precisa de um nome para o novo banco de dados. Se nenhum foi informado como argumento, o dBase emitirá uma mensagem, pedindo por ele. O nome do arquivo não pode exceder a oito posições, e deve contar somente os caracteres permitidos conforme o sistema operacional. Por exemplo, o MS-DOS reserva para si a barra e a barra invertida ("/ " e "\"). Se o programador não especificar o tipo do arquivo (o sufixo dos três caracteres que segue o ponto), o dBase fornece o tipo default, ".DBF.". Se o programador fornecer um tipo, como ".SAM" ou ".001", então o dBase nomeará o arquivo de banco de dados de acordo com o desejo do programador – mas o arquivo deve sempre ser referido ao seu nome completo para que o dBase o reconheça.

Vamos fazer uma simples operação de criação de um arquivo de banco de dados para o nosso clube de computação imaginário. A estrutura do registro deveria ter campos para cada nome e sobrenome dos sócios, endereço, cidade, estado e código postal. Já que todos eles serão do tipo de dado caractere (até códigos postais – veja o Capítulo 1 sobre tipos de dados), colocaremos um campo numérico para armazenar os dados referentes às contribuições para o clube e um campo lógico para gravar se as contribuições foram pagas ou não.

```
. CREATE SOCIOS                    (*Preparamos o default SOCIOS.DBF*)
ENTER RECORD STRUCTURE AS FOLLOWS:
 FIELD   NAME,TYPE,WIDTH,DECIMAL PLACES
```

```
 001     NOME,C,10
 002     SOBRENOME,C,15              (*O dBase fornece os números dos campos, *)
 003     ENDERECO,C,20               (*e nós digitamos o resto*)
 004     CIDADE,C,10
 005     ESTADO,C,2
 006     CEP,C,5
 007     CONTRIBUICOES,N,5,2
BAD NAME FIELD                       (*Isto acontece quando fazemos um*)
 007     CONTRIB,N,5,2               (*erro na declaração de um campo*)
 008     PAGAMENTO,L
 009
INPUT DATA NOW? N                    (*E volta ao prompt*)
```

Vamos fazer algumas observações sobre este CREATE. Primeiro, vejamos a linha de indicação que o dBase fornece. Ela nos lembra a ordem de entrada da definição – nome, tipo, largura e casas decimais – mas não são todos os tipos de dados que necessitam ter uma linha completa. Os caracteres e os dados lógicos não necessitam das casas decimais e nem da largura – nem poderia, visto que ele apenas pode ter um T ou um F.

Os nomes dos campos, e como veremos mais tarde, os nomes da variável de memória, devem ser escolhidos de acordo com as seguintes regras:

1. Um nome nunca pode ter mais do que dez caracteres alfanuméricos e ele deve iniciar com um alfa. Todos os alfas podem entrar como maiúsculas ou minúsculas, mas são tratados somente como maiúsculas.
2. Na versão 2.4 e na posterior, mais de um dois pontos (:) podem ser usados para tornar os nomes mais legíveis, mas cada dois pontos é contado para o limite de dez caracteres. Na versão 2.3 e anteriores, somente um dois pontos (:) é permitido.

As duas próximas colocações não são regras, mas são boas práticas de programação. Anote.

3. Limitar os nomes dos campos para nove caracteres, reservando o décimo como uma indicação para as variáveis de memória derivadas do campo.
4. Estabeleça consistência em seus nomes de campos no início e mantenha-os. Se chamar o campo primeiro-nome de FIRST:NAME em um arquivo de dados, não o chame de F:NAME em outro e FST:NAM em outro ainda. O dBase permite uma rapidez extrema na comunicação entre arquivos de dados, mas isto depende dos nomes dos campos serem os mesmos em todos os bancos de dados.

Uma última colocação sobre os campos numéricos: a declaração de largura com cinco posições deve incluir espaço para todos os dígitos mais um espaço para o ponto decimal. Como CONTRIB foi definido, a maior contribuição que ele pode conter é $99,99. Não podemos tratá-lo deixando os centavos de lado. Se tentarmos inserir "120", imaginando ocupar apenas três caracteres, o dBase completará com ".00", empurrando a entrada para além de seis caracteres, causando estouro de capacidade no campo.

Quando declaramos a largura para campos numéricos, devemos deixar lugar suficiente para todos os dígitos, o ponto decimal e um dígito extra para segurança.

USE

No comando CREATE acima, o dBase perguntou no final se queríamos entrar com os dados imediatamente e nós contestamos. Neste caso, o dBase simplesmente fechou o arquivo de dados vazio no disco e voltou para o prompt. "Fechou"? Em computação, os arquivos de disco, sejam eles arquivos de dados ou algum outro tipo, necessitam ser abertos quando o usuário quer lê-los ou alterá-los e fechados quando o processo estiver terminado. "USE" é o comando dBase para abrir e fechar bancos de dados.

Ele possui duas formas comuns:

```
.USE <nome do arquivo> <cr> (*Abre o arquivo*)
.USE <cr>   (*Fecha o arquivo de dados em uso*)
```

Antes que qualquer arquivo de dados possa ser lido, alterado ou escrito ou manipulado de qualquer forma, ele deve ser aberto com USE. Dois terços de todos os comandos dBase operam no arquivo em uso. Mais tarde, quando virmos

```
.COPY TO XYZ  <cr>
```

não pergunte "Copiar o quê?" Sempre será entendido que os comandos afetam os arquivos em uso.

Vamos abrir (USE) o nosso arquivo dos sócios:

```
.USE SOCIOS        (*Isto abre o arquivo de dados*)
```

O comando USE não possui efeito visível. SOCIOS.DBF foi aberto agora com o registro do topo (se tiver algum) pronto para ser mostrado. Se quiséssemos fechar SOCIOS, poderíamos USE outro arquivo de dados ou somente dizer ".USE".

DISPLAY (LIST) STRUCTURE

DISPLAY STRUCTURE mostra a imagem da estrutura dos registros para SOCIOS. Como podemos ver, ela está na mesma forma de quando foi usado anteriormente para mostrar o cabeçalho de arquivo de dados dBase. Neste caso, para o arquivo definido com CREATE, DISPLAY STRUCTURE lista as informações do cabeçalho do arquivo de dados, e mais nada.

```
. DISPLAY STRUCTURE                    (*Teríamos o mesmo efeito com "LIST STRUCTURE")
STRUCTURE FOR FILE:  B:SOCIOS  .DBF        (*O dBase fornece esta*)
NUMBER OF RECORDS:   00000                 (*figura de SOCIOS*)
DATE OF LAST UPDATE: 00/00/00
PRIMARY USE DATABASE
FLD      NAME      TYPE WIDTH   DEC
001     NOME          C    010
002     SOBRENOME     C    015
003     ENDERECO      C    020
004     CIDADE        C    010
005     ESTADO        C    002
006     CEP           C    005
007     CONTRIB       N    005     002
008     PAGAMENTO     L    001
** TOTAL **              00069
```

APPEND

O comando para incluir informações dentro de um arquivo de dados é o APPEND. O comando APPEND aceita uma variedade de argumentos e permite grande flexibilidade no processo de entrada de dados; porém, por enquanto mostraremos um APPEND simples que acrescenta registros ao arquivo de dados em uso um por um.

```
RECORD # 00001
NOME      :Mila      :                (* Quando inserimos cada campo *)
SOBRENOME :Moreira        :           (*a tecla cr leva o cursor     *)
ENDERECO  :R.Elma, 111         :      (*para o próximo campo.        *)
CIDADE    :Sao Paulo :                (*Quando completamos um campo  *)
ESTADO    :SP:                        (*o dBase emite um sinal sonoro. *)
CEP       :92109:
CONTRIB   : 5.00:
PAGAMENTO :y:
```

(* Aqui a tela desaparece e surge uma nova estrutura do APPEND*)

```
RECORD # 00002
NOME      :Ana       :
SOBRENOME :Amorim         :
ENDERECO  :R.Tres, 333         :
CIDADE    :Natal     :
ESTADO    :RN:
CEP       :01224:
CONTRIB   : 0.00:
PAGAMENTO :T:
```

```
RECORD # 00003
NOME      :Carla    :
SOBRENOME :Cardoso       :
ENDERECO  :R.Bel Monte, 222     :
CIDADE    :Belo Horiz:
ESTADO    :MG:
CEP       :20105:
CONTRIB   : 0.00:
PAGAMENTO : :
```

Para sairmos do comando APPEND, pressionamos a tecla <cr> no momento em que um novo registro aparecer na tela. Se desejamos fechar o APPEND no meio de um registro mal preenchido, usamos um ^Q em qualquer lugar do registro. As outras teclas de controle serão mencionadas na próxima seção.

Atenção nos seguintes pontos do comando APPEND. No registro 3, "Belo Horizonte" foi truncado porque o campo era muito pequeno (aprenderemos a consertar detalhes como este mais tarde), e o dBase preencheu CONTRIB com o valor zero quando pressionamos a tecla <cr> no campo vazio. O campo lógico PAGAMENTO foi deixado vazio e será considerado como se tivesse um valor .F. (falso).

O comando APPEND é o nosso meio de transporte para a entrada de dados; devemos experimentá-lo até nos sentirmos confiantes para usá-lo.

Controles de Edição em Tela-Cheia (Full-Screen)

O que aconteceria se estivéssemos usando o comando APPEND e, estando no segundo campo, percebêssemos um erro no primeiro campo – NOME "Mola" no lugar de "Mila" ou algo parecido com isto?

O dBase permite a edição dos dados, quando usamos APPEND, EDIT e muitos outros comandos que usam a tela-cheia como meio de comunicação com o usuário. Os controles de edição em tela-cheia possuem as mesmas funções das teclas de controle do WordStar e outros programas CP/M. Neste livro, "Ctrl" ou "^" referem-se à tecla *control*.

Controles do Cursor

Ctrl-X move o cursor para baixo
(Ctrl-F faz a mesma coisa).
Ctrl-E move o cursor para cima
(Ctrl-A faz a mesma coisa).
Ctrl-D move um caractere à frente
Ctrl-S move o cursor um caractere para trás.

Controles de Apagamento

Ctrl-G apaga o caractere sob o cursor.
<Rubout> ou <DEL> apaga o caractere à esquerda do cursor.
Ctrl-Y apaga o campo à direita do cursor.
Ctrl-V ativa/desativa o modo de INSERÇÃO.
Ctrl-W salva qualquer alteração efetuada e volta ao dBase.
Ctrl-Q sai do registro sem salvar e volta ao dBase.

Usando as Teclas de Controle

Para quem não conhece estes controles, a forma mais fácil de memorizá-los é usar suas localizações como um mnemônico – E, X, D e S formam uma cruz no teclado-padrão de uma máquina de datilografia como se fossem "Norte, Sul, Leste, Oeste".

^E – sobe uma linha/campo
^S – um caractere à esquerda ^D – um caractere à direita
^X – desce uma linha/campo

Esta é uma boa oportunidade para fazermos um APPEND para SOCIOS, utilizando as teclas de controle a fim de memorizá-las.

O dBase usa estas teclas para edição em muitos lugares e se praticarmos agora nos sentiremos mais familiarizados. Inclua: Sonia Simões; Av. Pará, 831; Belém; PA; CEP 21201; 0,00.

LIST

Quando o arquivo de dados em uso possui dados, eles podem ser visualizados usando o comando LIST.

```
. LIST
00001  Mila    Moreira   R.Elma, 111         Sao Paulo  SP 92109  5.00 .T.
00002  Ana     Amorim    R.Tres, 333         Natal      RN 01224  0.00 .T.
00003  Carla   Cardoso   R.Bel Monte, 222    Belo Horiz MG 21105  0.00 .F.
00004  Sonia   Simoes    Av.Para, 831        Belem      PA 21201  0.00 .F.
```

Se for necessário apenas um campo do arquivo de dados, este pode ser especificado.

```
. LIST NOME
00001 Mila
00002 Ana
00003 Carla
00004 Sonia
```

Vários campos também podem ser listados usando vírgulas para separá-los.

```
. LIST NOME,SOBRENOME,CIDADE
00001  Mila      Moreira       Sao Paulo
00002  Ana       Amorim        Natal
00003  Carla     Cardoso       Belo Horiz
00004  Sonia     Simoes        Belem
```

Não há restrições em relação à ordem que os campos são mencionados em um comando LIST ou quantas vezes um campo pode ser mencionado.

```
. LIST SOBRENOME,CIDADE,CEP,CIDADE,PAGAMENTO,CIDADE
00001  Moreira          Sao Paulo  92109 Sao Paulo  .T. Sao Paulo
00002  Amorim           Natal      01224 Natal      .T. Natal
00003  Cardoso          Belo Horiz 21105 Belo Horiz .F. Belo Horiz
00004  Simoes           Belem      21201 Belem      .F. Belem
```

Se um ^P (Ctrl-P) for digitado, o comando LIST pode enviar o banco de dados para a tela e para a impressora. (A impressora deve estar ligada, caso contrário travará o computador.) O ^P é uma **alavanca**, isto é, um comando que liga e desliga uma característica, como um interruptor. O primeiro ^P liga a impressora, o segundo desliga e assim por diante.

^S e <ESC>

Listar um arquivo de dados com três ou quatro registros é relativamente fácil. Eles são feitos em um instante e logo preenchemos a tela. Listar arquivos de dados com centenas ou até milhares de registros, porém, pode levar tempo e algumas vezes o usuário necessita encontrar uma forma de parar a listagem.

Em LIST e na maioria dos comandos de grandes operações do dBase, a tecla escape (simbolizada <esc>) cancelará o comando e voltará o usuário para o prompt. Em outras ocasiões, o problema não implica o usuário desejar abortar a listagem, ele quer somente "parar" a listagem por um momento. Nos casos em que o dBase está mandando materiais para a tela, podemos congelar o processo digitando um ^S. Como ^P, o ^S é uma alavanca na qual ao pressionar outro ^S (ou neste caso qualquer outra tecla) a listagem continuará sendo emitida.

Se tivermos registros suficientes em nosso SOCIOS.DBF, podemos tentar parar uma listagem com ^S ou cancelar com <esc>.

DISPLAY

Na colocação anterior sobre LIST STRUCTURE, dissemos que os comandos LIST e DISPLAY são parecidos e permutáveis. Na verdade, em muitos comandos dBase que detalharemos mais tarde tais com LIST STATUS ou LIST FILES, LIST e DISPLAY têm precisamente o mesmo significado.

A única diferença entre os dois comandos está nas ações de seus defaults quando eles são usados como simples verbos (verbos usados sozinhos sem nenhum argumento). O LIST vai ao topo do arquivo de dados e mostra cada registro em ordem até que alcance a base ou seja interrompido. O DISPLAY mostra o registro corrente e pára. No exemplo que segue, note que DISPLAY aceita os mesmos argumentos que LIST, mas faz com que tudo funcione em um único registro, sem mover o indicador.

```
. DISPLAY
00004 Sonia      Simoes           Av.Para, 831        Belem      PA 21201   0.00 .F.

. DISPLAY NOME
00004 Sonia



. DISPLAY NOME,CIDADE,CONTRIB,CIDADE,SOBRENOME,CIDADE
00004 Sonia      Belem        0.00 Belem      Simoes          Belem
```

Esta consideração será examinada melhor na Seção 2 – a movimentação dentro de um arquivo de dados. Até agora aprendemos somente duas maneiras de movimentar o indicador dentro do arquivo. Quando USE um arquivo (ou re-USE o arquivo em uso), levamos o indicador para o primeiro registro.

Quando usamos o comando LIST, após a conclusão da listagem, o indicador está na base. Falta ainda explorar a fase de direcionar o indicador para qualquer registro entre os dois extremos.

DELETE

Quando discutimos o comando CREATE, vimos que a estrutura de um registro é na verdade um byte ou caractere a mais do que a soma de seus campos. Naquela ocasião, dissemos que o byte extra é usado para armazenar um asterisco quando um registro é marcado para o apagamento.

O comando DELETE é usado para colocar a marca de apagamento (um asterisco) no campo reservado para isto na estrutura do registro. Como o comando DISPLAY, o DELETE atua no registro corrente – usando o comando DELETE sem qualquer outra condição e o dBase apaga o registro que veríamos se usássemos DISPLAY.

```
. LIST
00001  Mila     Moreira    R.Elma, 111         Sao Paulo  SP 92109   5.00 .T.
00002  Ana      Amorim     R.Tres, 333         Natal      RN 01224   0.00 .T.
00003  Carla    Cardoso    R.Bel Monte, 222    Belo Horiz MG 21105   0.00 .F.
00004  Sonia    Simoes     Av.Para, 831        Belem      PA 21201   0.00 .F.

. DISPLAY
00004  Sonia    Simoes     Av.Para, 831        Belem      PA 21201   0.00 .F.

. DELETE
00001 DELETION(S)

. DISPLAY
00004 *Sonia    Simoes     Av.Para, 831        Belem      PA 21201   0.00 .F.

. LIST
00001  Mila     Moreira    R.Elma, 111         Sao Paulo  SP 92109   5.00 .T.
00002  Ana      Amorim     R.Tres, 333         Natal      RN 01224   0.00 .T.
00003  Carla    Cardoso    R.Bel Monte, 222    Belo Horiz MG 21105   0.00 .F.
00004 *Sonia    Simoes     Av.Para, 831        Belem      PA 21201   0.00 .F.
```

O comando DELETE pode parecer confuso em primeira análise. Poderíamos perguntar por que o registro ainda permanece depois que o apagamos. Por duas razões: a primeira, é que o dBase escreve seus registros no disco em um arquivo seqüencial. Se um registro fosse apagado no meio de um arquivo de dados, não haveria nenhuma maneira de encontrar os outros registros gravados além daquele ponto. A segunda razão é até mais prática. Às vezes as pessoas apagam os registros por acidente; marcando os registros para apagamento em vez de destruí-los, o dBase permite a checagem e verificação.

A propriedade de marcar registros para apagamento oferece ao usuário uma chance de verificação. Quando chegarmos em mais comandos, verificaremos que muitos deles (SUM, COPY e APPEND FROM são exemplos) ignoram os registros marcados para apagamento. Quando estivermos mais experientes, encontraremos muitas ocasiões nas quais marcamos um registro ou um grupo deles para o apagamento, eles serão colocados de lado temporariamente, de forma que outro comando possa ser executado sem mexermos com eles.

RECALL

Se o apagamento pode ser temporário, parece razoável existir um comando que "desapague" os registros. Há este comando chamado RECALL. Como DELETE, ele atua no registro corrente.

```
. DISPLAY
00004 *Sonia     Simoes      Av.Para, 831          Belem       PA 21201   0.00 .F.

. RECALL
00001 RECALL(S)

. LIST
00001  Mila      Moreira     R.Elma, 111           Sao Paulo   SP 92109   5.00 .T.
00002  Ana       Amorim      R.Tres, 333           Natal       RN 01224   0.00 .T.
00003  Carla     Cardoso     R.Bel Monte, 222      Belo Horiz  MG 21105   0.00 .F.
00004  Sonia     Simoes      Av.Para, 831          Belem       PA 21201   0.00 .F.
```

PACK

Parece também razoável que se os registros podem ser marcados para o apagamento, deve haver um comando para livrar-nos deles para sempre e este comando é o PACK. O comando PACK é ESTRATÉGICO e deve somente ser usado quando estamos certos de que não queremos os registros marcados para o apagamento novamente.

```
. DISPLAY
00004  Sonia     Simoes      Av.Para, 831          Belem       PA 21201   0.00 .F.

. DELETE
00001 DELETION(S)
. DISPLAY
00004 *Sonia     Simoes      Av.Para, 831          Belem       PA 21201   0.00 .F.
. LIST
00001  Mila      Moreira     R.Elma, 111           Sao Paulo   SP 92109   5.00 .T.
00002  Ana       Amorim      R.Tres, 333           Natal       RN 01224   0.00 .T.
00003  Carla     Cardoso     R.Bel Monte, 222      Belo Horiz  MG 21105   0.00 .F.
00004 *Sonia     Simoes      Av.Para, 831          Belem       PA 21201   0.00 .F.
. PACK
PACK COMPLETE, 00003 RECORDS COPIED
. LIST
00001  Mila      Moreira     R.Elma, 111           Sao Paulo   SP 92109   5.00 .T.
00002  Ana       Amorim      R.Tres, 333           Natal       RN 01224   0.00 .T.
00003  Carla     Cardoso     R.Bel Monte, 222      Belo Horiz  MG 21105   0.00 .F.
```

As Condições

Chegamos em uma das mais importantes e poderosas partes da sintaxe do dBase, as condições. As condições serão discutidas mais minuciosamente no Capítulo 5. Porém, elas são muito importantes para serem ignoradas aqui. Antes de mais nada, vamos criar um registro para "São Paulo". Use APPEND e digite: Nancy Vieira; R. do Sol, 444; São Paulo; SP; CEP 92111; 1.33.

Agora vamos primeiro demonstrar o uso das condições nos comandos, considerando os seguintes usos de LIST – primeiro o default:

```
. LIST
00001 Mila    Moreira   R.Elma, 111          Sao Paulo  SP 92109   5.00 .T.
00002 Ana     Amorim    R.Tres, 333          Natal      RN 01224   0.00 .T.
00003 Carla   Cardoso   R.Bel Monte, 222     Belo Horiz MG 21105   0.00 .F.
00004 Nancy   Vieira    R.do Sol, 444        Sao Paulo  SP 92111   1.33 .T.
```

Agora com as condições:

```
. LIST FOR CIDADE = 'Sao Paulo'
00001 Mila    Moreira   R.Elma, 111          Sao Paulo  SP 92109   5.00 .T.
00004 Nancy   Vieira    R.do Sol, 444        Sao Paulo  SP 92111   1.33 .T.
```

As condições em dBase consistem em cláusulas "FOR" que dizem ao dBase quais os registros para agir. As condições são independentes dos comandos que elas modificam. Por exemplo, "FOR CIDADE = 'São Paulo' " é uma condição que define um conjunto de registros no arquivo de dados SOCIOS. Em um dado momento havia um grupo de registros para o qual esta condição foi verdadeira.

Sempre que incluímos uma condição em um comando, estamos trocando o escopo do comando do default para o escopo da condição. No comando LIST anterior, trocamos o default de "all" (todos = o default do LIST) para dois registros específicos. Se usássemos a mesma condição para DELETE, aumentaríamos o escopo do default do "registro corrente" para os mesmos dois registros específicos.

```
. LIST
00001 Mila    Moreira   R.Elma, 111          Sao Paulo  SP 92109   5.00 .T.
00002 Ana     Amorim    R.Tres, 333          Natal      RN 01224   0.00 .T.
00003 Carla   Cardoso   R.Bel Monte, 222     Belo Horiz MG 21105   0.00 .F.
00004 Nancy   Vieira    R.do Sol, 444        Sao Paulo  SP 92111   1.33 .T.

. DELETE FOR CIDADE = 'Sao Paulo'
00002 DELETION(S)
```

```
. LIST
00001 *Mila     Moreira    R.Elma, 111         Sao Paulo  SP 92109  5.00 .T.
00002  Ana      Amorim     R.Tres, 333         Natal      RN 01224  0.00 .T.
00003  Carla    Cardoso    R.Bel Monte, 222    Belo Horiz MG 21105  0.00 .F.
00004 *Nancy    Vieira     R.do Sol, 444       Sao Paulo  SP 92111  1.33 .T.
```

QUIT

Os arquivos de dados em dBase são armazenados nos drives de disco gravados magneticamente e permanecem quando o computador é desligado. O trabalho de funcionamento do dBase, porém, acontece na **Random Access Memory*** ou **RAM** e desaparece quando o computador é desligado. É importante levarmos todas as operações da RAM para o disco antes de desligarmos o computador.

Desta maneira, quando um arquivo de dados foi atualizado de qualquer forma pelo APPEND, EDIT, DELETE ou qualquer outro comando que escreve novas informações para o disco, o usuário deve ter certeza de que o dBase fechou o arquivo, enviando todas as mudanças da RAM para o disco. O dBase se oferece para tomar este cuidado automaticamente, com o comando QUIT. ele fecha todos os arquivos, libera todos aos buffers, varre o chão e apaga as luzes.

Obteremos na tela:

```
. QUIT
*** END RUN     dBASE II     ***
```

Devemos manter o hábito de desativarmos o dBase com QUIT (em vez de pressionar o OFF do computador), mesmo quando não alteramos nenhum arquivo.

Isto termina a primeira seção.

SEÇÃO 2 – LOCALIZAÇÃO E ORGANIZAÇÃO DE DADOS

O objetivo da segunda seção é aprender novos comandos para podermos nos movimentar pelo arquivo SOCIOS.DBF (ou qualquer outro arquivo de dados), aprender as maneiras de localizar registros específicos pela posição ou conteúdos e finalmente, como ordenar registros em um arquivo de dados.

* N.T. – Random Access Memory (RAM) – Memória de Acesso Aleatório.

Os comandos que aprenderemos são:

GOTO e #	SORT
SKIP	INDEX
LOCATE	FIND
CONTINUE	REINDEX

GOTO

Até agora, vimos que o dBase associa um número de registro a cada registro em nosso arquivo. Podemos considerar este processo como sendo a "ordem natural" dos registros – é a ordem da entrada e corresponde à organização física dos registros no disco.

Mais tarde, aprenderemos a trocar esta ordem, mas por enquanto, representaria algum progresso se pudéssemos nos movimentar pelos registros de uma forma controlável (em vez de tentar interromper um LIST no registro que desejamos).

Vamos exemplificar. Primeiro, chamamos nosso arquivo, e recuperamos todos os registros.

```
. USE SOCIOS

. RECALL ALL
00002 RECALL(S)
```

Agora usaremos o comando LIST para sabermos onde estamos:

```
. LIST
00001 Mila     Moreira    R.Elma, 111         Sao Paulo  SP 92109   5.00 .T.
00002 Ana      Amorim     R.Tres, 333         Natal      RN 01224   0.00 .T.
00003 Carla    Cardoso    R.Bel Monte, 222    Belo Horiz MG 21105   0.00 .F.
00004 Nancy    Vieira     R.do Sol, 444       Sao Paulo  SP 92111   1.33 .T.
```

Com certeza, depois do LIST alcançamos o último registro:

```
. DISPLAY

00004 Nancy    Vieira     R.do Sol, 444       Sao Paulo  SP 92111   1.33 .T.
```

Agora com o novo comando – GOTO:

```
. GOTO TOP
. DISPLAY
00001 Mila      Moreira     R.Elma, 111        Sao Paulo  SP 92109   5.00 .T.

. GOTO BOTTOM
. DISPLAY
00004 Nancy     Vieira      R.do Sol, 444      Sao Paulo  SP 92111   1.33 .T.
```

O comando GOTO TOP diz ao dBase para colocar o indicador dos registros no primeiro registro do arquivo. Foi tudo que o dBase fez – para olhar aquele registro, tivemos de usar o comando DISPLAY. Da mesma maneira, o GOTO BOTTOM reposicionou o indicador dos registros no último do arquivo. Pelo fato de o dBase reconhecer facilmente as abreviações, o GOTO também pode ser escrito como GO e BOTTOM pode ser reduzido a quatro letras, como todos os comandos dBase, assim GO BOTT é o equivalente a GOTO BOTTOM.

O Indicador de Registro (#)

Quando estamos usando um arquivo de dados, a qualquer momento podemos pedir para o dBase nos fornecer o número do registro corrente. O comando geral para a inquisição do dBase a partir do prompt é um simples ponto de interrogação. A abreviação que o dBase reconhece para o **indicador de registro** é o sinal (#).

```
. GOTO TOP    (*Não percebemos reação, mas o indicador moveu-se*)
. ? #         (*Então perguntamos onde está o indicador*)
    1         (*e o dBase responde com o número do registro*)
```

GOTO

Como as duas localizações logicamente posicionadas – TOP e BOTTOM (explicadas na próxima seção), GOTO pode ser usado para posicionar o indicador do registro em um número de registro específico.

```
. GOTO 2
. ? #
    2

. DISPLAY
00002 Ana       Amorim      R.Tres, 333        Natal      RN 01224   0.00 .T.
```

A Posição Lógica *Versus* A Posição Real

Durante o resto desta operação, estaremos nos movimentando bastante pelo arquivo SOCIOS. Seria bom observarmos que enquanto fazemos isto estamos utilizando os vários termos de posicionamento.

Todos os registros em um arquivo de dados possuem uma posição real no disco que correspondem às suas ordens de entrada. A ordem de entrada corresponde aos números do registro que o dBase associa a cada registro (o número que obtemos quando perguntamos usando o sinal numérico "#"). Lembrem-se de que este número pertence ao dBase e não faz parte constante do registro durante o uso do comando CREATE.

Consideremos como exemplo um banco de dados com estas letras maiúsculas:

# do dBase	Nosso campo LETRA
00001	C
00002	*A
00003	X
00004	G
00005	B

Os topos e bases real e lógico do arquivo de dados correspondem-se. Porém, como veremos mais tarde, se fôssemos colocar o arquivo em ordem alfabética, o topo e a base mudariam embora os números dos registros permanecessem os mesmos:

# do dBase	Nosso campo LETRA
00002	* A (*novo TOPO*)
00005	B
00001	C
00004	G
00003	X (*nova BASE*)

Além disto, não devemos nos tornar muito dependentes dos números dos registros quando programamos, porque o dBase os mudará se precisar. Em nosso arquivo original, por exemplo, um comando PACK renumeraria todos os registros abaixo após apagar o arquivo 2. Antes de usarmos PACK:

# do dBase	Nosso campo LETRA
00001	C
00002	*A
00003	X
00004	G
00005	B

Depois de usarmos PACK:

# do dBase	Nosso campo LETRA
00001	C
00002	X
00003	G
00004	B

Agora vamos fazer a distinção entre o TOPO e a BASE como locais lógicos e o NEXT como um escopo lógico (como em "LIST NEXT 5"). Comparando os números dos registros podem ser considerados absolutos, tendo em mente que o dBase os fornece e os retira. Se desejamos obter um número específico e inalterável e associado com um registro, é nossa responsabilidade colocá-lo como um campo que faça parte da estrutura do registro durante o CREATE.

SKIP

O comando SKIP é responsável pela função de movimentar o indicador do registro dentro de um arquivo de dados em uma ordem lógica. Podemos visualizar este processo em um exemplo:

```
. GOTO TOP
. DISPLAY
00001  Mila      Moreira       R.Elma, 111          Sao Paulo  SP 92109   5.00 .T.

. SKIP
RECORD: 00002
. DISPLAY
00002  Ana       Amorim        R.Tres, 333          Natal      RN 01224   0.00 .T.
```

O comando SKIP atua no próximo registro lógico – ele "salta" para o próximo – mas pode ser usado para saltar mais de um registro de cada vez e caminhar para as duas direções simplesmente usando "SKIP n" ou "SKIP-n".

```
. SKIP 2
RECORD: 00004
. DISPLAY
00004  Nancy     Vieira        R.do Sol, 444        Sao Paulo  SP 92111   1.33 .T.
. SKIP -1
RECORD: 00003
. DISPLAY
00003  Carla     Cardoso       R.Bel Monte, 222     Belo Horiz MG 21105   0.00 .F.
```

LOCATE e CONTINUE

Às vezes queremos localizar um registro pelo seu conteúdo em vez da sua posição lógica ou real no arquivo. Um comando muito útil para este objetivo é o LOCATE, o qual por razões óbvias sempre necessita de uma declaração de condição "FOR".

```
. GOTO TOP
. LOCATE FOR SOBRENOME = "Cardoso"
RECORD: 00003


. DISPLAY
00003  Carla      Cardoso         R.Bel Monte, 222     Belo Horiz MG 21105   0.00 .F.
```

O comando LOCATE inicia a pesquisa no topo do arquivo e continua até que encontre o registro que satisfaça a condição. Se o registro não for o desejado, a pesquisa poderá ser reiniciada a partir do ponto onde parou, usando o comando CONTINUE:

```
. LOCATE FOR CONTRIB = 0
RECORD: 00002
. DISPLAY
00002  Ana        Amorim          R.Tres, 333          Natal      RN 01224   0.00 .T.
. CONTINUE
RECORD: 00003
. DISPLAY
00003  Carla      Cardoso         R.Bel Monte, 222     Belo Horiz MG 21105   0.00 .F.
. CONTINUE
END OF FILE ENCOUNTERED
. DISPLAY

00004  Nancy      Vieira          R.do Sol, 444        Sao Paulo  SP 92111   1.33 .T.
```

Perceba o que acontece quando o LOCATE (ou um LOCATE reativado com CONTINUE) chega ao final do arquivo sem encontrar um registro que satisfaça sua condição. O dBase emite uma mensagem que encontrou o fim do arquivo e abandona o indicador dos registros na base lógica do arquivo, que pode ou não ser um registro que satisfaça a condição LOCATE.

Os programadores que usam LOCATE em seus programas devem lembrar-se de testar os registros que encontram para terem certeza se eles satisfazem a condição especificada ou se o dBase simplesmente não terminou no fim do arquivo. Felizmente este é um teste simples, como veremos mais tarde.

SORT

O comando SORT é usado em dBase para colocar ordem fixa nos registros em um arquivo de dados. Quando fazemos uma classificação, o dBase escreve um novo arquivo .DBF do mesmo tamanho que o arquivo original, com todos os registros iguais, mas com uma nova ordem.

Para fazermos uma classificação, o dBase necessita saber duas coisas – o nome do novo arquivo ordenado e o "campo-chave" para classificar. O dBase abrirá o arquivo classificado sob o novo nome (deste modo apagando qualquer arquivo antigo com aquele nome) e escreverá todos os registros na nova ordem (os registros marcados pelo comando DELETE não serão gravados).

A ordem no novo arquivo é determinada pela seqüência de intercalação ASCII do campo-chave na classificação. A classificação em dBase só pode ser feita em um campo, tal como código postal ou sobrenome. Seu objetivo específico é criar um novo arquivo com uma nova ordem. Usando o arquivo SOCIOS.DBF, vamos observar o exemplo:

```
. LIST         (*observaremos primeiro a ordem inicial*)
00001 Mila     Moreira    R.Elma, 111          Sao Paulo  SP 92109  5.00 .T.
00002 Ana      Amorim     R.Tres, 333          Natal      RN 01224  0.00 .T.
00003 Carla    Cardoso    R.Bel Monte, 222     Belo Horiz MG 21105  0.00 .F.
00004 Nancy    Vieira     R.do Sol, 444        Sao Paulo  SP 92111  1.33 .T.


. SORT ON SOBRENOME TO TEMPO      (*O dBase abrirá um arquivo TEMPO.DBF*)
SORT COMPLETE
. LIST         (*Pergunta: Qual arquivo estamos listando aqui?*)
00001 Mila     Moreira    R.Elma, 111          Sao Paulo  SP 92109  5.00 .T.
00002 Ana      Amorim     R.Tres, 333          Natal      RN 01224  0.00 .T.
00003 Carla    Cardoso    R.Bel Monte, 222     Belo Horiz MG 21105  0.00 .F.
00004 Nancy    Vieira     R.do Sol, 444        Sao Paulo  SP 92111  1.33 .T.

  (*Resposta: Ainda temos SOCIOS em uso!*)
. USE TEMPO
. LIST
00001 Ana      Amorim     R.Tres, 333          Natal      RN 01224  0.00 .T.
00002 Carla    Cardoso    R.Bel Monte, 222     Belo Horiz MG 21105  0.00 .F.
00003 Mila     Moreira    R.Elma, 111          Sao Paulo  SP 92109  5.00 .T.
00004 Nancy    Vieira     R.do Sol, 444        Sao Paulo  SP 92111  1.33 .T.
```

Note a característica da operação SORT. A integridade de cada registro foi respeitada (Ana ainda é de Natal), o sobrenome Vieira ainda é de Nancy, mas os registros foram colocados em ordem e o dBase designou novos números de registro para cada um. Depois de uma classificação, a ordem aparente e lógica em um arquivo são as mesmas com a referência para o campo-chave.

Vamos voltar ao arquivo SOCIOS (lembre-se, SOCIOS e TEMPO são arquivos diferentes) e fazer uma outra classificação usando o comando SORT:

```
. USE SOCIOS
. SORT ON CONTRIB TO TEMPO
SORT COMPLETE
. USE TEMPO
. LIST
00001  Ana     Amorim     R.Tres, 333       Natal      RN 01224  0.00 .T.
00002  Carla   Cardoso    R.Bel Monte, 222  Belo Horiz MG 21105  0.00 .F.
00003  Nancy   Vieira     R.do Sol, 444     Sao Paulo  SP 92111  1.33 .T.
00004  Mila    Moreira    R.Elma, 111       Sao Paulo  SP 92109  5.00 .T.
```

Já que usamos o mesmo nome que antes, o TEMPO antigo desapareceu e este novo TEMPO está em seu lugar. Não é uma prática ruim usar um único nome de arquivo (tal como "TEMPO") para os nossos arquivos transitórios, já que cada novo arquivo temporário sobregravará o anterior. Uma vez que cada TEMPO apaga o último, o usuário (ou o programa) não tem de apagar arquivos temporários já obsoletos.

Nas duas classificações, CONTRIB e SOBRENOME, os campos-chaves do registro ficaram com valores crescentes. Algumas vezes queremos os maiores valores no topo do arquivo, assim SORT pode aceitar um argumento a mais: "DESCENDING".

```
. SORT ON CONTRIB DESCENDING TO TEMPO2
SORT COMPLETE
. USE TEMPO2
. LIST
00001  Mila    Moreira    R.Elma, 111       Sao Paulo  SP 92109  5.00 .T.
00002  Nancy   Vieira     R.do Sol, 444     Sao Paulo  SP 92111  1.33 .T.
00003  Ana     Amorim     R.Tres, 333       Natal      RN 01224  0.00 .T.
00004  Carla   Cardoso    R.Bel Monte, 222  Belo Horiz MG 21105  0.00 .F.
```

A classificação em dBase possui uma má reputação comparada com outros programas gerenciadores de banco de dados e talvez haja justiça nesta colocação. O dBase permite somente um campo-chave por classificação, o que limita a ação do comando para uma categoria (por exemplo, não poderíamos classificar nosso arquivo por ESTADO e em seguida por CIDADE dentro de cada ESTADO). Além disso, as classificações do dBase são relativamente lentas em comparação a muitos outros programas de classificação.

Em defesa ao dBase, vamos colocar que as classificações, por mais enjoadas que sejam, não representam um problema maior do que os que têm alguns concorrentes da Ashton-Tate. A razão para isto é simples. Na maioria dos casos, os programadores em dBase não usam o comando SORT, porque o INDEX é muito mais rápido, poderoso e geralmente melhor. No Capítulo 9, aprenderemos a usar o comando INDEX para fazermos tudo o que poderíamos realizar com SORT.

INDEX

A sintaxe do comando INDEX é, para todos os objetivos e intenções, idêntica ao SORT. O INDEX também necessita de uma expressão-chave e o nome de um arquivo de saída. (DESCENDING não pode ser usado como um argumento com INDEX.)

Porém, o comando INDEX possui uma função completamente diferente do que SORT. Onde o SORT escreve um novo arquivo contendo os mesmos registros em uma ordem classificada, o INDEX escreve um arquivo menor e específico, com o default do tipo de arquivo ".NDX.". Além disso, os arquivos de índice do dBase são auto-organizáveis – eles mantêm a ordem dos registros em um arquivo mesmo quando inserimos novos registros.

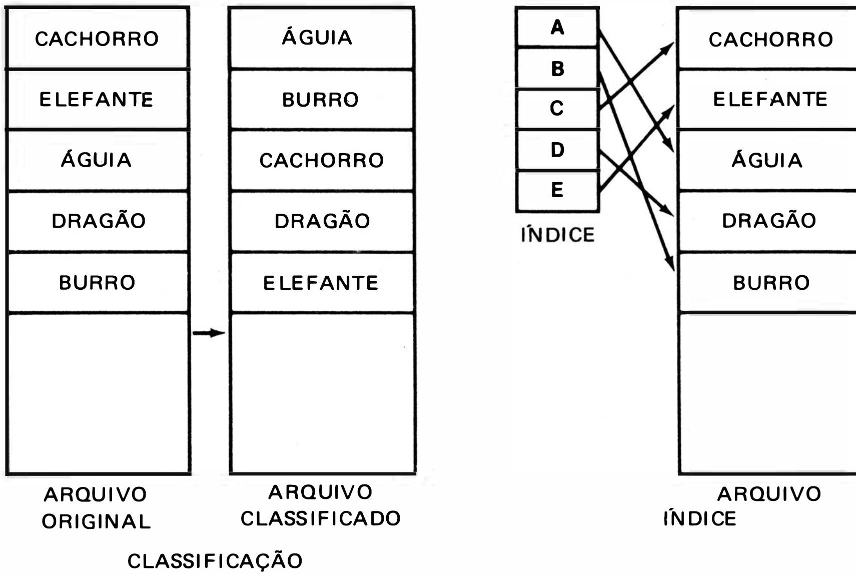


Figura 3-1 Classificação *versus* indexação.

Um exemplo de indexação usando nosso arquivo de dados SOCIOS ajudará a explicar.

```
. USE SOCIOS
. LIST
00001 Mila     Moreira    R.Elma, 111          Sao Paulo  SP 92109   5.00 .T.
00002 Ana      Amorim     R.Tres, 333          Natal      RN 01224   0.00 .T.
00003 Carla    Cardoso    R.Bel Monte, 222     Belo Horiz MG 21105   0.00 .F.
00004 Nancy    Vieira     R.do Sol, 444        Sao Paulo  SP 92111   1.33 .T.
```

```
. INDEX ON CIDADE TO SOCIO
00004 RECORDS INDEXED

. LIST
00003 Carla    Cardoso    R.Bel Monte, 222    Belo Horiz MG 21105   0.00 .F.
00002 Ana      Amorim     R.Tres, 333         Natal      RN 01224   0.00 .T.
00001 Mila     Moreira    R.Elma, 111         Sao Paulo  SP 92109   5.00 .T.
00004 Nancy    Vieira     R.do Sol, 444       Sao Paulo  SP 92111   1.33 .T.
```

Note que o número dos registros não foram mudados, mas o arquivo aparece em nova ordem – alfabética por cidade. É igual ao nosso exemplo do bibliotecário super-rápido do Capítulo 1, que podia retirar cada livro das prateleiras rapidamente quando apontávamos no catálogo.

Uma vez formado o índice, o dBase coloca-o em uso – isto é, os registros são apresentados em sua ordem lógica nova, sempre que forem chamados. Depois, para o comando LIST, o dBase procura o primeiro registro para mostrar no arquivo de índice, em vez do arquivo de dados. O primeiro registro neste caso é o número 3, Carla Cardoso de Belo Horizonte. O registro 3 tornou-se o "TOP" lógico do arquivo de dados e será mostrado, a partir de agora, quando usarmos o comando "GOTO TOP", assumindo que este índice está em uso. Da mesma maneira, se usarmos "SKIP" no registro 2 (Ana de Natal) o dBase colocará o indicador no próximo registro lógico, que é Mila de São Paulo.

A indexação em dBase é muito poderosa, além de ordenar o arquivo de dados. Consideremos um caso em que colocamos um novo registro em um pequeno arquivo em ordem alfabética que acabou de ser classificado.

Antes do APPEND:

```
00001 A
00002 B
00003 C
00004 F
```

Depois do APPEND:

```
00001 A
00002 B
00003 E
00004 F
00005 C        (*Que mancada! precisamos classificar novamente*)
```

A indexação em dBase é projetada especialmente para evitar este problema com arquivos classificados. Sempre que um arquivo de índice esteja sendo usado, todos os acréscimos ao arquivo são colocados em ordem, automaticamente, no índice, no momento da entrada. Lembre-se de que aqui nosso índice está no campo CIDADE.

```
RECORD # 00005
NOME      :Pedro     :
SOBRENOME :Penedo         :
ENDERECO  :R.Sao Luiz, 22       :
CIDADE    :Curitiba  :
ESTADO    :PR:
CEP       :92109:
CONTRIB   : 0.00:
PAGAMENTO :n:

RECORD # 00006                     (*Este é o registro 6, mas ele aparecerá*)
NOME      :Zilda     :             (*no topo lógico do arquivo*)
SOBRENOME :Zorzi          :
ENDERECO  :R.Pilar, 44          :
CIDADE    :Aracaju   :
ESTADO    :SE:
CEP       :     :
CONTRIB   : 8.77:
PAGAMENTO :T:

. LIST
00006  Zilda   Zorzi     R.Pilar, 44         Aracaju    SE        8.77 .T.
00003  Carla   Cardoso   R.Bel Monte, 222    Belo Horiz MG 21105  0.00 .F.
00005  Pedro   Penedo    R.Sao Luiz, 22      Curitiba   PR 92109  0.00 .F.
00002  Ana     Amorim    R.Tres, 333         Natal      RN 01224  0.00 .T.
00001  Mila    Moreira   R.Elma, 111         Sao Paulo  SP 92109  5.00 .T.
00004  Nancy   Vieira    R.do Sol, 444       Sao Paulo  SP 92111  1.33 .T.
```

Se o usuário não gostou desta ordem indexada, então o arquivo pode ser visto em sua ordem de entrada. Ele pode simplesmente ser USADO novamente.

```
. USE SOCIOS
. LIST
00001  Mila    Moreira   R.Elma, 111         Sao Paulo  SP 92109  5.00 .T.
00002  Ana     Amorim    R.Tres, 333         Natal      RN 01224  0.00 .T.
00003  Carla   Cardoso   R.Bel Monte, 222    Belo Horiz MG 21105  0.00 .F.
00004  Nancy   Vieira    R.do Sol, 444       Sao Paulo  SP 92111  1.33 .T.
00005  Pedro   Penedo    R.Sao Luiz, 22      Curitiba   PR 92109  0.00 .F.
00006  Zilda   Zorzi     R.Pilar, 44         Aracaju    SE        8.77 .T.
```

Se um arquivo de índice já existe, então ele pode ser aberto e colocado em uso no mesmo comando que abre o arquivo de dados.

```
. USE SOCIOS INDEX SOCID
. LIST
00006 Zilda   Zorzi     R.Pilar, 44        Aracaju    SE        8.77 .T.
00003 Carla   Cardoso   R.Bel Monte, 222   Belo Horiz MG 21105  0.00 .F.
00005 Pedro   Penedo    R.Sao Luiz, 22     Curitiba   PR 92109  0.00 .F.
00002 Ana     Amorim    R.Tres, 333        Natal      RN 01224  0.00 .T.
00001 Mila    Moreira   R.Elma, 111        Sao Paulo  SP 92109  5.00 .T.
00004 Nancy   Vieira    R.do Sol, 444      Sao Paulo  SP 92111  1.33 .T.
```

Se um arquivo de dados já está em uso, um índice preexistente pode ser colocado em uso pelo comando "SET INDEX TO <nome do índice>", com o indicador do registro caminhando automaticamente para o topo do arquivo, como faria para a declaração equivalente, "USE <arquivo> INDEX <nome do índice>".

Até sete índices podem ser abertos em um único arquivo de dados de uma só vez, embora (obviamente) o arquivo possa estar em apenas uma ordem. O primeiro índice nomeado é chamado **índice primário** e determina a ordem aparente do arquivo.

Índices múltiplos são separados por vírgulas quando abertos, com a sintaxe do comando:

```
.USE <nome do arquivo> INDEX <índice1>,<índice2>,<índice3>...
                        ou
.USE <nome do arquivo>
.SET INDEX TO <índice1>,<índice2>,<índice3>...
```

Embora possamos abrir sete índices de uma só vez, cada um deles necessitará ser reorganizado quando usarmos um APPEND ou EDIT para um registro; quando existirem mais de dois ou três índices abertos, o dBase ficará lerdo, especialmente quando o computador possui drive de discos flexíveis em vez de discos rígidos.

FIND

O comando FIND ilustra outra característica brilhante da indexação que a distingue da classificação. Até agora, o LOCATE foi o único comando que seleciona um registro de um arquivo de dados pelo seu conteúdo. Notamos que o comando LOCATE possui a desvantagem de ser relativamente lento quando movimenta-se entre os registros.

Uma vez que o arquivo foi indexado em uma chave, porém, o dBase possui uma maneira rápida de colocar o indicador em um registro que satisfaça um critério – desde que o critério usado na pesquisa seja algo armazenado no índice.

O comando é o FIND e apresentaremos a regra para usá-lo: se um arquivo de dados é indexado em uma *série* de caracteres-chave e se o índice está em uso, então com um FIND, o dBase colocará o indicador no primeiro registro cujo dado seja coincidente à chave pesquisada. Se nenhum registro possuir a chave de pesquisa, então o indicador de registro retornará com zero.

Vamos exemplificar. Temos em uso SOCIOS INDEX SOCID:

```
. GOTO TOP          (*Na verdade, FIND funciona a partir de qualquer lugar*)
. FIND Natal
. DISPLAY
00002  Ana      Amorim      R.Tres, 333        Natal      RN 01224   0.00 .T.
. FIND Curitiba
. DISPLAY
00005  Pedro    Penedo      R.Sao Luiz, 22     Curitiba   PR 92109   0.00 .F.
. FIND Mila         (*Esta é uma pergunta estratégica*)
NO FIND             (*Claro que não! "Mila" não é uma cidade.*)
. FIND SAO PAULO    (*Veja o problema aqui*)
NO FIND             (*Certo, "SAO PAULO" é diferente de "Sao Paulo"*)
. FIND Sao Paulo
. DISPLAY
00001  Mila     Moreira     R.Elma, 111        Sao Paulo  SP 92109   5.00 .T.
```

Como o comando INDEX, o FIND é um dos mais fundamentais e importantes em dBase e recomendamos experimentá-lo até que seja entendido completamente. Mantenha em mente o seguinte: a pergunta FIND diz ao dBase para pesquisar no índice primário em uso e encaixar-se na série informada. O dBase fará exatamente isto quando solicitado, porém o usuário ou o programador terão de ser responsáveis pela certeza de que a pergunta FIND tem sentido.

Se indexarmos por cidades e então tentamos encontrar (FIND) em nome como fizemos com "Mila", o dBase dirá que não pode encontrar Mila, mas não poderá nos dizer que perguntamos pela chave errada. Da mesma forma se pedirmos que encontre "Natal" e nos esquecermos de colocar o índice da cidade em uso, obteremos um NO FIND.

Outra característica de FIND a ser lembrada, é que o dBase colocará o indicador no primeiro registro que encaixe a chave de pesquisa. Se vários registros possuem a mesma chave no campo indexado, o indicador indicará o primeiro, e se abreviarmos a chave no FIND, ele indicará o primeiro registro que satisfaça a abreviação:

```
. FIND Sao          (*São Bernardo? Caetano? *)
. DISPLAY
00001  Mila     Moreira     R.Elma, 111        Sao Paulo  SP 92109   5.00 .T.
. FIND N
. DISPLAY
00002  Ana      Amorim      R.Tres, 333        Natal      RN 01224   0.00 .T.
```

É lógico que muitas vezes esta característica não é desejável e para ocasiões que a exatidão seja necessária o dBase nos permite ativar uma variável de sistema chamada "EXACT", que mudará as regras de uma pesquisa. Com o EXACT ativado, o campo-chave no FIND deve ser exatamente igual ao conteúdo do índice:

```
. SET EXACT ON
. FIND N              (*Chegamos ao "Natal" da última vez*)
NO FIND

. FIND Sao Pa
NO FIND

. FIND Sao Paulo
. DISPLAY
00001  Mila      Moreira        R.Elma, 111         Sao Paulo  SP 92109   5.00 .T.
```

Quando desejamos FIND de volta para sua regra original, desativamos o comando EXACT:

```
. SET EXACT OFF
. FIND C
. DISPLAY
00005  Pedro     Penedo         R.Sao Luiz, 22      Curitiba   PR 92109   0.00 .F.
```

REINDEX (Versão 2.4)

Consideremos a seguinte seqüência de eventos. Em uma segunda-feira, criamos um arquivo de dados com alguns registros. Indexamos em um campo específico e no fechamento da operação, deixamos o dBase com QUIT.

Na terça-feira, colocamos o arquivo de dados em uso mas esquecemos de colocar o índice em uso com ele. Um registro é acrescentado e então novamente usamos QUIT. A quarta-feira chega. Desta vez, quando usamos o arquivo, colocamos o índice em uso também e acrescentamos novos dados. Essa situação ocorre freqüentemente na prática. Possuímos um arquivo de dados em uso com um índice aberto, mas um ou mais registros no arquivo não possuem correspondentes no índice. Isto pode ocasionar efeitos estranhos – os registros podem não aparecer em uma listagem ou outros registros podem aparecer duas vezes.

Ocasionalmente o dBase fornecerá a mensagem de erro RECORD OUT OF RANGE explicando que não consegue alcançar um registro que pode estar lá. Todos estes são sintomas de que o índice está incorreto para o arquivo de dados real.

Vamos misturar um índice de propósito para demonstrar:

```
. LIST
00006 Zilda     Zorzi     R.Pilar, 44          Aracaju    SE        8.77 .T.
00003 Carla     Cardoso   R.Bel Monte, 222     Belo Horiz MG 21105  0.00 .F.
00005 Pedro     Penedo    R.Sao Luiz, 22       Curitiba   PR 92109  0.00 .F.
00002 Ana       Amorim    R.Tres, 333          Natal      RN 01224  0.00 .T.
00001 Mila      Moreira   R.Elma, 111          Sao Paulo  SP 92109  5.00 .T.
00004 Nancy     Vieira    R.do Sol, 444        Sao Paulo  SP 92111  1.33 .T.


. INDEX ON SOBRENOME TO SOBREN
00006 RECORDS INDEXED
. LIST
00002 Ana       Amorim    R.Tres, 333          Natal      RN 01224  0.00 .T.
00003 Carla     Cardoso   R.Bel Monte, 222     Belo Horiz MG 21105  0.00 .F.
00001 Mila      Moreira   R.Elma, 111          Sao Paulo  SP 92109  5.00 .T.
00005 Pedro     Penedo    R.Sao Luiz, 22       Curitiba   PR 92109  0.00 .F.
00004 Nancy     Vieira    R.do Sol, 444        Sao Paulo  SP 92111  1.33 .T.
00006 Zilda     Zorzi     R.Pilar, 44          Aracaju    SE        8.77 .T.

(*Este registro não será colocado em SOCID.NDX*)

. APPEND
RECORD # 00007
NOME      :Carlos    :
SOBRENOME :Covas          :
ENDERECO  :R. "A", 88          :
CIDADE    :Sao Paulo :
ESTADO    :SP:
CEP       :92101:
CONTRIB   : 0.25:
PAGAMENTO :Y:
```

```
. LIST       (*Nosso registro está em SOBRENOME, então o registro 7 está Ok*)
00002 Ana       Amorim    R.Tres, 333          Natal      RN 01224  0.00 .T.
00003 Carla     Cardoso   R.Bel Monte, 222     Belo Horiz MG 21105  0.00 .F.
00007 Carlos    Covas     R. "A", 88           Sao Paulo  SP 92101  0.25 .T.
00001 Mila      Moreira   R.Elma, 111          Sao Paulo  SP 92109  5.00 .T.
00005 Pedro     Penedo    R.Sao Luiz, 22       Curitiba   PR 92109  0.00 .F.
00004 Nancy     Vieira    R.do Sol, 444        Sao Paulo  SP 92111  1.33 .T.
00006 Zilda     Zorzi     R.Pilar, 44          Aracaju    SE        8.77 .T.
```

```
. USE SOCIOS
. SET INDEX TO SOCID        (*Este índice não possui o registro 7*)
. LIST
00006 Zilda   Zorzi    R.Pilar, 44         Aracaju    SE        8.77 .T.
00003 Carla   Cardoso  R.Bel Monte, 222    Belo Horiz MG 21105  0.00 .F.
00005 Pedro   Penedo   R.Sao Luiz, 22      Curitiba   PR 92109  0.00 .F.
00002 Ana     Amorim   R.Tres, 333         Natal      RN 01224  0.00 .T.
00001 Mila    Moreira  R.Elma, 111         Sao Paulo  SP 92109  5.00 .T.
00004 Nancy   Vieira   R.do Sol, 444       Sao Paulo  SP 92111  1.33 .T.

. SKIP           (*Podemos forçar o indicador com SKIP?*)
RECORD: 00004        (*Não! O índice precisa primeiro ser ajustado*)
```

Para corrigirmos esta situação, o arquivo de índice deve ser reconstruído, assim os conteúdos verdadeiros serão considerados.

Na versão 2.3 do dBase e nas anteriores, a responsabilidade de recriar os índices um por um, com os comandos apropriados, ficava com o usuário ou o programador:

```
.INDEX ON <expressão-chave> TO <nome do indice>
```

Com a versão 2.4 do dBase e provavelmente para todas as próximas, o comando "REINDEX" conserta automaticamente todos os índices em uso de acordo com as regras do índice original.

Além disso, o REINDEX é automático depois do comando PACK se houver algum arquivo de índice em uso.

```
. REINDEX

REINDEXING INDEX FILE - B:SOCID   .NDX
00007 RECORDS INDEXED
. LIST
00006 Zilda   Zorzi    R.Pilar, 44         Aracaju    SE        8.77 .T.
00003 Carla   Cardoso  R.Bel Monte, 222    Belo Horiz MG 21105  0.00 .F.
00005 Pedro   Penedo   R.Sao Luiz, 22      Curitiba   PR 92109  0.00 .F.
00002 Ana     Amorim   R.Tres, 333         Natal      RN 01224  0.00 .T.
00001 Mila    Moreira  R.Elma, 111         Sao Paulo  SP 92109  5.00 .T.
00004 Nancy   Vieira   R.do Sol, 444       Sao Paulo  SP 92111  1.33 .T.
00007 Carlos  Covas    R. "A", 88          Sao Paulo  SP 92101  0.25 .T.
```

Perceba que o índice do nosso sobrenome não foi afetado pela atualização do índice da cidade. Lembre-se de que todos os índices são arquivos separados.

Vamos mostrar que os dois índices foram atualizados colocando de volta SOBREN como índice primário e SOCID como índice secundário. Usaremos APPEND para o novo registro e então ativaremos cada um para verificarmos se os dois foram atualizados:

```
. SET INDEX TO SOBREN,SOCID
. LIST
00002  Ana      Amorim    R.Tres, 333          Natal      RN 01224  0.00 .T.
00003  Carla    Cardoso   R.Bel Monte, 222     Belo Horiz MG 21105  0.00 .F.
00007  Carlos   Covas     R. "A", 88           Sao Paulo  SP 92101  0.25 .T.
00001  Mila     Moreira   R.Elma, 111          Sao Paulo  SP 92109  5.00 .T.
00005  Pedro    Penedo    R.Sao Luiz, 22       Curitiba   PR 92109  0.00 .F.
00004  Nancy    Vieira    R.do Sol, 444        Sao Paulo  SP 92111  1.33 .T.
00006  Zilda    Zorzi     R.Pilar, 44          Aracaju    SE        8.77 .T.

RECORD # 00008
NOME       :Davi      :
SOBRENOME  :Dunas          :
ENDERECO   :Rua das Flores, 123 :
CIDADE     :Aracaju   :
ESTADO     :SE:
CEP        :92111:
CONTRIB    : 1.33:
PAGAMENTO  :t:
```

```
. LIST
00002  Ana      Amorim    R.Tres, 333          Natal      RN 01224  0.00 .T.
00003  Carla    Cardoso   R.Bel Monte, 222     Belo Horiz MG 21105  0.00 .F.
00007  Carlos   Covas     R. "A", 88           Sao Paulo  SP 92101  0.25 .T.
00008  Davi     Dunas     Rua das Flores, 123  Aracaju    SE 92111  1.33 .T.
00001  Mila     Moreira   R.Elma, 111          Sao Paulo  SP 92109  5.00 .T.
00005  Pedro    Penedo    R.Sao Luiz, 22       Curitiba   PR 92109  0.00 .F.
00004  Nancy    Vieira    R.do Sol, 444        Sao Paulo  SP 92111  1.33 .T.
00006  Zilda    Zorzi     R.Pilar, 44          Aracaju    SE        8.77 .T.
```

(*Ok, e quando trocamos os índices? *)

```
. SET INDEX TO SOCID,SOBREN
. LIST
00006  Zilda    Zorzi     R.Pilar, 44          Aracaju    SE        8.77 .T.
00008  Davi     Dunas     Rua das Flores, 123  Aracaju    SE 92111  1.33 .T.
00003  Carla    Cardoso   R.Bel Monte, 222     Belo Horiz MG 21105  0.00 .F.
```

```
00005 Pedro    Penedo     R.Sao Luiz, 22    Curitiba   PR 92109  0.00 .F.
00002 Ana      Amorim     R.Tres, 333       Natal      RN 01224  0.00 .T.
00001 Mila     Moreira    R.Elma, 111       Sao Paulo  SP 92109  5.00 .T.
00004 Nancy    Vieira     R.do Sol, 444     Sao Paulo  SP 92111  1.33 .T.
00007 Carlos   Covas      R. "A", 88        Sao Paulo  SP 92101  0.25 .T.

(*Continua Ok*)
```

Agora o registro oito aparece nos índices cidade e sobrenome e aparecerá em sua própria posição sempre que algum dos dois esteja em uso.

Mais Sobre Indexação

Há algumas limitações nos comandos INDEX e FIND em dBase. Em primeiro lugar, as indexações não podem ser feitas em campos lógicos. Se refletirmos sobre isto perceberemos que é lógico. Tal indexação, se pudermos fazê-la, separaria somente os registros em dois grupos.

A segunda restrição é que não podemos fazer nada exceto um FIND exato em um campo numérico. Em outras palavras,

```
.FIND Sao
```

encontraria São Paulo, São Bernardo e São Caetano; mas

```
.FIND 1.000
```

pode somente encontrar 1.000 – não encontra o primeiro 1.<alguma coisa>.

Apesar de não examinarmos as funções e expressões funcionais em dBase até o Capítulo 5, é necessário observar que até agora falamos superficialmente sobre a indexação. O dBase permite indexar (e classificar) em expressões complexas que podem ser construídas de bits ou pedaços de campos de dados retirados dos registros.

Na verdade, a indexação criativa é um aspecto em potencial da programação dBase. Vamos observar dois exemplos simples. O primeiro é uma indexação que ordenará nossos registros em SOCIOS pelo sobrenome *e* nome alfabeticamente:

```
. INDEX ON (SOBRENOME + NOME) TO NOVONOME <cr>
```

Já que não obtivemos nenhum efeito diferente em nosso arquivo de dados SOCIOS, pularemos o exemplo da listagem.

O segundo exemplo é um índice que coloca os sócios em ordem decrescente de contribuições:

```
. INDEX ON (-CONTRIB) TO CONTR
00008 RECORDS INDEXED

. LIST NOME,SOBRENOME,CONTRIB
00006  Zilda     Zorzi       8.77
00001  Mila      Moreira     5.00
00004  Nancy     Vieira      1.33
00008  Davi      Dunas       1.33
00007  Carlos    Covas       0.25
00002  Ana       Amorim      0.00
00003  Carla     Cardoso     0.00
00005  Pedro     Penedo      0.00
```

Nos dois exemplos, a expressão em parênteses é a regra pela qual o novo índice será escrito e todos os APPENDs subseqüentes serão colocados no índice de acordo com sua regra.

A habilidade de criar índices facilmente e usá-los para pesquisas rápidas é a razão primordial que tornou o programa dBase o sistema gerenciador de banco de dados dominante no mundo dos microcomputadores. Certifique-se de que entendeu os comandos INDEX, FIND e REINDEX antes de continuar com a Seção 3.

SEÇÃO 3 – ALTERAÇÃO E MANIPULAÇÃO DE DADOS

Na terceira seção, nos concentraremos na modificação de dados dentro de um arquivo e na interface entre os nossos arquivos e outros softwares. Os comandos que usaremos serão:

EDIT	APPEND BLANK
REPLACE	SDF e DELIMITED
COPY TO	MODIFY STRUCTURE
APPEND FROM	

Para evitarmos listagens desnecessárias durante a Seção 3, apresentamos abaixo uma listagem do arquivo de dados que será o assunto de todos os exercícios:

```
. USE SOCIOS
. LIST
00001  Mila     Moreira    R.Elma, 111         Sao Paulo  SP 92109   5.00 .T.
00002  Ana      Amorim     R.Tres, 333         Natal      RN 01224   0.00 .T.
00003  Carla    Cardoso    R.Bel Monte, 222    Belo Horiz MG 21105   0.00 .F.
```

```
00004 Nancy    Vieira    R.do Sol, 444         Sao Paulo  SP 92111   1.33 .T.
00005 Pedro    Penedo    R.Sao Luiz, 22        Curitiba   PR 92109   0.00 .F.
00006 Zilda    Zorzi     R.Pilar, 44           Aracaju    SE         8.77 .T.
00007 Carlos   Covas     R. "A", 88            Sao Paulo  SP 92101   0.25 .T.
00008 Davi     Dunas     Rua das Flores, 123   Aracaju    SE 92111   1.33 .T.
```

EDIT – Alteração de um Registro

Freqüentemente, no modo interativo – isto é, a partir do prompt – existem ocasiões para alterar os conteúdos de um campo em um registro específico. O comando EDIT do dBase é um instrumento versátil para chamarmos um registro, modificá-lo e depois movê-lo para outro registro ou voltar ao prompt.

A sintaxe do EDIT inclui:

1. EDIT sem argumentos

```
.EDIT     (*Sem argumentos — o dBase*)
          (*pede o número do registro*)
.EDIT n   (*Onde n = <um número de registro> *)
.EDIT #   (*Edita o registro corrente " #" *)
```

Durante um EDIT não formatado (isto é, um no qual o programador não define o formato de tela para o dBase), o registro que está sendo editado é trazido com a mesma tela que o APPEND usa. O comando EDIT usa os mesmos controles do cursor de tela-cheia descritos no Capítulo 2 para gravar, sair e movimentar o cursor pelo registro.

Os ^C e ^R possuem significados específicos durante um EDIT. Estes dois caracteres de controle podem ser usados durante a edição para movimentar-se rapidamente entre os registros em ordem lógica. Vamos dizer que estamos editando o registro número 4 em um arquivo não indexado. Quando tivermos a certeza de que o registro está correto, gravaremos normalmente digitando ^W.

Porém, se quiséssemos gravá-lo e prosseguir com a edição do registro 5 (ou o próximo registro lógico em um arquivo indexado), um ^C gravaria o 4 e passaria para o próximo registro. Da mesma forma, um ^R gravaria o 4 e voltaria ao registro anterior no arquivo. Podemos notar que os caracteres ^R e ^C ("avança um registro" e "retrocede um registro") estão localizados mnemonicamente ao lado direito do ^E e ^X ("um campo acima" e "um campo abaixo").

Exemplificaremos cada uma das três funções do EDIT:

1. EDIT sem argumentos

```
. EDIT (*Sem nenhum argumento - o dBase*)
       (*pede o numero do registro*)
```

```
. EDIT
ENTER RECORD # : 4


RECORD # 00004
NOME       :Nancy     :
SOBRENOME :Vieira         :
ENDERECO  :R.do Sol, 444         :
CIDADE    :Sao Paulo :
ESTADO    :SP:                  (*Nós podemos agora fazer as conversões*)
CEP       :92111:               (*e sair do EDIT com W*)
CONTRIB   : 1.33:
PAGAMENTO :T:


     ENTER RECORD #:            (*O dBase pede o próximo número do registro*)
                                (*a editar — pressionando a tecla cr *)
                                (*o dBase volta ao modo interativo*)
```

2. EDIT com o número do registro

```
   . EDIT n     (*Onde n = <um número do registro>*)

. EDIT 3
     (*Vamos diretamente ao registro 3*)

RECORD # 00003
NOME       :Carla     :
SOBRENOME :Cardoso         :
ENDERECO  :R.Bel Monte, 222      :
CIDADE    :Belo Horiz:          (*Aqui, um ^C gravará qualquer troca e*)
ESTADO    :MG:                  (*passará para o próximo registro (lógico) *)
CEP       :21105:
CONTRIB   : 0.00:               (*Agora podemos efetuar as trocas e*)
PAGAMENTO : :                   (*abandonar a edição com ^W *)
```

3. EDIT o registro corrente

```
. EDIT #    (*Edita o registro corrente (#) *)
```

Geralmente sabemos que precisamos editar um registro, mas só podemos identificar qual deles através de algum fato sobre seu conteúdo porque não lembramos ou não sabemos o número do registro. Nestes casos, um comando como FIND ou LOCATE deve ser usado para colocar o indicador no registro apropriado, e então usamos o EDIT com o símbolo do indicador ("#").

```
. LOCATE FOR CIDADE = "Aracaju"
RECORD: 00006
.EDIT #                              (*Não precisamos saber que o indicador = 6*)
RECORD # 00006
NOME      :Zilda     :
SOBRENOME :Zorzi          :
ENDERECO  :Rua Pilar, 44       :     (*Aqui, fazemos qualquer troca desejada e*)
CIDADE    :Aracaju   :               (*usamos ^W para gravar e voltamos ao prompt*)
ESTADO    :SE:
CEP       :     :
CONTRIB   : 8.77:
PAGAMENTO :T:
```

REPLACE – ALTERANDO CAMPOS DENTRO DE UM REGISTRO

O comando REPLACE é usado quando desejamos alterar os conteúdos de campos de um arquivo de dados. Sua sintaxe é:

```
. REPLACE <escopo><nome do campo> WITH <novos conteúdos> <condições>
```

O escopo e as condições podem ser deixadas de lado.

O default do REPLACE é o registro corrente quando não especificamos nenhum escopo. Os novos conteúdos podem ser um valor constante ou uma expressão que necessita de uma avaliação no momento da substituição – um exemplo poderia ser uma expressão para somar uma taxa de vendas de 6% em um determinado preço:

```
. REPLACE ALL CUSTO WITH (1.06 * PREÇO) <cr>
```

Este comando irá de registro em registro em um banco de dados e preencherá o campo CUSTO com 106% do campo PREÇO.

Vamos ilustrar com um exemplo do arquivo SOCIOS:

```
. DISPLAY      (*Onde está o indicador? *)
00006 Zilda      Zorzi          R.Pilar, 44         Aracaju   SE          8.77 .T.


. REPLACE ENDERECO WITH "Rua Pilar, 44"
00001 REPLACEMENT(S)
. DISPLAY
00006 Zilda      Zorzi          Rua Pilar, 44       Aracaju   SE          8.77 .T.


               (*Agora vamos substituir um campo usando uma condição*)


. REPLACE ALL PAGAMENTO WITH T FOR CONTRIB > 0
00005 REPLACEMENT(S)


. LIST
00001 Mila       Moreira        R.Elma, 111         Sao Paulo SP 92109    5.00 .T.
00002 Ana        Amorim         R.Tres, 333         Natal     RN 01224    0.00 .T.
00003 Carla      Cardoso        R.Bel Monte, 222    Belo Horiz MG 21105   0.00 .F.
00004 Nancy      Vieira         R.do Sol, 444       Sao Paulo SP 92111    1.33 .T.
00005 Pedro      Penedo         R.Sao Luiz, 22      Curitiba  PR 92109    0.00 .F.
00006 Zilda      Zorzi          Rua Pilar, 44       Aracaju   SE          8.77 .T.
00007 Carlos     Covas          R. "A", 88          Sao Paulo SP 92101    0.25 .T.
00008 Davi       Dunas          Rua das Flores, 123 Aracaju   SE 92111    1.33 .T.
```

COPY TO – Copiando e Conectando um Arquivo de Dados

O comando COPY TO é usado para fazermos um novo arquivo de disco contendo alguns ou todos os registros do arquivo em uso. Primeiro a sintaxe:

```
. COPY TO <arquivo de disco> [FIELD <lista de campos> ≡ FOR   | <condição> <parâmetros>]
                                                          WHILE |
```

Como com INDEX e SORT, é necessário o nome de um arquivo de saída, mas a lista de campos, condições e parâmetros (os parâmetros serão explicados rapidamente em sua própria seção) são todos opcionais.

A necessidade de um nome de arquivo de disco seria óbvia. O dBase não pode fazer uma cópia de um arquivo de dados sem ter um nome para a cópia. O que pode ser menos óbvio é que o dBase aceitará qualquer nome que for especificado e se um arquivo com aquele nome já existe, *o dBase o apagará sem nos fornecer nenhuma mensagem de aviso.*

Assim, quando SOCIOS.DBF está em uso, os exemplos de comandos corretos serão:

---

1. .COPY TO peqsoci FIELD nome, sobrenome, contrib
   (*Isto criou o arquivo PEQSOCI.DBF que possui os mesmos registros que SOCIOS.DBF, mas possui apenas uma estrutura de três campos — nome, sobrenome e contribuições.*)
2. .COPY TO curisoci FOR cidade = "Curitiba"
   (* Isto criará um novo arquivo chamado CURISOCI.DBF que contém todos os registros em SOCIOS para o qual a cidade é "Curitiba". A ordem em que os registros aparecem em SÓCIOS não afetará esta condição — cada "Curitiba" do topo até o final será copiado no novo arquivo.*)
3. .COPY TO aracjsoc WHILE cidade = "Aracaju"
   (*Diferente do caso 2 acima, a posição do registro é importante para esta condição. Consideremos esta situação: *)

```
00001   Aracaju    ← # (*O indicador do registro*)
00002   Aracaju
00003   Vila Bela
00004   Aracaju
00005   Aracaju
```

Neste caso, assumindo que o indicador iniciou no registro do topo, um novo arquivo chamado ARCJSOC.DBF com a mesma estrutura que SOCIOS seria aberto no disco, e o dBase iniciaria copiando os registros nele. Esta cópia continuou enquanto a condição na cláusula WHILE permaneceu verdadeira – neste caso, somente para os registros 1 e 2.

Note que se o arquivo fosse indexado em cidade, a cláusula WHILE teria o mesmo efeito que FOR; isto é, todos os registros Aracaju seriam copiados. Sempre que a cláusula WHILE ou FOR são verdadeiras sem registro (ou a cláusula WHILE é falsa para o registro corrente no início do comando) então um novo arquivo é criado com a mesma estrutura, mas sem registros.

Ocasionalmente, este é um resultado desejável, e para estes casos o dBase possui um comando especial:

---

4. .COPY STRUCTURE TO <novo nome de arquivo>
   (*Um comando COPY STRUCTURE faz uma réplica da estrutura do arquivo, mas sem nenhum registro. Esta é uma maneira rápida para duplicar uma estrutura de um arquivo sem ter de repetir o processo CREATE.*)

Depois de um comando de cópia, os registros no novo arquivo aparecerão na mesma ordem em que estavam no antigo arquivo – isto é, o dBase copia-os na ordem em que os encontra quando percorre o arquivo.

Isto significa que se um índice está em uso na ocasião da cópia, o arquivo de saída sairá efetivamente classificado.

APPEND FROM – Lendo os Dados de Outros Arquivos

Na Operação 1, usamos o APPEND sem nenhum argumento para iniciar a entrada de dados em tela-cheia através do teclado. O dBase também usa o comando APPEND para ler dados de outros arquivos em disco. Neste caso, a sintaxe completa é:

```
. APPEND FROM <nome do arquivo> [FOR    | <condição>] [<parâmetro>]
                                 WHILE  |
```

Novamente o arquivo de dados que sofrerá a ação – acrescentado – é o arquivo em uso. Quando um comando APPEND FROM é solicitado, o dBase sai para o disco e abre o arquivo cujo nome está ao lado de FROM. Os registros serão acrescentados ao arquivo em uso na ordem em que serão encontrados e de acordo com as condições de FOR/WHILE.

Se o arquivo FROM não puder ser encontrado – se ele não existe ou está em um drive de disco diferente do especificado – o dBase fornece uma mensagem de erro "FILE DOES NOT EXIST" e sai para o diálogo de correção de comandos. Se um ou mais índices estão em uso durante o APPEND FROM, todos os registros acrescentados serão automaticamente indexados durante o processo, o que diminui a velocidade, mas pode economizar tempo durante a operação. Não há proteção automática em relação a duplicação de registros durante um APPEND FROM, pela simples razão que esta situação poderia ser desejada dependendo de nosso banco de dados.

Há uma regra que precisamos lembrar para prevenir confusões possíveis sobre a cláusula de condição. Se esta cláusula em um APPEND FROM referir-se a um campo no arquivo em disco pelo nome, então aquele campo também deve existir no arquivo em uso. Em outras palavras, os dois comandos:

```
. USE SOCIOS
. APPEND FROM grupoidade FOR IDADE > 35
```

conduziria a um erro desde que SOCIOS não possui um campo chamado "IDADE". Este erro ocorreria mesmo que GRUPOIDADE.DBF possuísse um campo IDADE. O campo deve existir no arquivo que está sendo acrescentado.

APPEND BLANK – Um Caso Especial

O comando APPEND BLANK inclui um registro em branco ao final do arquivo de dados em uso, deixando o indicador no registro em branco. Apesar da utilidade deste comando especial

não ser óbvia no ponto que estamos, nós o usaremos (junto com REPLACE) extensivamente nos capítulos de programação.

```
. APPEND BLANK
. DISPLAY
00009                                                                 0.00 .F.

        (*Note que um número em "branco" é 0.00 e*)
        (*o lógico branco é falso                  *)

. REPLACE NOME WITH "Mickey", SOBRENOME WITH "Mouse"
00001 REPLACEMENT(S)
. DISPLAY
00009 Mickey    Mouse                                                 0.00 .F.
. LIST
00001 Mila      Moreira      R.Elma, 111          Sao Paulo  SP 92109  5.00 .T.
00002 Ana       Amorim       R.Tres, 333          Natal      RN 01224  0.00 .T.
00003 Carla     Cardoso      R.Bel Monte, 222     Belo Horiz MG 21105  0.00 .F.
00004 Nancy     Vieira       R.do Sol, 444        Sao Paulo  SP 92111  1.33 .T.
00005 Pedro     Penedo       R.Sao Luiz, 22       Curitiba   PR 92109  0.00 .F.
00006 Zilda     Zorzi        Rua Pilar, 44        Aracaju    SE        8.77 .T.
00007 Carlos    Covas        R. "A", 88           Sao Paulo  SP 92101  0.25 .T.
00008 Davi      Dunas        Rua das Flores, 123  Aracaju    SE 92111  1.33 .T.
00009 Mickey    Mouse                                                  0.00 .F.
```

O novo registro nove foi acrescentado em branco e os conteúdos de NOME e SOBRENOME foram inseridos com REPLACE. Devemos notar que os registros APPEND BLANK que são preenchidos desta forma não são indexados automaticamente.

Agora apague com DELETE e PACK o registro do Mickey para ele não atrapalhar o próximo exemplo.

SDF e DELIMITED – Outras Formas de Arquivos de Dados

No Capítulo 2, dois tipos de formatos de arquivo – "extensão fixa" e "delimitada" – foram descritos. Já que a discussão nesta seção depende do entendimento da distinção entre aqueles formatos de arquivo, revise aquela seção se for necessário.

Quando falamos sobre os comandos COPY TO e APPEND FROM, comentamos sobre os parâmetros opcionais em suas sintaxes. Dois parâmetros de arquivo, "SDF" e "DELIMITED" são permitidos para estes comandos. Quando qualquer um destes parâmetros é incluído na linha de comando, o dBase é alertado que o arquivo que está sendo lido ou escrito não está no formato padrão do dBase. A combinação destas duas opções de entrada de dados, juntamente com a

opção default, permite escolher um dos três tipos de arquivo de saída para a cópia do arquivo em uso. O uso direto de "COPY TO nome de arquivo" sem outros argumentos cria um novo arquivo dBase completo, com cabeçalho, nomes de campos, tipos de dados e extensão de campo (que são os mesmos do arquivo copiado).

Este novo arquivo pode ou não ser uma cópia exata do arquivo antigo, dependendo de fatores como um índice em uso ou uma lista de campo especificada, ou até uma condição incluída ao comando COPY. Mas mesmo que estas variações tenham alterado os conteúdos ou estrutura do novo arquivo, este possui a mesma estrutura, nomes de campos, extensões e tipos que o arquivo antigo. Assim, o dBase pode realizar APPEND FROMs "inteligentes" entre arquivos novos e antigos, já que o APPEND FROM traz os dados para dentro de um arquivo pelo nome do campo.

Porém, às vezes desejamos escrever nossos dados para um arquivo novo que não possua um cabeçalho dBase, de forma que os registros possam ser usados por outro software. Nestas situações, incluindo um SDF ou DELIMITED ao comando COPY TO, podemos instruir o dBase para retirar o cabeçalho da saída e dispor os dados ou em registros de extensão fixa **(Standard Data Format** ou **SDF***) ou registros delimitados.

```
. COPY TO novonome SDF (*extensão fixa*)
. COPY TO novonome DELIMITED (*arquivo delimitado*)
```

Nos dois casos, um arquivo de texto será aberto para a saída com o nome NOVONOME.TXT (a menos que outro tipo de arquivo tenha sido especificado). O SDF é apropriado para Pascal e COBOL e os arquivos DELIMITED são usados pelo BASIC.

Vamos dar alguns exemplos:

DELIMITED

```
. LIST          (*Iniciamos com SOCIOS em uso*)
00001 Mila     Moreira   R.Elma, 111           Sao Paulo  SP 92109  5.00 .T.
00002 Ana      Amorim    R.Tres, 333           Natal      RN 01224  0.00 .T.
00003 Carla    Cardoso   R.Bel Monte, 222      Belo Horiz MG 21105  0.00 .F.
00004 Nancy    Vieira    R.do Sol, 444         Sao Paulo  SP 92111  1.33 .T.
00005 Pedro    Penedo    R.Sao Luiz, 22        Curitiba   PR 92109  0.00 .F.
00006 Zilda    Zorzi     Rua Pilar, 44         Aracaju    SE        8.77 .T.
00007 Carlos   Covas     R. "A", 88            Sao Paulo  SP 92101  0.25 .T.
00008 Davi     Dunas     Rua das Flores, 123   Aracaju    SE 92111  1.33 .T.
```

(*Chamaremos o primeiro arquivo BASIC.TX*)

```
. COPY TO BASIC DELIMITED
00008 RECORDS COPIED
```

(*A cópia se parece com isto*)

```
TYPE BASIC.TXT
'Mila','Moreira','R.Elma, 111','Sao Paulo','SP','92109', 5.00,'T'
'Ana','Amorim','R.Tres, 333','Natal','RN','01224', 0.00,'T'
'Carla','Cardoso','R.Bel Monte, 222','Belo Horiz','MG','21105', 0.00,' '
'Nancy','Vieira','R.do Sol, 444','Sao Paulo','SP','92111', 1.33,'T'
'Pedro','Penedo','R.Sao Luiz, 22','Curitiba','PR','92109', 0.00,'n'
'Zilda','Zorzi','Rua Pilar, 44','Aracaju','SE',' ', 8.77,'T'
'Carlos','Covas','R. "A", 88','Sao Paulo','SP','92101', 0.25,'T'
'Davi','Dunas','Rua das Flores, 123','Aracaju','SE','92111', 1.33,'T'
```

Esta saída pode ser considerada como o default do dBase para arquivo delimitado e merece uma análise. Em primeiro lugar, note a estrutura. Vírgulas separam todos os campos. Os campos de caracteres tiveram todos os espaços em branco eliminados e foram cercados por aspas simples, e os valores numéricos foram listados sem o delimitador de aspas simples.

E sobre o PAGAMENTO, o campo lógico? O dBase simplesmente copiou os conteúdos dos campos como caracteres – o mesmo que digitamos originalmente. Se sete arquivos fossem conectados com um programa escrito em BASIC, teríamos de incluir uma rotina em código BASIC que reconheceria os caracteres no campo PAGAMENTO como valores verdadeiros ou falsos.

Vamos observar também o código postal para o registro seis, Zilda Zorzi. Em SOCIOS, este registro não possui código postal. Porém, na saída em BASIC.TXT, o registro possui um campo CEP que consiste em um espaço delimitado com aspas simples. Na convenção de arquivos delimitados este não é um campo vazio, mas um campo preenchido com um espaço.

```
'Aracaju', 'PA', ' ', 8.77, 'y'     (*Há um espaço aqui.*)
'Aracaju', 'PA', , 8.77, 'y'   (*Este é um campo de CEP vazio.*)
```

Agora compare os dois exemplos isolados abaixo:

```
    'Ana', 'Amorim', 'R. Tres, 333', 'Natal', 'RN', '01224', 0.00, 'T'
e
    'Ana', 'Amorim', 'Rua Tres s/n', 'RN', 01224', 0.00, 'T'
```

No segundo exemplo, está claro que "Rua Tres s/n" é um simples endereço, ao passo que o interpretador Basic pense que o "333" seja a cidade, "Natal" o Estado e assim por diante.

Já devemos ter percebido a confusão criada pelas vírgulas que fazem parte do endereço de cada registro. Nem o BASIC nem qualquer outro software vai conseguir diferenciar o que é uma vírgula delimitadora de campo. Isto é um problema grave!

Interface com o WordStar

Vários programas de computação podem "conversar" com o dBase através de arquivos delimitados. A opção MailMerge do popular Processador de Texto WordStar, por exemplo, permite aos usuários intercalar arquivos de dados delimitados com documentos para aplicações tais como cartas. Porém, o WordStar não reconhece o delimitador de aspas simples em volta dos campos de caracteres – WordStar usa aspas duplas.

Nestas situações, o dBase permite especificar um outro caractere delimitador que não as aspas simples. Vamos escrever um arquivo de texto chamado WORDSTAR.TXT:

```
. COPY TO WORDSTAR DELIMITED WITH "
00008 RECORDS COPIED

(*Agora vamos observar*)

B>TYPE WORDSTAR.TXT
"Mila","Moreira","R.Elma, 111","Sao Paulo","SP","92109", 5.00,"T"
"Ana","Amorim","R.Tres, 333","Natal","RN","01224", 0.00,"T"
"Carla","Cardoso","R.Bel Monte, 222","Belo Horiz","MG","21105", 0.00," "
"Nancy","Vieira","R.do Sol, 444","Sao Paulo","SP","92111", 1.33,"T"
"Pedro","Penedo","R.Sao Luiz, 22","Curitiba","PR","92109", 0.00,"n"
"Zilda","Zorzi","Rua Pilar, 44","Aracaju","SE"," ", 8.77,"T"
"Carlos","Covas","R. "A", 88","Sao Paulo","SP","92101", 0.25,"T"
"Davi","Dunas","Rua das Flores, 123","Aracaju","SE","92111", 1.33,"T"
```

Note que quanto mais manipulável é esta especificação de um delimitador, mais erros poderemos cometer. No registro Carlos Covas, o endereço já contém aspas duplas na rua "A". Estamos com outro problema. Uma boa prática para evitar situações como estas é prever a utilização de DELIMITED e criar normas para preenchimento dos campos.

"SDF e Arquivos de Extensão Fixa

Se a discussão acima fez-nos considerar que arquivos de dados delimitados representam um aborrecimento, isto é normal. É bom termos uma opção delimitadora para BASIC e WordStar, mas também é bom possuirmos a opção de formato de dados padrão quando não desejamos nos envolver em vírgulas e aspas. Vamos escrever um arquivo SDF chamado PASCAL.TXT:

```
. COPY TO PASCAL SDF

00008 RECORDS COPIED
```

```
B>TYPE PASCAL.TXT
Mila     Moreira   R.Elma, 111          Sao Paulo SP92109 5.00T
Ana      Amorim    R.Tres, 333          Natal     RN01224 0.00T
Carla    Cardoso   R.Bel Monte, 222     Belo HorizMG21105 0.00
Nancy    Vieira    R.do Sol, 444        Sao Paulo SP92111 1.33T
Pedro    Penedo    R.Sao Luiz, 22       Curitiba  PR92109 0.00n
Zilda    Zorzi     Rua Pilar, 44        Aracaju   SE      8.77T
Carlos   Covas     R. "A", 88           Sao Paulo SP92101 0.25T
Davi     Dunas     Rua das Flores, 123 Aracaju   SE92111 1.33T
```

O ponto mais importante para notarmos aqui é que os campos aparecem sem nenhum espaço a mais entre eles, como em "SP92109" – o Estado e CEP do registro 1. Mesmo onde os campos aparecem separados, como em "Mila Moreira", os espaços entre os dois representam espaços não preenchidos no campo nome.

Os programadores em Pascal devem ter cuidado ao desenhar estruturas de registros para que correspondam às extensões de campos do arquivo de saída SDF do dBase.

MODIFY STRUCTURE – Alterando uma Estrutura dBase

Existem ocasiões em que necessitamos trocar a estrutura de um arquivo de dados dBase. As extensões de campo não combinam ou o campo definido como um número precisa ser redefinido como uma série de caracteres ou, ainda, o nome de campo necessita de uma troca. Às vezes, campos necessitam ser adicionados ou apagados da estrutura. E geralmente essas trocas tornam-se necessárias depois que uma quantia significativa de dados tenha entrado; os dados que desejamos gravar sob a nova estrutura.

O comando dBase MODIFY STRUCTURE permite-nos fazer estas trocas chamando o cabeçalho do arquivo e editando-o como um arquivo de texto. Depois que as mudanças são feitas, o novo arquivo será automaticamente redefinido com as mudanças incorporadas em sua estrutura. Há apenas um porém. Quando MODIFY STRUCTURE é usado para alterar um cabeçalho, torna-se impossível para o dBase manter os dados antigos na nova estrutura – então a parte dos dados do arquivo antigo é apagada. Por esta razão, antes de usarmos o comando MODIFY STRUCTURE, o operador deve fazer uma cópia temporária do arquivo usando COPY TO. Depois de usarmos MODIFY STRUCTURE, os dados podem então entrar novamente no arquivo modificado usando APPEND FROM.

```
. COPY TO GUARDA
00008 RECORDS COPIED


. MODIFIY STRUCTURE
```

```
. MODIFY STRUCTURE
MODIFY ERASES ALL DATA RECORDS ... PROCEED? (Y/N) Y
```

(*O dBase nos avisa sobre o apagamento*)

```
              NAME         TYPE  LEN  DEC
FIELD 01  : NOME           C     010  000  :  (*Trocar LEN para 008.*)
FIELD 02  : SOBRENOME      C     010  000  :
FIELD 03  : ENDERECO       C     020  000  :
FIELD 04  : CIDADE         C     015  000  :
FIELD 05  : EST            C     002  000  :
FIELD 06  : CEP            C     005  000  :
FIELD 07  : CONTRIB        N     005  002  :
FIELD 08  : PAGAMENTO      L     001  000  :
FIELD 09  :                                :
FIELD 10  :                                :  (*Estes campos não são usados*)
FIELD 11  :                                :
FIELD 12  :                                :
FIELD 13  :                                :
FIELD 14  :                                :
FIELD 15  :                                :
FIELD 16  :                                :
FIELD 17  :                                :
FIELD 18  :                                :
FIELD 19  :                                :
FIELD 20  :                                :
FIELD 21  :                                :
FIELD 22  :                                :
```

(*Aqui um ^W grava as trocas.*)

Durante a edição de MODIFY STRUCTURE, as combinações de (^E, ^S, ^D e ^X) movem o cursor e ^Q (termina sem gravar) e ^W (grava). Dois novos caracteres de controle aparecem aqui; o ^T apaga um campo movimentando todos os campos de baixo para cima e o ^N insere um espaço em branco acima do cursor para colocarmos um novo campo.

```
. LIST STRUCTURE
STRUCTURE FOR FILE:  B:SOCIOS  .DBF
NUMBER OF RECORDS:   00000
DATE OF LAST UPDATE: 01/01/80
PRIMARY USE DATABASE
```

```
FLD       NAME       TYPE WIDTH   DEC
001       NOME         C   008
002       SOBRENOME    C   015
003       ENDERECO     C   020
004       CIDADE       C   010
005       ESTADO       C   002
006       CEP          C   005
007       CONTRIB      N   005     002
008       PAGAMENTO    L   001
** TOTAL **               00067

. APPEND FROM GUARDA              (*Obtemos os dados de volta*)
00008 RECORDS ADDED
```

Note que os dados podem ser armazenados em outros formatos de arquivo além de um arquivo dBase padrão. Como podíamos presumir os mesmos parâmetros que podem controlar os formatos dos arquivos de dados em COPY TO podem ser usados com APPEND FROM.

Entretanto, o usuário deve ter cuidado em prever a interação dos parâmetros para que obtenha os resultados desejados. Vamos adicionar os conteúdos do arquivo de extensão fixa PASCAL.TXT ao conteúdo presente de SOCIOS.DBF, lembrando que tiramos dois caracteres do campo NOME.

```
. APPEND FROM PASCAL SDF
00008 RECORDS ADDED

. LIST
00001 Mila    Moreira         R.Elma, 111           Sao Paulo  SP 92109   5.00 .T.
00002 Ana     Amorim          R.Tres, 333           Natal      RN 01224   0.00 .T.
00003 Carla   Carduso         R.Bel Monte, 222      Belo Horiz MG 21105   0.00 .F.
00004 Nancy   Vieira          R.do Sol, 444         Sao Paulo  SP 92111   1.33 .T.
00005 Pedro   Penedo          R.Sao Luiz, 22        Curitiba   PR 92109   0.00 .F.
00006 Zilda   Zorzi           Rua Pilar, 44         Aracaju    SE         8.77 .T.
00007 Carlos  Covas           R. "A", 88            Sao Paulo  SP 92101   0.25 .T.
00008 Davi    Dunas           Rua das Flores, 123   Aracaju    SE 92111   1.33 .T.
00009 Mila       Moreira         R.Elma, 111           Sao Paul o SP921   9.00 .F.
00010 Ana        Amorim          R.Tres, 333           Natal      RN012  24.00 .F.
00011 Carla      Cardoso         R.Bel Monte, 222      Belo Hor iz MG211  5.00 .F.
00012 Nancy      Vieira          R.do Sol, 444         Sao Paul o SP921  11.00 .F.
00013 Pedro      Penedo          R.Sao Luiz, 22        Curitiba   PR921   9.00 .F.
00014 Zilda      Zorzi           Rua Pilar, 44         Aracaju    SE      8.00 .F.
00015 Carlos     Covas           R. "A", 88            Sao Paul o SP921   1.00 .F.
00016 Davi       Dunas           Rua das Flores, 123 Aracaju    SE921  11.00 .F.
```

Os primeiros oito registros foram acrescentados ao arquivo dBase. Os nomes dos campos foram checados e os dados surgiram claramente. Os oito últimos registros, porém, foram acrescentados como dados de extensão fixa e conseqüentemente foram todos deslocados para a direita por dois caracteres, a partir do sobrenome. Realmente, estamos com um arquivo "furado".

Regras do Comando MODIFY STRUCTURE

Apresentaremos aqui algumas regras para selecionarmos a classificação dos parâmetros de COPY TO para usarmos com MODIFY STRUCTURE:

1. Se vamos incluir ou apagar campos ou trocar a ordem deles, usamos COPY TO sem parâmetros (fazemos um outro arquivo .DBF).
2. Se vamos mudar um nome de campo, usamos os parâmetros DELIMITED ou SDF para COPY TO e APPEND FROM.
3. Se vamos mudar um tipo de dado, usamos os parâmetros SDF.
4. Se quisermos experimentar tudo sobre o mecanismo de um MODIFY STRUCTURE, devemos fazer uma cópia extra do arquivo antes de começar a brincadeira. Em outras palavras, se "acreditamos" que o primeiro passo correto é "COPY TO GUARDA SDF", por que não fazer "COPY TO BACKUP" primeiro? Isso pode ser seguro.

Como exercício, para nos certificarmos de entendimento claro do uso de MODIFY STRUCTURE, sugerimos a confecção de um pequeno banco de dados. Preencha-o com alguns registros e então modifique sua estrutura de várias maneiras. O desafio pode iniciar em tentar modificar dois aspectos simultaneamente; por exemplo, uma extensão de campo e outra de nome, ou uma extensão de campo e outra de tipo, ou ainda as três.

Lembre-se de que, sempre que uma estrutura tenha sido modificada, a indexação para aquele arquivo de dados deve ser refeita através de um processo de REINDEX, a menos que o índice tenha usado um campo cujo nome tenha sido trocado. Neste caso, o INDEX deve ser usado para formar um novo índice.

CAPÍTULO

4

O Modo Interativo (II)

Este capítulo contém as Seções 4, 5 e 6, e apresentará os últimos comandos básicos dBase.

SEÇÃO 4 – CONTROLANDO A CONFIGURAÇÃO

Quando ligamos um computador, seus circuitos e microplaquetas podem ser considerados como uma configuração interna, a composição que determinará exatamente o que acontecerá em cada uma das situações. A configuração é definida pelo tipo da unidade de microplaqueta do processador central da máquina e pelo sistema operacional que estamos usando. No passo seguinte, o programa de aplicação carregado na máquina determinará o que deve ser feito.

Algumas condições de configuração como os "níveis de ajuda" que os programas como WordStar possuem são ativados quando digitamos o programa e sempre podem ser ativadas – "instaladas" – quando quisermos mudar o default. Os exemplos de configuração instaladas que o dBase possui são o pedido de entrada de data, que pode ser instalado para "mês/dia/ano" ou "dia/mês/ano", e o comando de diálogo de correção que pode ser suprimido na instalação. Outros fatores de configuração dependem da conveniência. Quais arquivos estão no nosso drive de disco? Seria muito bom se pudéssemos saber abandonar o dBase. Onde nossos dados serão gravados? Não queremos ficar o tempo todo digitando "B:" na frente dos nomes de arquivo.

O dBase possui um conjunto de comandos de configuração para alterar a maneira que o programa funcionará durante uma operação. Existem 25 comandos de configuração específicos dos quais 24 iniciam com o verbo SET e muitos deles encontram-se nesta seção – na verdade, alguns são desconhecidos até para programadores avançados. Os efeitos de alguns destes comandos não podem ser demonstrados nas páginas de um livro. Na Seção 4, teremos de operar em nossa máquina se quisermos ver os efeitos.

Os comandos que veremos nesta operação incluem:

LIST (ou DISPLAY) STATUS	CLEAR
SET	RENAME
LIST < ... > OFF	RESET
DISPLAY FILE [LIKE]	QUIT TO
DELETE FILE [LIKE]	

Dos 24 SETs, veremos:

BELL	ESCAPE
CARRY	EXACT
CONFIRM	INTENSITY
CONSOLE	RAW
DEFAULT	TALK
DELETE	

Já vimos o SET INDEX TO, que pode ser discutido em capítulos mais apropriados.

DISPLAY STATUS – Observando a Configuração

O comando DISPLAY STATUS é manipulável em nível de comando para dizer ao operador qual o status de 18 SETs:

```
. DISPLAY STATUS


TODAYS DATE        - 00/00/00
DEFAULT DISK DRIVE - A:

ALTERNATE - OFF   BELL      - ON
CARRY     - OFF   COLON     - ON
CONFIRM   - OFF   CONSOLE   - ON
DEBUG     - OFF   DELETE    - OFF
ECHO      - OFF   EJECT     - ON
ESCAPE    - ON    EXACT     - OFF
INTENSITY - ON    LINKAGE   - OFF
PRINT     - OFF   RAW       - OFF
STEP      - OFF   TALK      - ON
```

(*Quando um arquivo de dados está em uso, o dBase nos fornece duas telas do DISPLAY STATUS. A primeira mostra o arquivo em uso (com todas as suas indexações e chaves), e a segunda tela mostra os SETs.*)

```
. DISPLAY STATUS

DATABASE SELECTED - B:SOCIOS  .DBF
PRIMARY USE DATABASE

INDEXES:           KEY EXPRESSION:
 B:SOCID   .NDX    CIDADE
 B:SOBREN  .NDX    L:NOME
WAITING
```

(*Esta mensagem de espera congela a tela*)
(*a próxima tecla pressionada (qualquer uma) libera*)

```
TODAYS DATE         - 00/00/00
DEFAULT DISK DRIVE - B:

ALTERNATE - OFF    BELL      - ON
CARRY     - OFF    COLON     - ON
CONFIRM   - OFF    CONSOLE   - ON
DEBUG     - OFF    DELETE    - OFF
ECHO      - OFF    EJECT     - ON
ESCAPE    - ON     EXACT     - OFF
INTENSITY - ON     LINKAGE   - OFF
PRINT     - OFF    RAW       - OFF
STEP      - OFF    TALK      - ON
```

Os "SETs"

Estes 18 indicadores de status são chamados "SETs" porque o comando para trocá-los é:

```
. SET <nome> OFF/ON
```

Os valores na listagem do status acima são os defaults do dBase.

As definições para os SETs mais freqüentemente usados no modo interativo são (o default aparece em letras maiúsculas):

BELL – (ON) O alarme do computador acionará sempre que um campo for completamente preenchido durante um APPEND ou sempre que for feita uma entrada com dados incorretos, tal como letras em um campo numérico. Quando o BELL está desligado, um dado incorreto também é recusado, mas não há alarme. O alarme pode perturbar um pouco, mas é muito útil para alertar o digitador de que um campo terminou.

CARRY – (OFF) Durante os APPENDs, cada tela de registro novo normalmente aparece vazia. Quando ativamos CARRY, cada registro novo aparece com a imagem do último registro entrado. O CARRY é muito útil quando inserimos muitos registros que usam campos comuns, tais como cidade, estado, código postal e assim por diante.

CONFIRM – (OFF) Normalmente, durante os APPENDs, o dBase pula para o próximo campo quando alcançamos o final de um campo (veja BELL). Quando ativamos CONFIRM, o dBase deixará o cursor no campo que foi preenchido, para confirmação, até que <cr> ou uma tecla de controle do cursor seja pressionada.

CONSOLE – (ON) Quando ativamos CONSOLE, todas as saídas vão para a tela. Quando desativamos CONSOLE, nada vai para a tela até que a trazemos de volta.

DELETE – (OFF) Quando desativamos o DELETE, um FIND em um arquivo de dados indexado colocará o indicador no primeiro registro que satisfaça a pesquisa. Quando DELETE é ativado, porém, os registros marcados para o apagamento não serão reconhecidos pelo FIND, LOCATE, COUNT, LIST ou qualquer outro verbo que aceite uma frase escopo com NEXT.

ESCAPE – (ON) Normalmente, comandos de operação longa (LIST, por exemplo) ou arquivos de comandos podem ser cancelados digitando um <esc> (ou alguma tecla de controle). Quando desativamos ESCAPE, porém, não há nenhuma maneira de interromper tais comandos.

EXACT – (OFF) Os FINDs normais com indexações ativam o indicador para o primeiro registro que não foi excluído – isto é, "Simões" seria localizado por FINDs em "S", "Si", "Sim" e assim por diante. Quando ativamos EXACT, "Simões" só pode ser encontrado por "Simões". Os brancos são ignorados.

INTENSITY – (ON) Na maioria das edições de tela-cheia, o dBase coloca a área de entrada para o usuário em vídeo inverso (caracteres pretos em uma tela branca ou verde em vez de uma branca em preto normal). Quando desativamos INTENSITY, o dBase não usa vídeo inverso. Como BELL, é uma questão de preferência; a menos que possuamos um terminal sem uma micro-plaqueta de caractere inverso, neste caso teremos de desativar INTENSITY se quisermos ver os dados.

RAW – (OFF) Normalmente em um LIST ou DISPLAY, o dBase coloca espaços entre os campos para melhor clareza na leitura:

```
. LIST NEXT 4   (*Normal*)
00001  Mila   Moreira    R.Elma, 111          Sao Paulo  SP 92109   5.00 .T.
00002  Ana    Amorim     R.Tres, 333          Natal      RN 01224   0.00 .T.
00003  Carla  Cardoso    R.Bel Monte, 222     Belo Horiz MG 21105   0.00 .F.
00004  Nancy  Vieira     R.do Sol, 444        Sao Paulo  SP 92111   1.33 .T.
. GOTO TOP

. SET RAW ON
                                  (*Note que o campo lógico é mostrado*)
. LIST NEXT 4                     (*como realmente foi digitado      *)
```

```
00001 Mila   Moreira   R.Elma, 111        Sao Paulo SP92109 5.00T
00002 Ana    Amorim    R.Tres, 333        Natal     RN01224 0.00T
00003 Carla  Cardoso   R.Bel Monte, 222   Belo HorizMG21105 0.00
00004 Nancy  Vieira    R.do Sol, 444      Sao Paulo SP92111 1.33T
```

TALK – (ON) O TALK refere-se a todas as telas de resposta do dBase para nossos comandos – "00008 REPLACEMENT(S)" por exemplo. Normalmente é muito útil, mas às vezes ele diminui a velocidade dos programas. Em geral, mantenha TALK ativado, especialmente ao testar comandos e programas.

Além destes SETs ON/OFF, outros SETs podem ter seus valores ativados. Apresentaremos um aqui:

DEFAULT – Como vimos anteriormente, o comando de entrada:

```
B > A:DBASE
```

não funcionará, porque o dBase deve estar no drive de disco default. Entretanto, freqüentemente o usuário ou o programador quer ter o dBase em um drive e os programas e dados em outro.

Para reativarmos o drive default de dentro do dBase, usamos o comando:

```
. SET DEFAULT TO x:  (*"x" é o designador de drive.*)
```

Este é o novo default do dBase e todos os nomes de arquivos subseqüentes serão assumidos para iniciar com "x: " até que algum outro drive seja especificado.

LIST <...> OFF

Na definição anterior de SET RAW, notamos que este comando de configuração nos permite fazer LISTs de registros longos sem ultrapassar limite à direita da tela. O LIST também permite o parâmetro extra "OFF", que simplesmente esconde os números dos registros:

```
. LIST NEXT 4 OFF
Mila    Moreira    R.Elma, 111        Sao Paulo  SP 92109   5.00 .T.
Ana     Amorim     R.Tres, 333        Natal      RN 01224   0.00 .T.
Carla   Cardoso    R.Bel Monte, 222   Belo Horiz MG 21105   0.00 .F.
Nancy   Vieira     R.do Sol, 444      Sao Paulo  SP 92111   1.33 .T.
```

DISPLAY FILES – Observando o Diretório do Disco

O comando DISPLAY FILES do dBase é o equivalente ao DIR do sistema operacional; isto é, ele mostra o diretório do disco. Como DIR, DISPLAY FILES reconhece a indicação de "coringas" e seu default é o drive corrente mas ele pode ser direcionado para outro drive.

A diferença de DIR é que o comando sozinho sem argumentos não mostra o diretório inteiro do disco. Ao contrário, mostra somente os arquivos de dados que terminam com o tipo de arquivo ".DBF". Nesta aplicação específica, DISPLAY FILES também abre cada cabeçalho de arquivo e mostra a última data que foi atualizada e o número de registros no arquivo.

A sintaxe completa de DISPLAY FILES é:

```
. DISPLAY FILES [ON x: ] [LIKE <coringa>]
```

Aqui, como nos outros, as chaves indicam as partes opcionais do comando.

```
. DISPLAY FILES        (*O default — .DBFs somente*)

DATABASE FILES   # RCDS   LAST UPDATE

SOCIOS   DBF     00008    00/00/00
TEMPO    DBF     00008    00/00/00

. DISPLAY FILES ON A:         (*Pede os .DBFs em A: *)

DATABASE FILES   # RCDS   LAST UPDATE

None
              (*Algum índice no drive do default? *)
. DISPLAY FILES LIKE *.NDX

SOCID   .NDX        SOBREN  .NDX        CONTR   .NDX
```

Como em muitos outros casos, DISPLAY e LIST são sinônimos.

```
. LIST FILES            (*O mesmo que DISPLAY FILES*)

DATABASE FILES   # RCDS   LAST UPDATE

SOCIOS   DBF     00008    00/00/00
TEMPO    DBF     00008    00/00/00
```

DELETE FILE – Limpando o Disco

De vez em quando desejaremos apagar arquivo sem abandonar o dBase, ou de dentro de um programa ou no modo interativo. O comando DELETE FILE fará isto, se o arquivo mencionado não estiver aberto (em uso, sendo gravado ou lido).

O comando DELETE FILE não pode ter dúvidas sobre o nome do arquivo. Os coringas não funcionarão. Se o arquivo nomeado não existir (ou não existe no drive de disco indicado pelo default), o dBase nos dirá.

```
. DELETE FILE TEMPO.DBF
FILE HAS BEEN DELETED
. DELETE FILE LIKE *.NDX
FILE DOES NOT EXIST
. DELETE FILE SOCIOS.DBF
FILE CURRENTLY OPEN
```

RENAME – Mudando um Nome de Arquivo

Quanto mais estudamos computação, mais reconhecemos a importância das convenções de nomear um arquivo. O dBase nos permite mudar um nome de arquivo usando o comando RENAME. Sua sintaxe é:

```
. RENAME <nome antigo> TO <novo nome>
```

Não podemos mudar o nome de qualquer arquivo que esteja aberto atualmente. Se o arquivo que estamos renomeando não possuir um tipo de arquivo, devemos finalizar seu nome com um ponto ou o dBase não o reconhecerá.

Podemos notar que RENAME não possui checagem de entrada e podemos usá-lo para criar nomes de arquivo completamente ilegais, como "D?: .DBF", que o dBase acessará normalmente, mas que o sistema operacional recusará manipular.

Faça isto se desejar, mas não recomendamos.

RESET – Liberando o Disco para "Leitura/Gravação"

Sempre que os discos são trocados em um drive em CP/M, temos de digitar um ^C para iniciar um carregamento antes que possamos gravá-los. Sem o carregamento, o novo disco é marcado como sendo "R/O" ou "somente leitura", pelo sistema operacional.

A razão para isto está fora do escopo deste livro, mas a implicação para um programador em dBase é óbvia. Se planejarmos fazer alguma atualização durante a operação com o dBase

ou dentro de nossos programas, necessitaremos de um comando para acionar o novo disco para ler/gravar. Este comando é o RESET.

O RESET possui apenas duas regras. A primeira, é que ele sempre precisa de um designador de drive como um argumento:

```
. RESET x: (*Onde "x: " pode ser qualquer drive ligado*)
```

A segunda, é que todos os arquivos no drive RESET devem estar fechados (isto é, nada "em uso"). Quando o RESET aparece em um programa no sistema operacional IBM, ele é ignorado, assim um programa dBase em CP/M pode ser operado em uma máquina tipo IBM sem modificações.

QUIT TO – Chamada Automática de Programa

Somente em CP/M o dBase permitirá encadear outro programa ou séries de comandos do sistema operacional quando o desativarmos. A sintaxe é:

```
. QUIT TO "<comando 1>", "<comando 2>", ...
```

Note as aspas nos comandos – cada comando é uma *série.*

O dBase afeta o encadeamento usando estas *séries* para escrever um pequeno arquivo SUBMIT apresentado apenas no momento da saída. O SUBMIT.COM não precisa estar mencionado – ele se auto-executa.

Como exemplo, vamos imaginar que desejamos sair do dBase, chamar um diretório de todos os arquivos .TXT no drive C: e então chamar o WordStar no drive A:, abrindo um "C:PASCAL;TXT" para a edição do WordStar. O comando de uma linha para isto seria:

```
. QUIT TO "DIR C: *.TXT", "A: WS C: PASCAL.TXT"
```

O dBase escreverá o arquivo SUBMIT e passará o controle ao CP/M. Conseguiremos um diretório de C:, e o WordStar surgirá editando C: PASCAL.TXT. Para maiores informações sobre os arquivos SUBMIT pesquise a documentação do CP/M.

SEÇÃO 5 – MANIPULANDO VARIÁVEIS DE MEMÓRIA

Uma das áreas de configuração do dBase é tão especificada em seu uso que merece sua própria seção – a memória.

Na Seção 5, veremos os comandos que criam, atualizam e gravam as variáveis de memória, bem como alguns dos comandos que as utilizam. Finalmente, o ponto de interrogação será examinado pela segunda vez.

Os comandos na Seção 5 são:

?	RESTORE
STORE	COUNT
DISPLAY MEMORY	SUM
RELEASE	& (macros)
SAVE	

Variáveis de Memória

O dBase permite ao usuário a criação de até 64 variáveis de memória de uma vez.

As variáveis de memória podem ser consideradas como prateleiras nas quais o usuário ou programador possa guardar itens de dados que possam ser úteis mais tarde. Por exemplo, se um usuário deseja determinar que um campo de sobrenome no registro 1 seja comparado ao mesmo campo no registro 2, ele pode visualizar o registro 1, lembrar o sobrenome, pular para o registro 2 e aí comparar.

Mas como o computador pode fazer esta comparação? O comando:

```
. ?SOBRENOME (REC# 00001) = SOBRENOME (REC#00002) (*Não funcionará*)
```

não existe. Se o computador for comparar os conteúdos de um registro com outro, necessitará de um lugar para armazenar dados variáveis. Para isto, usamos a memória.

As variáveis de memória podem ser consideradas como campos especiais, criados para o momento. Como os campos e registros, uma variável possui nome, extensão e um tipo de dado. Se ela for de tipo numérico, também possuirá um lugar definido para as casas decimais. As variáveis de memória do dBase são dinâmicas em vez de declaradas. Isto é, o fato de armazenar algo na memória cria a variável de memória e define seu tipo, extensão e assim por diante. Os nomes das variáveis de memória seguem as mesmas regras que os nomes dos campos — eles podem ter até 10 caracteres alfanuméricos, sempre iniciando com uma letra (dois pontos (:) são permitidos). As letras maiúsculas e minúsculas são consideradas da mesma forma para os nomes das variáveis, mas não em seus conteúdos.

Vamos exemplificar. Atualize o nosso arquivo SOCIOS.DBF conforme a listagem a seguir. Ele servirá para os próximos exemplos:

```
. USE SOCIOS

. LIST

00001 Mila     Moreira   R.Elma, 111           Sao Paulo SP 92109  5.00 .T.
00002 Ana      Amorim    R.Tres, 333           Natal     RN 10010  0.00 .T.
00003 Sonia    Simoes    R.Para, 831           Belem     PA 21105  0.00 .F.
00004 Monica   Medeiros  R. Diva, 2219         Curitiba  PR 44110  5.99 .T.
00005 Pedro    Penedo    R.Sao Luiz, 22        Sao Paulo SP 92109  0.00 .F.
00006 Zilda    Zorzi     Rua Pilar, 44         Aracaju   SE        8.77 .T.
00007 Carlos   Covas     R. "A", 88            Sao Paulo SP 92101  0.25 .T.
00008 Davi     Dunas     Rua das Flores, 123   Aracaju   SE 36222  1.00 .T.
00009 Ires     Imare     R.Neusa, 99           Belem     PA 21233  7.50 .F.
00010 Henrique Hipolito  R.Doze, 21            Sao Paulo SP 92111 21.00 .T.
```

O ? – O Comando de Impressão do dBase

Até agora usamos nas operações o ponto de interrogação como um comando para perguntar ao dBase o que significa alguma coisa. Digitamos ".? NOME" e o dBase responde o conteúdo do campo NOME para o registro indicado. Da mesma forma, ".? 7.000/2" responde "3.500".

Usaremos o comando "?" extensivamente na Seção 5. A explicação que segue pode ajudar em nosso entendimento: o ponto de interrogação em dBase é o equivalente a uma declaração "PRINT" em BASIC ou uma declaração "writeln" em Pascal. Ele avalia uma expressão e imprime os resultados como uma saída. Se a expressão for um nome de uma variável, o comando de ponto de interrogação localizará a variável e sairá com os conteúdos. Se for usado por si só, ele imprimirá a tecla return, que será útil mais tarde para o espacejamento de saída.

```
. GOTO TOP

. DISPLAY
00001 Mila     Moreira   R.Elma, 111           Sao Paulo SP 92109  5.00 .T.

. ? NOME
Mila

. ? SOBRENOME
Moreira

. ? NOME,SOBRENOME          (*Note a vírgula*)
Mila     Moreira
```

```
. ? 2 * CONTRIB
  10.00                    (*Duas casas decimais, como em CONTRIB*)

. ? PAGAMENTO
.T.                        (*Os dois pontos nos diz que é "Verdadeiro", não letra "T." *)

. ? (2.000/7)+3
    3.285
```

STORE – O Comando de Designação do dBase

Cada linguagem de computação possui pelo menos um comando para designar valores para variáveis. Em dBase, a primeira declaração de designação está no verbo STORE. Sua sintaxe é:

```
STORE <valor ou expressão> TO <nome da variável>
```

Lembre-se de que STORE só afeta as variáveis de memória – ele nunca troca os conteúdos de um arquivo de dados.

Em outras palavras, o comando "STORE 'Artur' TO NOME" não produziria uma mensagem de erro, mas ele não trocaria o conteúdo do campo NOME no registro corrente. O que ele poderia fazer é criar uma variável de memória que também é chamada "NOME", mas que é independente do arquivo de dados. O comando para alterar o registro é, logicamente, REPLACE.

```
. STORE "Esta e uma string" TO STRING
Esta e uma string          (*O dBase repete a entrada para checagem*)

. ? STRING                 (*Vamos observar novamente*)
Esta e uma string
```

LIST (ou DISPLAY) MEMORY

Os conteúdos de uma memória podem ser listados ou visualizados a qualquer tempo. Vamos iniciar com uma memória vazia:

```
. LIST MEMORY
** TOTAL **     00 VARIABLES USED  00000 BYTES USED
```

Agora a memória depois que armazenamos "Esta é uma string".

```
. LIST MEMORY
STRING      (C)  Esta e uma string
** TOTAL **      01 VARIABLES USED  00018 BYTES USED
```

Note que a saída fornece o nome da variável, o tipo e o número usado (o limite é 64, lembra-se?) e o espaço total ocupado.

Agora colocaremos um número na memória:

```
. STORE 6.345 TO NUMERO
 6.345                        (*O dBase repete*)

. ? NUMERO/2                  (*Aritmética básica*)
      3.172                   (*Truncado por três decimais*)

. LIST MEMORY
STRING      (C)  Esta e uma string
NUMERO      (N)   6.345
** TOTAL **      02 VARIABLES USED  00025 BYTES USED

. STORE F TO BOOLEAN          (*Não digitamos .F. *)
.F.                           (*O dBase cuida dos pontos*)

. LIST MEMORY
STRING      (C)  Esta e uma string
NUMERO      (N)   6.345
BOOLEAN     (L)  .F.
** TOTAL **      03 VARIABLES USED  00027 BYTES USED
```

Note que quando dividimos NÚMERO por 2, o conteúdo de NÚMERO não foi afetado. A única maneira de alterar o conteúdo da variável de memória é fazer um outro STORE:

```
. STORE NUMERO/2 TO NUMERO       (*Correto*)
 3.172

. LIST MEMORY
STRING      (C)  Esta e uma string
NUMERO      (N)   3.172
BOOLEAN     (L)  .F.
** TOTAL **      03 VARIABLES USED  00027 BYTES USED
```

Lembre-se de que quando manipulamos variáveis de memória, devemos respeitar seus tipos. O comando ".?NUMBER/STRING" produziria um erro de sintaxe.

RELEASE – Liberando Variáveis da Memória

Já que a versão 2.4 do dBase possui uma limitação de 64 variáveis na memória de cada vez, é incumbência dos programadores limpá-las após o uso para que seus programas não abortem com a horrível mensagem "TOO MANY MEMORY VARIABLES". O comando para liberar as variáveis de memória é RELEASE. O RELEASE pode ser usado para eliminar as variáveis pelo seu nome ou podem ser usadas para a eliminação total.

```
. RELEASE STRING              (*Uma de cada vez pelo nome*)

. LIST MEMORY
NUMERO      (N)   3.172
BOOLEAN     (L)  .F.
** TOTAL **     02 VARIABLES USED  00009 BYTES USED

. RELEASE ALL                 (*Um tanto precipitado*)

. LIST MEMORY
** TOTAL **     00 VARIABLES USED  00000 BYTES USED
```

Na versão 2.4 do dBase, o RELEASE também pode ser usado com coringa para liberar variáveis de memória específicas por classe. O programador ou usuário deve fornecer os nomes para as variáveis de memória que responderão a um RELEASE coringa. Vamos voltar ao nosso quadro original de memória para demonstrar isto.

```
. LIST MEMORY
STRING      (C)  Esta e uma string
NUMERO      (N)   6.345
BOOLEAN     (L)  .F.
** TOTAL **     03 VARIABLES USED  00027 BYTES USED
```

Agora vamos criar uma classe de variáveis de memória que iniciem com um "T:".

```
. STORE 1 TO T:VAL1
  1
. STORE 2 TO T:VAL2
  2
. STORE 3 TO T:VAL3
  3
```

```
. LIST MEMORY
STRING      (C)  Esta e uma string
NUMERO      (N)   6.345
BOOLEAN     (L)  .F.
T:VAL1      (N)    1
T:VAL2      (N)    2
T:VAL3      (N)    3
** TOTAL **     06 VARIABLES USED  00048 BYTES USED

. RELEASE ALL LIKE T:*        (*O asterisco é o coringa*)

. LIST MEMORY
STRING      (C)  Esta e uma string
NUMERO      (N)   6.345
BOOLEAN     (L)  .F.
** TOTAL **     03 VARIABLES USED  00027 BYTES USED
```

CLEAR – Um Caso Especial para Limpar a Memória

CLEAR é um comando de caso especial que poderia ser introduzido em vários lugares em um texto dBase. Ele possui vários efeitos poderosos na configuração do dBase, o primeiro servirá para liberar todas as variáveis de memória, o outro fecha todos os arquivos. E seu terceiro efeito é para voltar o usuário para a área primária, um conceito que será explicado no Capítulo 8.

O valor de CLEAR surge do fato que saberemos precisamente com que configuração o dBase se apresentará depois de chamado. Por exemplo, poderíamos usar CLEAR antes da mudança de um disco para fechar todos os arquivos e preparar o sistema operacional para um novo disco.

```
. CLEAR     (*Fecha todos os arquivos*)
. RESET C:  (*Prepara o sistema para um novo disco*)
```

SAVE e RESTORE – Gravando uma Imagem de Memória no Disco

As variáveis de memória em dBase são perdidas quando desligamos a máquina ou quando saímos do dBase com QUIT.

Porém, o dBase escreverá uma imagem de memória no disco para ser recuperada posteriormente, se o comando SAVE for usado. Esta imagem pode ser colocada de volta na memória usando o comando RESTORE. Já que SAVE e RESTORE referem-se a arquivos de disco, suas sintaxes requerem um nome de arquivo:

```
. SAVE TO <nome de arquivo> [ALL LIKE <variável do coringa>]
. RESTORE FROM <nome de arquivo> [ADDITIVE]
```

Vamos fazer uma operação:

```
. LIST MEMORY    (*A figura do início*)
STRING      (C)  Esta e uma string
NUMERO      (N)   6.345
BOOLEAN     (L)  .F.
** TOTAL **      03 VARIABLES USED  00027 BYTES USED

. SAVE TO GRAVA    (*O arquivo GRAVA.MEM é formado no disco*)

. RELEASE ALL

. LIST MEMORY   (*Poderia estar vazio*)
** TOTAL **      00 VARIABLES USED  00000 BYTES USED

. DISPLAY FILES LIKE *.MEM    (*Existe um GRAVA.MEM? *)

GRAVA   .MEM    (*Existe, com certeza*)


. RESTORE FROM GRAVA    (*Chamando de volta*)

. LIST MEMORY   (*Poderia estar como antes*)
STRING      (C)  Esta e uma string
NUMERO      (N)   6.345
BOOLEAN     (L)  .F.
** TOTAL **      03 VARIABLES USED  00027 BYTES USED
```

O arquivo RECORD.MEM permanece no disco depois do RESTORE e nenhuma das manipulações ou trocas na memória irão alterar o GRAVA.MEM. Para trocarmos a imagem no arquivo MEM, temos de usar o SAVE novamente.

Nas versões do dBase anteriores a 2.4 rearmazenar um arquivo MEM significava um propósito de tudo-ou-nada. Se possuíssemos um valor importante na memória e necessitássemos rearmazenar uma imagem de memória, teríamos de fazer proezas para conseguirmos juntar tudo na memória. Consideremos uma memória BANCO:NUM que contém a string "90-1234/1222". É importante rearmazenar o conteúdo de FINANÇAS.MEM sem perder BANCO:NUM. Antigamente, poderíamos fazer isto assim:

```
. APPEND BLANK   (*Um lugar para guardar*)
. REPLACE NOME WITH BANCO:NUM    (*Esta errado, mas e dai ?*)
. RESTORE FROM FINANCAS  (*Vai apagar tudo que estava na memoria*)
. STORE NOME TO BANCO:NUM  (*Leva BANCO:NUM de volta a memoria*)
. DELETE   (*Apaga o registro em branco - obrigado*)
```

Agora, porém, RESTORE possui o argumento de inclusão ADDITIVE, para dizer que a nova memória será acrescentada à antiga. Esta simples alteração é uma das melhorias mais significativas da programação dBase. Primeiro exemplificaremos a maneira antiga – RESTORE elimina a memória antiga aqui:

```
. RELEASE ALL

. STORE "QUALQUER COISA" TO X
QUALQUER COISA

. LIST MEMORY
X            (C)  QUALQUER COISA
** TOTAL **     01 VARIABLES USED  00015 BYTES USED

. RESTORE FROM GRAVA

. LIST MEMORY    (*A variável de memória "X" desapareceu*)
STRING       (C)  Esta e uma string
NUMERO       (N)   6.345
BOOLEAN      (L)  .F.
** TOTAL **     03 VARIABLES USED  00027 BYTES USED
```

Agora, um exemplo da maneira nova:

```
. RELEASE ALL

. STORE "Exemplo" TO VAR:NOVA
Exemplo

. LIST MEMORY
VAR:NOVA     (C)  Exemplo
** TOTAL **     01 VARIABLES USED  00008 BYTES USED

. RESTORE FROM GRAVA ADDITIVE
```

```
. LIST MEMORY
VAR:NOVA    (C)  Exemplo
STRING      (C)  Esta e uma string
NUMERO      (N)   6.345
BOOLEAN     (L)  .F.
** TOTAL **      04 VARIABLES USED  00035 BYTES USED
```

Também com a versão 2.4 surgiu a habilidade em usar SAVE com coringa. Em outras palavras, um arquivo MEM pode ser escrito em um disco contendo somente uma classe específica de variáveis na memória.

Vamos recuperar de GRAVA e incluir uma variável lógica chamada "NOPE" (falsa), para que possamos ter duas variáveis que iniciem com um "N", NOPE e NÚMERO.

```
. RESTORE FROM GRAVA

. STORE F TO NOPE
.F.
```

Agora fazemos uma gravação usando o coringa:

```
. SAVE TO N-TESTE ALL LIKE N*

. RELEASE ALL        (*Apaga tudo*)

. RESTORE FROM N-TESTE      (*O que gravou? *)

. LIST MEMORY
NUMERO      (N)   6.345
NOPE        (L)  .F.
** TOTAL **      02 VARIABLES USED  00009 BYTES USED
```

COUNT – Contando Registros Específicos

COUNT é um dos comandos mais úteis em dBase, já que ele nos fornece um método para determinar rapidamente quantos registros em um arquivo de dados encontram um critério qualquer. A sintaxe de COUNT é:

```
. COUNT [TO <nome da var. mem.>] [<escopo>] [FOR/WHILE <condição>]
```

Em sua forma simples, o comando COUNT conta todos os registros em um arquivo de dados. Ele conta os registros marcados para o apagamento se o SET DELETE for desativado e ignora-os se SET DELETE for ativado. Atualize o SOCIOS.DBF conforme a listagem abaixo. Ele servirá para os próximos exemplos.

```
. USE SOCIOS

. LIST
00001 Mila     Moreira   R.Elma, 111           Sao Paulo SP 92109  5.00 .T.
00002 Ana      Amorim    R.Tres, 333           Natal     RN 10010  0.00 .T.
00003 Sonia    Simoes    R.Para, 831           Belem     PA 21105  0.00 .F.
00004 Monica   Medeiros  R. Diva, 2219         Curitiba  PR 44110  5.99 .T.
00005 Pedro    Penedo    R.Sao Luiz, 22        Sao Paulo SP 92109  0.00 .F.
00006 Zilda    Zorzi     Rua Pilar, 44         Aracaju   SE        8.77 .T.
00007 Carlos   Covas     R. "A", 88            Sao Paulo SP 92101  0.25 .T.
00008 Davi     Dunas     Rua das Flores, 123   Aracaju   SE 36222  1.00 .T.
00009 Ires     Imare     R.Neusa, 99           Belem     PA 21233  7.50 .F.
00010 Henrique Hipolito  R.Doze, 21            Sao Paulo SP 92111 21.00 .T.

. COUNT
COUNT = 00010

. COUNT FOR CIDADE = "Sao Paulo"
COUNT = 00004
```

O uso de COUNT no modo interativo é tão valioso quanto no modo programado. Qualquer número conseguido com ele pode ser armazenado em uma variável de memória no momento da contagem.

```
. COUNT TO MEM:VAR FOR CIDADE = "Sao Paulo"
COUNT = 00004

. LIST MEMORY   (*Para provar que está lá*)
MEM:VAR     (N)    4
** TOTAL **      01 VARIABLES USED  00007 BYTES USED

. ? MEM:VAR     (*Usar o "?" é uma outra maneira de saber*)
         4

. ? MEM:VAR/2   (*Como um número, MEM:VAR é assunto para aritmética*)
         2
```

Muitas vezes a resposta para uma pergunta complicada sobre um arquivo de dados pode ser encontrada através de um comando COUNT, seguido da condição especificada na cláusula FOR.

SUM – Somando Campos em um Banco de Dados

Como COUNT, SUM caminha pelo arquivo pegando dados dos registros; porém, neste caso, o resultado será a soma dos conteúdos em um ou mais campos numéricos, até cinco campos. SUM ignora os registros apagados. A sintaxe é:

```
. SUM <campo> [ <campo>] [TO <lista varmem>] [ <escopo>] [FOR/WHILE <cond>]
```

Vamos trabalhar com SUM usando a mesma imagem de SOCIOS acima:

```
. SUM CONTRIB   (*Isto representa todos os registros não apagados*)
49.51

. SUM CONTRIB TO DINH FOR CIDADE = "Aracaju"
9.77

. ? DINH
      9.77
                  (*Os campos somados também podem ser usados em uma expressão*)
. SUM (CONTRIB * 0.06) TO SAOPA:TX FOR ESTADO = "SP"
1.5750

. ? SAOPA:TX
    1.5750
```

Notem a disparidade sintática entre a maneira que SUM e COPY manipulam os registros apagados e a maneira com que LIST e COUNT os processa. LIST e COUNT consideram todos os registros igualmente quando SET DELETE está desativado (seu default) e ignoram os registros apagados se SET DELETE está ativado. SUM e COPY simplesmente não fazem nada com os registros marcados para apagamento.

Macros – Substituições de String

Macros representam um dos instrumentos mais poderosos disponíveis aos programadores dBase e são normalmente a primeira coisa que aparecem em mente quando um programador pensa sobre o dBase. Apresentaremos as macros aqui e faremos alguns testes, mas a sua potencialidade ficará clara mais tarde. Por enquanto, vamos aprender o básico, acreditando que será importante.

O conceito de uma macro é simples: armazenamos uma longa string como uma variável de memória com um nome curto. Depois, podemos chamar a string por aquele nome e o dBase responderá a string como se tivéssemos digitado a forma longa.

Em dBase, o comando para chamar uma macro é o &, que representa o sinal "e" em "Botelho & Irmãos". Vamos observar um exemplo sofisticado:

```
. STORE "Senatus Populusque Romanorum" TO SPQR
Senatus Populusque Romanorum
. ? "&SPQR"          (*Note as aspas — lembre-se de que esta é uma string! *)
Senatus Populusque Romanorum
```

Aqui estão algumas regras importantes de macros:

1. O & deve estar junto do nome da variável – qualquer espaço, como em "& SPQR" e o & será apenas outro caractere.
2. As macros servem para puxar somente os conteúdos das variáveis de memória, não os campos do arquivo de dados. A chamada para "&NOME" quando SOCIOS está em uso produziria uma mensagem de erro.
3. *Macros são usadas para strings e somente strings.* Os dados na variável de memória DEVEM ser de tipo caractere para serem chamados como uma macro. E o lugar onde a chamamos DEVE ser um comando pelo qual a string esteja sintaticamente correta.

Em suas aplicações mais comuns, as macros podem eliminar algumas digitações gravando alguns comandos usados freqüentemente para um arquivo chamado TRAB.MEM.

```
. STORE "USE SOCIOS" TO U
USE SOCIOS

. STORE "LIST" TO L
LIST


. STORE "MEMORY" TO M
MEMORY

. &U   (*O mesmo que digitar "USE SOCIOS no prompt*)

. &L   (*O mesmo que digitar "LIST" *)
00001 Mila    Moreira    R.Elma, 111     Sao Paulo  SP 92109   5.00 .T.
00002 Ana     Amorim     R.Tres, 333     Natal      RN 10010   0.00 .T.
00003 Sonia   Simoes     R.Para, 831     Belem      PA 21105   0.00 .F.
```

```
00004 Monica    Medeiros  R. Diva, 2219         Curitiba  PR 44110   5.99 .T.
00005 Pedro     Penedo    R.Sao Luiz, 22        Sao Paulo SP 92109   0.00 .F.
00006 Zilda     Zorzi     Rua Pilar, 44         Aracaju   SE         8.77 .T.
00007 Carlos    Covas     R. "A", 88            Sao Paulo SP 92101   0.25 .T.
00008 Davi      Dunas     Rua das Flores, 123   Aracaju   SE 36222   1.00 .T.
00009 Ires      Imare     R.Neusa, 99           Belem     PA 21233   7.50 .F.
00010 Henrique  Hipolito  R.Doze, 21            Sao Paulo SP 92111  21.00 .T.

          (*Vamos observar a memória*)
. &L &M   (*O mesmo que "LIST MEMORY"*)
U           (C)  USE SOCIOS
L           (C)  LIST
M           (C)  MEMORY
** TOTAL **     03 VARIABLES USED  00025 BYTES USED
```

As macros podem encaixar-se, aparecer dentro de strings ou podem ser expandidas nas variáveis de memória:

```
. STORE "&L &M" TO B     (*Expansão na memória*)
LIST MEMORY

. &B
U           (C)  USE SOCIOS
L           (C)  LIST
M           (C)  MEMORY
B           (C)  LIST MEMORY
** TOTAL **     04 VARIABLES USED  00038 BYTES USED

. STORE "TRIB" TO X     (*O final do nome do campo conTRIB*)
TRIB

. DISPLAY
00010 Henrique  Hipolito  R.Doze, 21            Sao Paulo SP 92111  21.00 .T.

. ? CON&X/2     (*O mesmo que "? CONTRIB/2"*)
  10.50
```

As macros também podem ser **concatenadas** (ajustadas) com outras variáveis de string, usando ou os manipuladores de string (que serão explicados no Capítulo 5) ou um comando especial de macros, o ponto ("."). Vamos armazenar "Ele" à memória e depois incluir "fante":

```
. RELEASE ALL

. STORE "Ele" TO E
Ele

. ? "&Efante"          (*Primeira tentativa — o dBase considera Efante" *)
&Efante                (*como um nome de varmem então ele repete a string*)

. ? "&E fante"         (*Segunda tentativa — usamos um espaço*)
Ele fante              (*mas o espaço foi incluído — é mal*)

. ? "&E.fante"         (*Esta é a maneira correta*)
Elefante               (*E este é o resultado*)
```

TRABALHANDO COM A MEMÓRIA DO dBASE

Pratique os comandos da operação cinco até que tenha em mente uma boa imagem de seus efeitos. Controlar a memória e saber o que está acontecendo com o dBase, são necessidades fundamentais para a programação.

Quando nos tornamos mais capazes devemos adotar bons hábitos como:

1. Gravar as imagens da memória freqüentemente e com nomes significativos. "SAVE TO GUARDA" foi adequado para a nossa seção, mas pode não ser ideal quando houver alguma coisa mais importante na memória.
2. Libere as variáveis de memória quando não forem mais necessárias. Isto será muito importante mais tarde em programação. Sessenta e quatro variáveis de memória podem parecer muito, mas logo tornam-se uma limitação restrita.
3. Se for apropriado, estabeleça um código sistemático para as variáveis de memória, tais como "S:" (não confunda com designador de drive) para iniciar todas as variáveis a serem gravadas no arquivo "S.MEM". A habilidade em enviar variáveis diferentes para arquivos de memórias diferentes é poderosa e a prática excelente.

SEÇÃO 6 – USANDO A IMPRESSORA

Na sexta e última seção deste capítulo aprenderemos o básico para usarmos a impressora em conjunto com o dBase. Algumas das colocações serão práticas, consistindo em técnicas de impressão adequadas para o modo interativo do dBase mas não para relatórios formais. Alguns dos comandos, principalmente REPORT FORM e ALTERNATE, são usados extensivamente na programação dBase.

Nesta seção, mais do que nas outras, seremos forçados a dizer o que acontecerá quando desenvolvermos um procedimento determinado, acreditando que será experimentado em seus próprios computadores.

Os comandos que veremos na seção 6 são:

? e ??	REPORT FORM
^P	SET DATE
SET PRINT ON/OFF	SET HEADING
SET ALTERNATE	SET MARGIN
EJECT e SET EJECT	

Impressão

A impressão como uma operação é raramente discutida em uma documentação e nós temos de aprender através de tentativas e erros. Para ajudarmos a reduzir os problemas apresentaremos algumas colocações fundamentais.

Durante a operação, o computador envia à impressora uma corrente de caracteres, um de cada vez. A maioria destes caracteres é impressa, mas alguns controlam outros aspectos da operação da impressora, tais como a separação ou o tipo de estilo dos caracteres, a cor da fita ou a posição do papel. Estes caracteres de controle não são muito bem padronizados e enquanto alguns são universais, outros são particulares da impressora e têm de ser compreendidos por fora de sua documentação.

Freqüentemente estes controles são seqüências de caracteres, chamadas **seqüências de escape** já que é uma prática comum iniciar uma seqüência de controle da impressora com o caractere <esc> (decimal 27 de ASCII). Um exemplo seria a impressora matricial Epson, que usa "<esc> E" (escape e E maiúsculo), para voltar ao estilo de tipo enfatizado ou "<esc> ^O" (escape e control O), para voltar ao estilo de tipo comprimido (tamanho reduzido à metade).

Essas seqüências de escape podem e deveriam ser chamadas de dentro dos programas dBase, já que elas são sempre necessárias para realizarmos saídas variadas – o tipo comprimido, por exemplo, pode colocar um relatório extenso em uma folha de papel-padrão. Todos os exemplos neste livro serão seqüências controladas para a Epson MX-100; porém, elas podem ser adaptadas para nossa impressora substituindo as seqüências de controle apropriadas.

Alguns caracteres de controle são padronizados. Tais caracteres são a tecla return(^M), que sinaliza o final de uma linha, a tecla backspace (^H) ou o avanço de linha (^J), que avança uma linha no papel.

A maioria das impressoras eletrônicas inclui a estes o **alimentador de formulários** (^L), que sinaliza a impressora para avançar o papel ao próximo "topo de formulário". Em outras palavras, quando o computador envia um ^L para a impressora, ela avança tanto papel quanto necessário para alcançar o topo da próxima página, que é geralmente marcado por uma linha picotada para fácil ruptura.

É de responsabilidade do usuário observar se a impressora entenderá corretamente onde é o topo do formulário – do contrário, se a impressora iniciar a impressão no meio da página, continuará a avançar o papel para o mesmo lugar nas páginas subseqüentes sempre que for recebido um alimentador de formulários, desajustando todos os relatórios. O topo do formulário pode ser ativado através do software de dentro de um programa e algumas impressoras possuem "TOF" ou "TOP" para uso no painel de controle. Se nada disso funcionar, ajuste o papel com a mão, desligue a máquina e ligue de novo.

As impressoras são muito mais lentas que os computadores que as dirigem – é como comparar minutos com dias – depois que o computador mandar um caractere para a impressora, ele espera pelo reconhecimento da impressora e está pronto para outro. O resultado prático disto para o programador é que se o computador inicia a impressão e não há impressora em linha, o computador trava. Ele considera que está esperando pelo reconhecimento de uma impressora lenta, mas para o operador, ele parece desativado – pressiona as teclas e nada acontece. Veremos rotinas nos capítulos de programação que nos protegem contra isto.

Finalmente, às vezes a impressão pode ser operada considerando a página impressa como se fosse uma tela do computador, um método que será demonstrado no Capítulo 7. Lembre-se de que quando usar este método, porém, o cursor pode ser movimentado pela tela e a impressora só funciona da esquerda para a direita, de cima para baixo. Se nunca usarmos em nossa impressora nada mais complicado do que uma tecla backspace ou uma linha de impressão sobreposta, descobriremos o quanto são horríveis os alarmes de aviso.

^P – Repetindo a Saída na Impressora

Os usuários dos sistemas operacionais CP/M e MS-DOS conhecem os efeitos de digitar um ^P. Este caractere de controle age como um interruptor, ligando e desligando a impressora, de maneira que cada caractere que é digitado na tela também será repetido na impressora.

O ^P também funciona no modo interativo do dBase. A seqüência

```
. ^P
. LIST
```

mostrará o arquivo de dados em uso, na tela e na impressora; porém, o comando "LIST" também será impresso.

Para evitar isto, digite a seguinte seqüência:

```
. LIST ^P <cr> (* ^P antes de <cr> *)
```

Lembre-se de digitar outro ^P quando terminar a listagem, ou os comandos subseqüentes também serão impressos.

O comando ^P funcionará para todos os comandos que enviam saídas para a tela, exceto APPEND e EDIT. Nestes dois comandos, a tela mostra, na verdade, uma visão do buffer do arquivo, e o sistema operacional não possui uma rotina para mandá-lo para a impressora.

EJECT – Enviando um Sinal Alimentador de Formulário

Como mencionamos na seção de impressão, quando uma impressora recebe um ^L ou um caractere de alimentação de formulário, ela avançará o papel para o próximo início do formulário. Esta habilidade é tão útil que o dBase possui o comando EJECT, que faz apenas isto – envia um ^L. Se a impressora foi colocada em linha através de um ^P ou algum outro comando, a página saltará. Se a impressora estiver fora de linha, o EJECT não terá efeito.

O comando EJECT é útil em outra situação. Muitas impressoras possuem um pequeno buffer de memória que recebem caracteres, geralmente o equivalente a duas linhas. Ocasionalmente quando completamos uma impressão, os caracteres no buffer da impressora permanecem lá, mesmo depois que o computador tenha terminado a rotina. Nestas situações, o uso de um EJECT no final de cada impressão forçará a saída daqueles caracteres de fora do buffer para o papel.

? e ?? – Mais Comandos de Impressão

O comando de impressão (?) do dBase possui um companheiro: é o comando (??).

Traduzindo, este comando seria "Imprima na mesma linha o que está na de cima".

Assim, em um programa, os comandos

```
. ? 'Helena'
. ?? 'Pereira'
```

sairia

```
Helena Pereira
```

SET PRINT ON/OFF – Imprimindo de Dentro dos Programas

Embora o ^P seja manipulável e efetivo em acionar a impressora quando começarmos a programar, necessitaremos de um comando para ligarmos e desligarmos a impressora de dentro do software. O dBase substitui o comando SET PRINT ON para o primeiro ^P e SET PRINT OFF para o segundo.

Os dois SET PRINTs são absolutos; isto é, se a impressora já estiver desligada, SET PRINT OFF não terá efeito e se estiver ligada, SET PRINT ON não fará nada. Isto é ótimo para os programadores, já que não temos de contar os ^Ps e nos mantermos informados em relação aos números pares ou ímpares.

SET TALK ON/OFF – Calando o dBase

Já que o dBase imprime repetindo na impressora o que é impresso na tela e conseqüentemente envia mensagens de aviso por si próprio, precisamos de um dispositivo que nos permita desligar estas mensagens. Elas estão reunidas com o título de TALK e podem ser escondidas pelo "SET TALK OFF".

TALK é muito útil e deve sempre ser ligado depois de uma operação de impressão, se estivermos trabalhando no modo interativo. Quando enviamos um comando para o qual esperamos um resultado satisfatório é importante não esconder a mensagem de volta, "00000 REPLACEMENT(S)". É algo que precisamos saber. Por outro lado, uma mensagem como esta não possui significado no meio de uma impressão de etiquetas de correio, mas isto é o que acontecerá se não desligarmos o TALK.

SET ALTERNATE TO/ON/OFF – A Saída em Arquivo de Disco

Os destinos mais freqüentes para a saída em dBase e qualquer outra linguagem são a tela ou a impressora. Como já vimos, o dBase não faz distinção entre os dois e um ^P ou SET PRINT ON é o suficiente para enviar um relatório de tela para a impressora.

Ocasionalmente, porém, poderíamos desejar enviar a saída para o drive de disco, para ser armazenado como um arquivo de texto. Um exemplo disto poderia ser incluir um relatório gerado por um programa de cliente em um documento que foi editado por um processador de texto. O dBase possui um dispositivo para abrir um arquivo de disco e repetir todos os caracteres enviados para a tela naquele arquivo de disco.

O processo possui duas etapas. A primeira, o arquivo é nomeado e aberto no disco com o comando:

```
. SET ALTERNATE TO <nome de arquivo>
```

A segunda, a seqüência de caracteres repetidos no arquivo de ALTERNATE é ligado ou desligado com o dispositivo:

```
. SET ALTERNATE ON/OFF
```

Quando o arquivo ALTERNATE é aberto, está no estado OFF. Entretanto, ele está aberto e qualquer tentativa de trocar os discos ou usar RESET colocará o arquivo aberto em dificuldade.

Se ativarmos o ALTERNATE para outro nome de arquivo ou usarmos CLEAR ou QUIT, o arquivo se fechará imediatamente e não poderá ser aberto, já que uma chamada subseqüente de "SET ALTERNATE TO <mesmo nome do arquivo>" abrirá um novo arquivo para aquele nome e apagará o antigo. Como com ^P, o arquivo ALTERNATE pegará e gravará cada caractere que está indo para a tela com exceção dos APPENDs e EDITs.

Os arquivos criados possuirão o tipo de arquivo de ".TXT" por default, a menos que outro seja especificado. Estes arquivos são de texto-padrão para o sistema operacional usado e podem ser editados com um processador de textos, impressos etc.

Arquivos alternados não substituem o comando COPY TO com SDF e DELIMITED. O COPY TO é mais útil para a conexão com outros programas de aplicação, tal como planilhas. Arquivos alternados são mais úteis para gravar relatórios.

REPORT FORM – Gerando Relatórios

A escolha do comando adequado leva o programador a economizar tempo, que é a parte mais cara em computação, e a produzir relatórios aceitáveis e úteis que podem ser usados de várias maneiras com um único arquivo de dados ou vários outros parecidos.

O REPORT FORM é um bom economizador de tempo, entretanto, alguns programadores e seus clientes rejeitam a forma simples dos seus relatórios, preferindo pagar duas ou três vezes mais para ter relatórios que mostram os mesmos dados de uma forma mais vistosa. Esta escolha depende de cada um.

Vamos iniciar com uma revisão. O REPORT FORM gerará uma imagem para visualizar os dados do arquivo (ou arquivos – veja o Capítulo 8) em uso de uma maneira pré-definida. Como no comando CREATE do dBase, REPORT FORM indica ao usuário para desenvolver o relatório através do diálogo no modo interativo e a saída do diálogo é então gravada em um arquivo .FRM que poderá ser usada novamente no futuro.

Neste estágio, vamos criar um pequeno relatório para o arquivo de dados SOCIOS que mostrará as contribuições feitas por cidade. O primeiro passo é colocar o arquivo de dados em uso. E o comando para definir o relatório é:

```
. REPORT (*O dBase pedirá um nome de arquivo*)
                         ou
. REPORT FORM <novo nome de arquivo>
```

Na segunda forma, o dBase checará para ver se o arquivo com aquele nome já existe e se existir, o relatório será impresso. Se o nome de arquivo for novo, porém, o dBase irá para o diálogo para defini-lo. Primeiro vamos fazer isto e depois comentaremos:

```
. USE SOCIOS

. REPORT FORM Contrib    (*O que segue é o diálogo*)

ENTER OPTIONS, M=LEFT MARGIN, L=LINES/PAGE, W=PAGE WIDTH
PAGE HEADING? (Y/N) Y
ENTER PAGE HEADING: RELAT. TESTE:CONTRIBUICOES
DOUBLE SPACE REPORT? (Y/N) N
ARE TOTALS REQUIRED? (Y/N) Y
SUBTOTALS IN REPORT? (Y/N) Y
ENTER SUBTOTALS FIELD: CIDADE
SUMMARY REPORT ONLY? (Y/N) N
EJECT PAGE AFTER SUBTOTALS? (Y/N) N
ENTER SUBTOTAL HEADING: Cidade =
COL     WIDTH,CONTENTS
001     10,nome
ENTER HEADING:        (*para deixar em branco*)
002     14,sobrenome
ENTER HEADING: <sobrenome
003     8,contrib
ENTER HEADING: Contrib
ARE TOTALS REQUIRED? (Y/N) y
004     <cr>
```

Quando chamamos o REPORT FORM, dizemos ao dBase que queríamos nomear o arquivo de formulário "CONTRIB" – o tipo de arquivo ".FRM" será incluído pelo default. O dBase checou o drive de disco default para "CONTRIB.FRM" e não o encontrou, assumindo que este era um arquivo novo e iniciou o diálogo.

A primeira opção relaciona-se com o tamanho da página do relatório:

```
ENTER OPTIONS, M = LEFT MARGIN, L = LINES/PAGE, W = PAGE WIDTH <cr>
```

Preferimos aceitar os valores do default digitando a tecla return. Eles são M = 8, L = 57 e W = 80, mas poderíamos ter trocado um ou mais daqueles valores digitando com algo como "M = 12, W = 128". O valor da largura é apenas para centralizar o cabeçalho.

A opção dois pergunta se haverá algum cabeçalho, e se dissermos que sim, seremos indagados sobre qual será:

```
PAGE HEADING? (Y/N) Y
ENTER PAGE HEADING: RELAT.TESTE;CONTRIBUIÇÕES
```

Note o ponto e vírgula no cabeçalho. Aqui e nos outros lugares, o dBase interpretará o ponto e vírgula como sendo um sinal de fim de linha e nosso cabeçalho aparecerá centralizado em duas linhas. Além do ponto e vírgula, os caracteres especiais "<" e ">" podem ser usados como o primeiro caractere de qualquer cabeçalho definido em REPORT FORM para fazer com que o cabeçalho ajuste-se à esquerda ou direita respectivamente.

A pergunta sobre o espaço duplo é clara por si mesma. Para que o dBase pergunte sobre os totais, um campo numérico deve estar presente na estrutura do arquivo de dados. Com uma resposta positiva, o dBase perguntará sobre os subtotais e um sim perguntará qual o campo a ser subtotalizado.

```
ARE TOTALS REQUIRED? (Y/N) Y
SUBTOTALS IN REPORT? (Y/N) Y
ENTER SUBTOTALS FIELD:CIDADE
```

O campo do subtotal é crítico, já que fornece ao programador a melhor opção para separar registros dentro do relatório. O mecanismo é: o dBase processará os registros no arquivo um por um através do relatório em sua ordem aparente. Sempre que mudar o conteúdo do campo subtotal para o próximo – em nosso relatório, sempre que mudar a cidade – o dBase fará o subtotal antes de continuar.

É da responsabilidade do programador usar uma indexação ou classificação apropriada para que a ordem aparente do arquivo forneça os subtotais quando desejado. Geralmente, isto significa indexar no campo do subtotal antes de operar o relatório. Os exemplos que seguem deixarão isto mais claro.

As três próximas perguntas referem-se ao tratamento dos subtotais:

```
SUMMARY REPORT ONLY? (Y/N) N
EJECT PAGE AFTER SUBTOTALS? (Y/N) N
ENTER SUBTOTAL HEADING:Cidade
```

Um "Relatório Resumido" mostrará apenas as linhas de subtotais e não imprimirá os registros que produziram os subtotais – muito útil às vezes.

Um salto de página depois dos subtotais é a nossa alimentação de formulário, e o efeito será que cada categoria do subtotal (neste caso, cada cidade) estará em sua própria página ou páginas. Como regra para o manuseio, se cada classe de subtotal possuir um número grande de registros, um salto de página fará sentido.

Observaremos os exemplos a seguir para ver onde o dBase colocará o cabeçalho dos subtotais no relatório. Como uma outra regra, o próximo nome do campo pode ser um bom cabeçalho.

Ele leva a uma redundância visual como em:

```
     Nome              Cidade          Contrib

* Cidade = Sao Paulo
Mila       Moreira    Sao Paulo         5.00
Pedro      Penedo     Sao Paulo         0.00
Carlos     Covas      Sao Paulo         0.25
Henrique   Hipolito   Sao Paulo        21.00
** SUBTOTAL **
                                       26.25
```

O restante do processo de REPORT FORM consiste em definir os campos como no comando CREATE. Diferente do CREATE, porém, a largura do campo é definida primeiro e em seguida os conteúdos dos campos. Depois que cada campo é definido, o dBase pergunta pelo seu cabeçalho. Os caracteres de controle de saída ponto e vírgula, "<" e ">" podem ser usados.

Os conteúdos dos campos do relatório podem ser desenhados em qualquer combinação, com campos do arquivo, variáveis de memória (elas devem existir), literais e constantes numéricas combinadas em qualquer expressão legítima.

A largura do campo do relatório pode ser maior ou menor que a largura do campo do arquivo que ele mostra. Se o campo do relatório for maior, os dados serão simplesmente enviados com os caracteres ajustando-se à esquerda na coluna e os números ajustando-se à direita. Se o arquivo for maior do que o campo do relatório, os dados serão distribuídos em quantas linhas forem necessárias. Um campo com 80 caracteres colocados em um campo de relatório de 50 caracteres ultrapassará duas linhas. O dBase tenta quebrar estes campos nos espaços.

Depois que o programador definiu o último campo do relatório, um <cr> em um campo vazio terminará o processo da definição. O dBase mandará imediatamente um novo relatório para a tela, onde ele pode ser interrompido com um ^S. Se o relatório for o que o programador possuía em mente, ele pode ser enviado para a impressora ou para um arquivo ALTERNATE, como qualquer outra saída de tela. Ele pode ser chamado novamente a qualquer momento pelo comando:

```
. REPORT FORM <nome do formulário> [<escopo>] [TO PRINT] [PLAIN] [FOR/WHILE
  <condições>]
```

Destas opções, a única nova é a "PLAIN" – ele suprime a data do relatório, o número da página e o cabeçalho opcional de página (veja a explicação de SET HEAD TO).

Um relatório, uma vez definido, pode ser usado com qualquer arquivo de dados que possua os campos mencionados em sua estrutura. Entretanto, um relatório pode fornecer saídas diferentes significativas, dependendo da indexação do arquivo de entrada. Por exemplo, SOCIOS produziria dois relatórios diferentes se fosse submetido ao mesmo .FRM duas vezes, primeiro indexado por SOBRENOME e depois por CIDADE.

Mais sobre o processo — apresentaremos aqui o que aconteceu enquanto nos interamos do diálogo do REPORT FORM com o dBase. O dBase abriu um arquivo de texto no disco e armazenou suas respostas às perguntas do REPORT FORM. Na próxima vez que chamarmos o relatório, o dBase abrirá o arquivo e usará nossas respostas antigas. É simples.

Apresentaremos como se parece o nosso arquivo CONTRIB.FRM:

```
CONTRIB.FRM
=====================  (*O arquivo começa na linha seguinte *)
                       (*Linha em branco onde estaria o "W = 128" *)
Y
RELAT. TESTE:CONTRIBUICOES
N
Y
Y
CIDADE
N
N
Cidade =
10,nome
                       (*Linha em branco do cabeçalho não informado *)
14,sobrenome
<sobrenome
8,contrib
Contrib
y
=======================  (*O arquivo terminou na linha acima *)
```

A implicação disto é que um relatório poderia ser modificado usando um processador de texto ou um editor e na verdade isto funciona. Como a longa lista de "Ys" e "Ns" mostra, é melhor estarmos muito familiarizados com o REPORT FORM antes de manusearmos com os yes/no do arquivo. Porém, é relativamente simples trocar os cabeçalhos dos campos, larguras e os caracteres de controle.

Aqui está uma parte de nosso relatório CONTRIB;FRM, com SOCIOS sem qualquer indexação. Como podemos ver, desde que o dBase mande os registros em sua ordem de entrada, e cada um possui uma cidade diferente, o relatório gerou um subtotal para cada registro.

```
PAGE NO. 00001

                    RELAT. TESTE
                    CONTRIBUICOES

          sobrenome       Contrib

* Cidade = Sao Paulo
Mila      Moreira           5.00
** SUBTOTAL **
                            5.00

* Cidade = Natal
Ana       Amorim            0.00
** SUBTOTAL **
                            0.00

* Cidade = Belem
Sonia     Simoes            0.00
** SUBTOTAL **
                            0.00

* Cidade = Curitiba
Monica    Medeiros          5.99
** SUBTOTAL **
                            5.99

* Cidade = Sao Paulo
Pedro     Penedo            0.00
** SUBTOTAL **
                            0.00
```

Agora vamos chamar o mesmo relatório depois de ativar o índice para SOCID.NDX, isto é, usando uma indexação no campo do subtotal:

```
. SET INDEX TO SOCID

. REPORT FORM CONTRIB

     PAGE NO. 00001

                                  RELAT. TESTE
                                  CONTRIBUICOES

              sobrenome         Contrib

     * Cidade = Aracaju
     Zilda     Zorzi              8.77
     Davi      Dunas              1.00
     ** SUBTOTAL **
                                  9.77
     * Cidade = Belem
     Sonia     Simoes             0.00
     Ires      Imare              7.50
     ** SUBTOTAL **
                                  7.50
     * Cidade = Curitiba
     Monica    Medeiros           5.99
     ** SUBTOTAL **
                                  5.99
     * Cidade = Natal
     Ana       Amorim             0.00
     ** SUBTOTAL **
                                  0.00
     * Cidade = Sao Paulo
     Mila      Moreira            5.00
     Pedro     Penedo             0.00
     Carlos    Covas              0.25
     Henrique  Hipolito          21.00
     ** SUBTOTAL **
                                 26.25

     ** TOTAL **
                                 49.51
```

SET DATE TO – Mudando a Data do Sistema

Quando inserimos o dBase em CP/M, o programa pede uma data. Em MS/DOS, o dBase procura a data no sistema operacional para buscá-la – normalmente está errada porque o computador não possui um calendário. Em qualquer caso, o dBase terminará com uma idéia, certa ou errada, de qual é a data.

A menos que a data seja escondida usando a opção "REPORT FORM PLAIN" ou anulada ("00/00/00" ou um <cr> em CP/M), ela aparecerá no topo de cada página de um relatório. Já que o relatório pode ter datas ocasionalmente diferentes da real, o comando SET DATE TO <nova data> pode ser usado para trocar a data do sistema.

```
. SET DATE TO 03/23/99

. REPORT FORM CONTRIB


        PAGE NO. 00001
        03/23/99

                              RELAT. TESTE
                              CONTRIBUICOES

                sobrenome   Contrib

        * Cidade = Aracaju
        Zilda   Zorzi          8.77
        Davi    Dunas          1.00
        ** SUBTOTAL **
                               9.77
```

E assim por diante.

SET HEADING TO – Um Cabeçalho Opcional para o Relatório

Além do cabeçalho de relatório definido durante o diálogo REPORT FORM, um segundo cabeçalho opcional pode ser incluído em cada página do relatório pelo comando "SET HEADING TO <string>":

```
. SET HEADING TO Pagamento dos Contribuintes

. REPORT FORM CONTRIB FOR PAGAMENTO
        PAGE NO. 00001      Pagamento dos Contribuintes
        03/23/99

                                   RELAT. TESTE
                                   CONTRIBUICOES

                 sobrenome       Contrib

        * Cidade = Aracaju
        Zilda      Zorzi             8.77
        Davi       Dunas             1.00
        ** SUBTOTAL **
                                     9.77
```

O resto do relatório é igual, exceto que a nova linha do cabeçalho aparecerá em cada página. Lembre-se de esvaziar a linha do cabeçalho depois de terminar o relatório, para que o nosso próximo relatório não possua o mesmo cabeçalho.

Para esvaziar a linha do cabeçalho, usamos o comando "SET HEADING TO" sem nenhuma string seguindo o "TO".

SET MARGIN – Especificações da Margem

Com o comando REPORT FORM ou qualquer outro comando de saída para impressão, o dBase permite ao usuário especificar a margem esquerda para o relatório na ocasião da impressão.

O comando é:

```
. SET MARGIN TO <n>   (*"n" é um número ou um espaço em branco *)
```

O valor do default de "n" para REPORT FORM é oito e um argumento em branco liberará a margem de volta para aquele valor. A margem, na verdade, pode ser ativada durante a definição do relatório com uma resposta de "M = n" para a primeira pergunta ou pode ser incluída na primeira linha em branco de um arquivo ".FRM", com um processador de texto.

SET EJECT OFF – Controle do Salto

Já que os relatórios normalmente são emitidos um após o outro, o dBase os mantém em suas próprias páginas iniciando cada um com uma alimentação de formulário. Desta maneira, a página que antecede o novo relatório é empurrada para fora da impressora.

Se *desejarmos* que o novo relatório inicie na mesma página que a anterior, poderemos esconder esta característica pelo comando:

```
. SET EJECT OFF
```

É lógico que ele pode ser trazido de volta.

Algumas Técnicas de Relatório

Antes de iniciarmos o Capítulo 5, vamos mencionar algumas técnicas do REPORT FORM que tornam estes relatórios mais versáteis. Lembre-se de que o REPORT FORM não se importa de onde vem o dado para preencher um campo no relatório. Todos os tipos de expressão podem ser usados nas definições daqueles campos e enquanto a saída do relatório refletirá estas trocas, os dados no arquivo permanecerão inalterados.

Consideremos as seguintes definições de campo (eles são campos de REPORT FORM):

```
COL        WIDTH,CONTENTS

001        10,PROD:NOME+'...'                 (*Incluindo reticências *)
ENTER HEADING: <Nome:Produto:--------         (*Grifando *)
002        1,'!'                              (*Separador vertical *)
ENTER HEADING: !;!                            (*Mais dois verticais *)
003        10,CUSTO*1.06                      (*Calculando taxas *)
ENTER HEADING: >Custo + Taxa;;--------        (*Grifo espaçado *)
004        1,'!'                              (*Outro separador *)
ENTER HEADING: !;!                            (*E mais verticais *)
```

Devemos usar a nossa imaginação em nossas definições de campo de relatório. Este é um comando excelente para definirmos saídas úteis rápida e facilmente. A única limitação é que o dado só pode ser visualizado em colunas, parecido com a saída de LIST. Se for necessário que nossa saída seja empilhada, como os endereços em etiquetas de correio, então um programa dBase deverá ser escrito.

```
REPORT FORM                          ETIQUETAS POSTAIS

(*linear*)                             (*empilhada*)

Nome,Ender..Cidade,Est.,CEP          Nome
                                     Endereco
                                     Cidade
                                     Estado - CEP
```

Isto conclui os exercícios do dBase no modo interativo. Neste ponto, o leitor que seguiu as seções, experimentando os comandos conforme sua seqüência, deve ter uma boa idéia sobre as sintaxes e capacidades dos comandos dBase.

CAPÍTULO

5

As Funções e Expressões do dBase

Neste capítulo, aprenderemos as expressões do dBase, subdivididas um tanto arbitrariamente em funções, variáveis de sistema, declarações booleanas e manipulações de string.

O QUE É UMA EXPRESSÃO?

A ciência da computação surgiu da matemática e vários termos foram emprestados. *Expressão* é um deles. Uma expressão poderia ser melhor definida como um símbolo, variável ou valor; ou ainda uma combinação significativa de símbolos, variáveis e valores.

Um ponto importante para lembrarmos é que: uma expressão pode necessitar ser resolvida antes de conhecermos o seu valor, mas sempre sabemos que ela representa um valor único e realizável. Podemos não saber antecipadamente o que a expressão "CUSTO*TAX" resultará em uma linha de comando dBase, mas estamos seguros de que ela possuirá um e somente um valor para cada registro.

Uma expressão pode ser um valor constante ou pode necessitar ser resolvida pelo dBase para que vejamos seu valor. Dependendo da expressão, ela pode transformar-se em qualquer um dos tipos de dados (caractere, numérico e lógico).

O dBase espera que usemos expressões em comandos para que seus tipos de dados não criem erros de sintaxe. Em outras palavras, se usarmos uma expressão que se transforme em um valor lógico a partir de um cálculo numérico, o dBase ficará chocado e manifestará todas as mensagens de erro usuais. No modo interativo, isto parece aborrecer-nos. Quando isto acontece dentro de um programa, chamamos de "furo". Os programas que furam são, entre outras coisas, comercialmente inviáveis.

Quando usamos uma expressão em um programa dBase, geralmente queremos colocar o valor obtido em algum lugar. Conseqüentemente os comandos STORE e REPLACE são

freqüentemente usados com expressões. Um dos comandos mais simples e versáteis que podem resolver expressões é o "?", que imprime o valor resolvido e que usaremos exclusivamente neste capítulo, já que ele mostra como resolver expressões sem nenhuma controvérsia.

EXPRESSÕES ARITMÉTICAS

O dBase só possui quatro operadores aritméticos, + (adição), – (subtração), * (multiplicação) e / (divisão).

Como explicamos no Capítulo 1, o dBase faz aritmética com números armazenados no formato de "ponto flutuante". Ele trunca suas respostas ao número de dígitos significativos indicados pelo usuário, pelos campos definidos ou pela entrada das casas decimais nos números envolvidos no cálculo:

```
. ? 5/2
  2               (*Dividindo dois inteiros ... *)
                  (*obtemos uma resposta de inteiros truncados.*)
. ? 5.0/2         (*A mesma divisão com uma casa decimal*)
  2.5             (*Esta é a melhor resposta*)
. ? 5.0/3.0       (*Uma resposta para dois decimais? *)
  1.6             (*Não*)
                  (*E este aqui? *)
. ? 5.00/3        (*Sim*)
  1.66
```

ORDEM DE PRECEDÊNCIA

Nós aprendemos na escola que as expressões algébricas possuem uma ordem de precedência em suas soluções – primeiro a multiplicação e divisão, depois a adição e subtração. As operações de igual precedência são resolvidas da esquerda para a direita. O dBase segue as mesmas convenções.

```
. ? 8/4+4     (*"8/8" ou "2 + 4"? *)
  6           (*A divisão primeiro – "2 + 4" *)
```

Novamente lembrando a álgebra escolar, quando queremos resolver as expressões em uma ordem diferente de operações do que a precedente, podemos forçar a ordem incluindo operações entre parênteses.

```
. ? 8/(4+4)     (*Isto significa "Faça a adição primeiro*)
  1             (*O mesmo que "8/8" *)
```

Os parênteses redundantes são perfeitamente aceitáveis e representam uma prática de programação que conduz à clareza de entendimento.

```
. ? (8/4)+4    (*Não há nenhuma dúvida aqui*)
  6
```

As expressões aritméticas podem utilizar variáveis de memória ou conteúdos de campos numéricos do arquivo de dados em uso.

```
. STORE 8 TO HARRY
  8

. ? HARRY/4
  2
```

AS FUNÇÕES DO dBASE

Uma **função** é uma expressão que transforma um valor. Em alguns casos, este valor transformado será um número ou uma string e em outros valores, ele poderia ser apenas um valor verdadeiro ou falso. As funções podem ser usadas para responder perguntas sobre o *status* de determinados dados, traduzir os dados de um tipo de dado para outro, ou ainda como condição para controlar a execução de outros comandos.

O dBase possui uma completa gama de funções, o que facilita a vida do programador. Algumas destas funções, como "fim-de-arquivo" e a data de sistema, são chamadas **variáveis do sistema**, já que elas possuem um valor dentro do dBase durante todo o tempo. Outras, como VAL( ) e INT( ) são consideradas **funções de transformação**, já que elas mudam um tipo de dado para outro. E ainda outras fazem chamadas para as funções do sistema operacional ("FILE( )") ou o controle direto do cursor ("@"). Com a exceção de DATE( ), todas as funções que são seguidas por parênteses esperam uma expressão dentro delas. Na maioria dos casos, o tipo de dado das expressões é crítico e podemos esperar um erro de sintaxe se a expressão for do tipo errado.

As funções estão listadas alfabeticamente para referência posterior.

@ – Controle de Endereçamento do Cursor

O símbolo **arroba** ("@") é usado para controlar o movimento do cursor de dentro dos programas dBase, uma tarefa de programação que é chamada **cursor endereçável.**

Ela sempre é associada aos comandos SAY ou GET. Os dois serão explicados extensivamente no Capítulo 7. Embora a arroba funcione no modo interativo, enviando o cursor para qualquer canto da tela, ela é quase sempre usada dentro dos programas. Sua sintaxe é:

```
@ y, x SAY/GET <expressão> (*O "y" e o "x" são números inteiros*)
```

Quando endereçamos o cursor com a arroba, a coordenada Y refere-se ao número de linha com a margem de 0 a 23 na tela. Devemos ficar fora da linha 0 porque o dBase a utiliza para comunicar-se com o usuário (notadamente a mensagem "INSERT").

A coordenada X refere-se ao número de coluna e sua margem é de 0 a 79 – o dBase esconderá automaticamente qualquer linha que aparecer fora deste limite. Quando usamos a arroba, devemos lembrar que a coordenada Y aparece primeiro e que o canto esquerdo mais alto da tela está na posição (0, 0).

O dBase permite que as coordenadas sejam expressões, como em:

```
@ (CONT*4), 12 SAY sobrenome
```

Se desejarmos simplesmente imprimir na próxima linha (abaixo de onde estava o cursor), o "Y" pode ser expresso como um sinal de cifrão:

```
@ 5,5 SAY "Ola"
@$,5 SAY "ou Oi" (*Estará na linha 6*)
```

Vamos fazer alguns exemplos:

```
. @ 11,5 SAY 'XYZ'   (*A string será impressa lá.*)
. @ 11,5 SAY 8/2    (*O valor 4 será impresso em cima de 'X'. *)
```

@ (<string1>, <string2>) – A Função da Posição Substring

A função de pesquisa de substring é o uso da arroba não relacionada a endereçamento do cursor e as duas funções não podem ser confundidas. A função @ (<string1>, <string2>) pesquisa a string 2 para a ocorrência da string 1. Se a função encontrar a primeira string na 2, fornece-nos a posição inicial dela dentro da string 2. Se a primeira string não for uma substring da segunda, ela retorna com 0.

```
. STORE "Sic Transit Gloria Mundi" TO string
Sic Transit Gloria Mundi

. ? @("Glor",string)
  13              (*Isto é, "Glor" inicia no caractere 13 em "string"*)

. ? @("Guaruja",string)
   0
                  (*Zero porque "Guarujá" não está na string*)
```

*** – A Marca de Apagamento como uma Variável de Sistema**

Como vimos na Seção 1, o dBase marca um registro para o apagamento colocando um asterisco em um espaço de um caractere reservado na estrutura do registro. O asterisco também pode ser considerado como uma variável do sistema que possui um valor verdadeiro/falso em qualquer momento durante a operação do dBase. Resumindo, o comando ".? *" significa: "O registro corrente está marcado para apagamento?"

Usando um subconjunto de SOCIOS, vamos apagar dois registros e manipulá-los usando a variável de sistema *:

```
. LIST NEXT 5

00001  Mila     Moreira    R.Elma, 111        Sao Paulo  SP 92109   5.00 .T.
00002  Ana      Amorim     R.Tres, 333        Natal      RN 10010   0.00 .T.
00003  Sonia    Simoes     R.Para, 831        Belem      PA 21105   0.00 .F.
00004  Monica   Medeiros   R. Diva, 2219      Curitiba   PR 44110   5.99 .T.
00005  Pedro    Penedo     R.Sao Luiz, 22     Sao Paulo  SP 92109   0.00 .F.

. DELETE FOR NOME = "M"
00002 DELETION(S)

. GOTO TOP

. LIST NEXT 5
00001 *Mila     Moreira    R.Elma, 111        Sao Paulo  SP 92109   5.00 .T.
00002  Ana      Amorim     R.Tres, 333        Natal      RN 10010   0.00 .T.
00003  Sonia    Simoes     R.Para, 831        Belem      PA 21105   0.00 .F.
00004 *Monica   Medeiros   R. Diva, 2219      Curitiba   PR 44110   5.99 .T.
00005  Pedro    Penedo     R.Sao Luiz, 22     Sao Paulo  SP 92109   0.00 .F.

. GOTO 1
```

```
. DISPLAY
00001 *Mila       Moreira        R.Elma, 111          Sao Paulo  SP 92109   5.00 .T.

. ? *           (*Esta pergunta só atinge o registro 1*)
.T.             (*Porque, na verdade, ele foi apagado*)
                (*Vamos tentar um registro diferente*)
. SKIP
RECORD: 00002

. DISPLAY
00002 Ana         Amorim         R.Tres, 333          Natal      RN 10010   0.00 .T.

. ? *           (*Ele foi apagado?" — sabemos que não*)
.F.             (*O dBase concorda*)

. LIST FOR * (*Usando o "*" como uma condição*)
00001 *Mila       Moreira        R.Elma, 111          Sao Paulo  SP 92109   5.00 .T.
00004 *Monica     Medeiros       R. Diva, 2219        Curitiba   PR 44110   5.99 .T.

. RECALL ALL
00002 RECALL(S)
```

Lembre-se de que o asterisco é um valor booleano ou lógico. Ele sempre possui um valor verdadeiro/falso, nunca um valor numérico ou de caractere. Quando não há nenhum arquivo de dados em uso, o valor do asterisco é falso.

— O Indicador do Registro como Variável do Sistema

Consideramos que o leitor já está familiarizado com o conceito de indicador de registro. Sempre penso em indicador de registro como um dedo que o dBase mantém parado no arquivo de dados, apontando para algum lugar. O indicador de registro em dBase é abreviado como um sinal numérico ("#").

De vez em quando desejamos saber exatamente qual o registro que o dBase possui no indicador, geralmente porque iremos abandonar aquele registro por um momento e necessitamos voltar para ele mais tarde. Nestas situações, o comando ".? #" poderia ser resumido como "Qual é o número do registro que o indicador está apontando?"

```
. GOTO TOP

. ? #
     1
```

```
. GOTO 6

. ? #
    6

. SKIP
RECORD: 00007


. ? #
    7
```

Isto parece simples quando o dBase volta o número do registro como resposta aos nossos SKIPs e GOTOs, mas durante a execução de programas a comunicação será desligada e muitos destes pedidos para o número de registro acontecerão sem que o operador esteja envolvido. Além disso, as posições TOPO e BASE são lógicas, não absolutas, e podem ser trocadas radicalmente do registro 00001 para o TOPO e a última entrada do registro para a BASE, quando um índice estiver em uso. Nestes casos, "? #" é uma pergunta muito razoável.

```
. SET INDEX TO SOCID
. GOTO TOP
. ? #

    6

  GOTO BOTTOM

. ? #

   10

. LIST
00006 Zilda    Zorzi     Rua Pilar, 44         Aracaju   SE        8.77 .T.
00008 Davi     Dunas     Rua das Flores, 123   Aracaju   SE 36222  1.00 .T.
00003 Sonia    Simoes    R.Para, 831           Belem     PA 21105  0.00 .F.
00009 Ires     Imare     R.Neusa, 99           Belem     PA 21233  7.50 .F.
00004 Monica   Medeiros  R. Diva, 2219         Curitiba  PR 44110  5.99 .T.
00002 Ana      Amorim    R.Tres, 333           Natal     RN 10010  0.00 .T.
00001 Mila     Moreira   R.Elma, 111           Sao Paulo SP 92109  5.00 .T.
00005 Pedro    Penedo    R.Sao Luiz, 22        Sao Paulo SP 92109  0.00 .F.
00007 Carlos   Covas     R. "A", 88            Sao Paulo SP 92101  0.25 .T.
00010 Henrique Hipolito  R.Doze, 21            Sao Paulo SP 92111 21.00 .T.
```

A função # pode ser usada em qualquer comando onde um número de registro estaria apropriado. Por exemplo, se quisermos editar o registro corrente, não importa como, ".EDIT #" realizará a tarefa.

Uma função importante do # é para nos dizer quando o comando FIND não funcionou. O dBase numera os registros a partir de 00001, uma contradição da prática de programação de começar com 0. O zero é, entretanto, um valor válido para # — o indicador de registro será igual a 0 sempre que um comando FIND retomar com NO FIND ou se nenhum arquivo de dados estiver em uso.

! (<string expression>) — A Função da Escrita em Maiúscula

O ponto de exclamação representa uma função da qual a operação serve para devolver uma string com todas as letras minúsculas escritas em maiúsculas. Nenhum outro caractere é afetado. Novamente, com SOCIOS:

```
. GOTO TOP

. DISPLAY
00006 Zilda     Zorzi          Rua Pilar, 44       Aracaju    SE          8.77 .T.

. ? SOBRENOME
Zorzi

. ? !(SOBRENOME)
ZORZI

. ? !('ABCabc123!#$')       (*Apenas três letras minúsculas*)
ABCABC123!#$                (*Todas transformadas em maiúsculas*)

. ? !(ENDERECO)             (*Outro exemplo*)
RUA PILAR, 44
```

Isto é prático. Quando colocamos coisas em ordem alfabética, automaticamente colocamos em nossa mente todas as letras em maiúsculas. Os computadores não são tão inteligentes — para o dBase, um "Z" é uma letra tão minúscula quanto um "a". Por exemplo, colocaremos em uso o índice SOCIDADE, que possui a expressão-chave CIDADE e, propositalmente, acrescentaremos um registro com a cidade em minúscula "aracaju".

```
. USE SOCIOS

. APPEND
```

```
RECORD  # 00011
NOME      :X          :
SOBRENOME :Y              :
ENDERECO  :Z                        :
CIDADE    :aracaju    :
ESTADO    : :
CEP       :     :
CONTRIB   : 0.00:
PAGAMENTO : :

. LIST
00006  Zilda     Zorzi       Rua Pilar, 44          Aracaju   SE          8.77 .T.
00008  Davi      Dunas       Rua das Flores, 123    Aracaju   SE 36222    1.00 .T.
00003  Sonia     Simoes      R.Para, 831            Belem     PA 21105    0.00 .F.
00009  Ires      Imare       R.Neusa, 99            Belem     PA 21233    7.50 .F.
00004  Monica    Medeiros    R. Diva, 2219          Curitiba  PR 44110    5.99 .T.
00002  Ana       Amorim      R.Tres, 333            Natal     RN 10010    0.00 .T.
00001  Mila      Moreira     R.Elma, 111            Sao Paulo SP 92109    5.00 .T.
00005  Pedro     Penedo      R.Sao Luiz, 22         Sao Paulo SP 92109    0.00 .F.
00007  Carlos    Covas       R. "A", 88             Sao Paulo SP 92101    0.25 .T.
00010  Henrique  Hipolito    R.Doze, 21             Sao Paulo SP 92111   21.00 .T.
00011  X         Y           Z                      aracaju               0.00 .F.
```

O problema aqui não é a habilidade do dBase em manter a indexação. O índice foi atualizado; porém, nossa expressão-chave original para a indexação foi ingênua. Se escrevermos um índice novo com letras maiúsculas para cidade, obteremos:

```
. INDEX ON !(CIDADE) TO TESTCID
00011 RECORDS INDEXED

. LIST
00006  Zilda     Zorzi       Rua Pilar, 44          Aracaju   SE          8.77 .T.
00008  Davi      Dunas       Rua das Flores, 123    Aracaju   SE 36222    1.00 .T.
00011  X         Y           Z                      aracaju               0.00 .F.
00003  Sonia     Simoes      R.Para, 831            Belem     PA 21105    0.00 .F.
00009  Ires      Imare       R.Neusa, 99            Belem     PA 21233    7.50 .F.
00004  Monica    Medeiros    R. Diva, 2219          Curitiba  PR 44110    5.99 .T.
00002  Ana       Amorim      R.Tres, 333            Natal     RN 10010    0.00 .T.
00001  Mila      Moreira     R.Elma, 111            Sao Paulo SP 92109    5.00 .T.
00005  Pedro     Penedo      R.Sao Luiz, 22         Sao Paulo SP 92109    0.00 .F.
00007  Carlos    Covas       R. "A", 88             Sao Paulo SP 92101    0.25 .T.
00010  Henrique  Hipolito    R.Doze, 21             Sao Paulo SP 92111   21.00 .T.
```

Agora o registro XYZ está no lugar apropriado.

Note que a função de letras maiúsculas usada na regra de indexação não tem efeito no campo CIDADE do registro.

$ (< expressão-string >, n1, n2) – A Função Substring

A função substring com o sinal ($) voltará uma parte de uma string ou campo iniciando no caractere "n1", caminhando por "n2" caracteres.

```
. USE SOCIOS

. DISPLAY
00001 Mila      Moreira       R.Elma, 111          Sao Paulo  SP 92109   5.00 .T.
*
. ? SOBRENOME
Moreira

. ? $(SOBRENOME,1,4)   (*Inicia no caractere 1 e produzirá 4 caracteres*)
More

. ? $(SOBRENOME,3,4)   (*Agora fornece 4 caracteres iniciando no caractere 3*)
reir
```

A função substring é utilizada para cortar strings como datas para que possam ser comparadas ou usadas em chaves de indexação. Um problema comum nos programas de computação, por exemplo, é que a data "25/12/84" é uma string "maior" para o computador do que "25/01/85". Isto significa que datas que sucedem a passagem dos anos serão classificadas.

Alguns programadores (até os em dBase!) resolvem este problema armazenando datas com ano/mês/dia, um exemplo perfeito de forçar as pessoas a adaptarem-se à máquina.

Em dBase, esta situação é melhor manipulada indexando uma expressão que repete o ano da string da data em frente à data:

```
INDEX ON $ (data, 7, 2) + data TO novadata
```

De fato, os campos de datas dentro do arquivo de dados permanecerão intocáveis, mas as duas datas no exemplo acima seriam indexadas com base na comparação de concatenação do ano na frente da data completa:

```
8525/01/85 > 8425/12/84
```

< string1 > $ (< string2 >) – A Função Substring Booleana

O que faríamos se tivéssemos um grande arquivo de dados de empresas-clientes e desejássemos localizar um registro para uma empresa que possuísse a palavra "Solar" em seu nome? Talvez fosse "Dimensões Solares" ou "Divisão Solar"; quem pode lembrar?

Neste caso, usaríamos uma função que voltaria com um valor falso/verdadeiro se uma string fosse uma substring de outra. Nosso comando para o problema acima seria:

```
LOCATE FOR "Solar" $ (CLI:NOME)
```

Isto seria relativamente demorado para executar, mas não há mais nada que resolveria nosso problema.

CHR(n) – Enviando ASCII Para a Saída

Ocasionalmente, um programador em dBase desejará enviar um caractere de controle de saída para o terminal, o cabo ou a impressora. O dBase permite isto através da função CHR(n), onde "n" é um inteiro de 0 a 127. Lembre-se de usar SET PRINT ON antes de usar esta função. Os exemplos mostrarão um caractere de saída visível e depois alguns caracteres de controle.

```
. ? CHR(65)          (*ASCII para a maiúscula A*)
A
.
. ? CHR(7)           (* ^G, o caractere de controle "BELL"*)

(*Toca o alarme*)
. ? CHR(15)          (*^O, inverte o vídeo para a maioria dos CP/M*)

(*A tela fica branca*)
. ? CHR(14)          (* ^N, inverte de volta (CP/M*)
(*A tela volta ao normal*)
```

DATE( ) – A Data do Sistema

A data do sistema é a data que digitamos quando entramos com o dBase (CP/M), ou a data que o dBase copia do selo da data no sistema operacional (MS-DOS). Como já dissemos, isto pode ou não refletir a data no mundo real, dependendo do que nós ou o sistema operacional dissemos ao dBase. Além disto, DATE( ) pode ser trocado arbitrariamente usando "SET DATE TO", sem se preocupar com o que esteja gravado lá.

```
. ? DATE()
00/00/00
```

Mas DATE( ) é uma variável de sistema disponível. É utilizável em qualquer lugar de dentro de um programa e pode ser usada para preencher campos automaticamente, como em:

```
. REPLACE novo:dia WITH date()
```

Os "parênteses" no símbolo da data de sistema servem para nos lembrar de que ela é uma função e os mantém para não confundir com nenhum campo que possamos nomear de "DATE".

O DATE( ) volta com uma string de oito caracteres que consiste em "mm/dd/aa" ou "dd/mm/aa", dependendo da instalação. A string pode ser manipulada pela função da substring — "$ (date( ), 7, 2)", por exemplo, volta o ano — e o dBase automaticamente troca qualquer delimitador de entrada pela ' / ' (barra):

```
. SET DATE TO 03,23,99      (*Não há parênteses no comando SET*)

. ? DATE()
03/23/99

. ? $(DATE(),1,2)
03
```

EOF – A Função Booleana de End-of-File*

Em quase todas as operações de longos programas dBase, desenvolveremos alguns processos até que o final do arquivo de dados seja alcançado. Certamente, se o fim-de-arquivo foi alcançado, não desejaremos continuar o processo, mesmo que não tenha sido nossa intenção ir tão longe!

Teremos em mãos uma variável de sistema chamada "EOF", lida como "Fim-de-Arquivo". É essencial lembrarmos que EOF tenha um valor booleano, verdadeiro/falso, e que ele sempre possui um valor ou o outro.

EOF é quase sempre falso, já que apenas um processo pode transformá-lo para verdadeiro. EOF torna-se verdadeiro somente quando há uma tentativa para pular o último registro no arquivo com um SKIP ou um LIST. Logo que tentamos o salto, EOF transforma-se em verdadeiro e o indicador volta para o último registro. Qualquer movimento posterior no arquivo de dados (outra tentativa para saltar além da base) transformará EOF em falso.

* N.T. – End-of-File – Fim-de-Arquivo, em português a abreviação modificaria a função.

```
. USE SOCIOS

. GOTO 11
. DELETE
00001 DELETION(S)
. PACK
PACK COMPLETE, 00010 RECORDS COPIED

. LIST
00001  Mila       Moreira    R.Elma, 111          Sao Paulo  SP 92109   5.00 .T.
00002  Ana        Amorim     R.Tres, 333          Natal      RN 10010   0.00 .T.
00003  Sonia      Simoes     R.Para, 831          Belem      PA 21105   0.00 .F.
00004  Monica     Medeiros   R. Diva, 2219        Curitiba   PR 44110   5.99 .T.
00005  Pedro      Penedo     R.Sao Luiz, 22       Sao Paulo  SP 92109   0.00 .F.
00006  Zilda      Zorzi      Rua Pilar, 44        Aracaju    SE         8.77 .T.
00007  Carlos     Covas      R. "A", 88           Sao Paulo  SP 92101   0.25 .T.
00008  Davi       Dunas      Rua das Flores, 123  Aracaju    SE 36222   1.00 .T.
00009  Ires       Imare      R.Neusa, 99          Belem      PA 21233   7.50 .F.
00010  Henrique   Hipolito   R.Doze, 21           Sao Paulo  SP 92111  21.00 .T.

. DISP    (*LIST nos deixou na base*)
00010  Henrique   Hipolito   R.Doze, 21           Sao Paulo  SP 92111  21.00 .T.

. ? EOF    (*Sim — LIST tenta saltar para o final*)
.T.

. GOTO TOP   (*Uma pequena viagem . . .*)
. ? EOF
.F.          (*. . . e o EOF transforma-se em falso*)

. GOTO 10   (*Volta para a base, mas sem um salto*)

. ? EOF
.F.       (*Não é verdadeiro até o salto*)

. SKIP    (*Tentaremos saltar passando o registro 10*)
RECORD: 00010

. ? #
    10    (*De fato, não podemos*)

. ? EOF
.T.       (*Verdadeiro depois do salto*)
```

FILE("< nome do arquivo >") – Há um Arquivo?

Precisamos encontrar uma maneira para checar a existência de um arquivo nos drives de disco de dentro dos programas. Torna-se embaraçoso quando tentamos abrir um arquivo de dados que não existe e pode ser uma economia de tempo checar o diretório antes de fazê-lo. O dBase possui o FILE( ), que checa pelo nome do arquivo e volta com valor booleano verdadeiro/falso.

```
. ? FILE("SOCIOS.DBF")    (*Note as aspas*)
.T.

. ? FILE("DINGBAT.DBF")
.F.

. ? FILE("A:WS.COM")    (*Podemos especificar um disco*)
.T.

. ? FILE("A:*.COM")    (*Mas não podemos esperar que os coringas funcionem*)
.F.    (*Resposta errada – existem arquivos ".COM" aqui*)
```

Como com as outras expressões, o nome de arquivo em FILE( ) pode ser preenchido com qualquer string de expressão válida de qualquer fonte, incluindo macros como em ? FILE ("&mem: arqnom").

INT(< expressão numérica >) – Números Reais para Inteiros (Truncados)

A função de números inteiros pega qualquer número real (número com uma fração) dado e descarta a fração. Obteremos um número inteiro.

```
. ? INT(12.333)
 12

. ? INT(12.333/2)
   6
```

O INT só funciona em entradas numéricas – não podemos esperar que ele corte a parte decimal de uma string numérica, porque ela representa um tipo de dado errado:

```
. STORE "12.333" TO NUM:STR
12.333

. ? INT(NUM:STR)
```

```
*** SYNTAX ERROR ***
  ?
? INT(NUM:STR)
CORRECT AND RETRY (Y/N)? N
```

O INT é manipulável algumas vezes em matemática comercial quando queremos obter o número exato de centavos de um número de ponto flutuante que tenha sido arredondado.

```
. ? INT(CUSTO:FIN*100)+0.5)/100
```

O processo de multiplicar por 100, truncar e dividir por 100 apagando as casas decimais além de duas. Somar 0.5 arredonda.

LEN(< string >) – A Função de Extensão

O fato de conhecermos a extensão de uma string, uma variável de string de caracteres ou um campo de arquivo de dados pode ser valioso. A função LEN( ) voltará com um inteiro representando aquela extensão:

```
. DISPLAY
00010  Henrique   Hipolito       R.Doze, 21          Sao Paulo  SP 92111  21.00 .T.
.
. ? LEN(SOBRENOME)
  15

. STORE "Hipolito" TO mSOBR    (*"m" para indicar uma varmem*)
Hipolito

. ? LEN(mSOBR)
   8                  (*Oito caracteres menor que o campo*)

. ? LEN(mSOBR)*2      (*LEN ( ) volta para numérico*)
   16                 (*que pode ser manipulado aritmeticamente*)
```

RANK(< expressão de caractere >) – A Seqüência de Intercalação

Assim como CHR(n) fornece um caractere que corresponde a um inteiro, RANK("x") retorna um inteiro correspondente a um caractere. A conversão é a mesma, de caracteres ASCII para seus equivalentes básicos em decimal:

```
. ? RANK("A")
  65                (*"A" é o 65º no código ASCII*)

. ? RANK("a")
  97                (*As letras minúsculas em ASCII são maiúsculas + 32*)
```

Se RANK( ) for uma expressão de string, ele voltará com o inteiro ASCII do primeiro caractere naquela string:

```
. DISPLAY
00010 Henrique   Hipolito       R.Doze, 21          Sao Paulo  SP 92111  21.00 .T.

. ? RANK(SOBRENOME)
  72    (*ASCII de "H" *)
```

STR(< número >, < extensão >, [< decimais >]) – O Número para a String

As funções-padrão em todas as linguagens de computação sempre incluem aquelas que transformam as strings e números entre si. Surgem as situações quando tais conversões devem ser feitas.

STR(< expressão >, n1, n2) transformará em string o número da expressão, ajustando-o à direita no campo de extensão "n1", com "n2" casas decimais. Se n1 não for extenso o bastante para a string transformada, o dBase fará o possível e incluirá asteriscos no campo para mostrar uma condição de erro. N2 é opcional, e o seu default é nenhuma casa decimal.

```
. ? CONTRIB
 21.00              (*Aqui, é um número*)

. ? STR(CONTRIB,5,2)
21.00               (*Aqui, é uma string*)

. ? STR(CONTRIB,5)
   21               (*O default sem decimais*)
```

A string transformada pode ser usada em campos de caracteres concatenados; porém, ela não pode ser usada em aritmética:

```
. ? STR(CONTRIB,5,2)/2                  (*Um pequeno cálculo, por favor*)

*** SYNTAX ERROR ***                    (*Não! agora é uma string*)
                  ?
? STR(CONTRIB,5,2)/2
CORRECT AND RETRY (Y/N)? N
```

Porém, um cálculo aritmético pode ser feito no número antes que se transforme em uma função STR( ):

```
. ? STR(CONTRIB/3,5,2)
 7.00
```

TEST(< expressão >) – Uma Análise para Checar a Sintaxe

Normalmente nossos programas pedirão para que o usuário digite a expressão de um comando, que será ligado a ele sob a forma de uma macro. Se o usuário digitar uma declaração tal como:

```
CIDADE = "Belem"
```

então o programa funcionará; porém, ele não vai gostar se o usuário digitar:

```
CIDADE is "Belem"
```

A função TEST( ) permite ao programador proteger seus programas contra esta segunda possibilidade. Ele volta com um valor 0 se a expressão testada não for "analisável" – reconhecível – pelo interpretador do dBase e com 1 se estiver correta:

```
ACCEPT "Digite a expressão-teste" TO Mtest
IF TEST (Mtest) = 0
      *"errata .cmd" é a rotina de erro
      DO errata
ELSE
      *não faça nada – continue
ENDIF
```

TRIM(< expressão string >) – Eliminando os Brancos

Como discutiremos na seção sobre manipulação de strings, o dBase nos fornece vários instrumentos para cortar as strings quando elas operam fora de caracteres. Uma delas é TRIM( ), que elimina os espaços em branco à direita da string.

```
. ? NOME,SOBRENOME          (*Três espaços porque SOBRENOME tem 10 caracteres*)
Henrique   Hipolito

. ? TRIM(NOME),SOBRENOME
Henrique Hipolito           (*Um espaço porque a vírgula está no comando*)

. ? CIDADE,ESTADO
Sao Paulo  SP               (*Novamente, com o campo CIDADE completo*)

. ? TRIM(CIDADE),ESTADO
Sao Paulo SP                (*Um espaço para a vírgula*)
```

Note que TRIM( ) elimina *todos* os espaços e os espaços entre os campos acima são criados pela presença da vírgula no comando. Na seção sobre a manipulação de string, veremos as maneiras de eliminá-los quando necessário.

TYPE(< expressão >) – Descobrindo o Tipo de Dado

Já que o tipo de dado é tão importante, parece razoável que o dBase nos forneça uma maneira de checar o tipo de uma variável antes de colocá-la em uma operação. Assim, podemos prevenir furos e mensagens como "****SINTAX ERROR****".

A função TYPE( ) voltará com um caractere representando o tipo de dado da expressão mencionada. Se o tipo de dado não estiver definido porque o campo ou a variável de memória não existe, TYPE( ) voltará com um "U". Aqui estão alguns dos tipos de dados de SOCIOS.

```
. ?TYPE(F:NOME)
```

```
. ? TYPE(NOME)
C               (*Dado caractere, apropriado para nomes*)

. ? TYPE(CONTRIB)
N               (*... e ele era um número*)

. ? TYPE(PAGAMENTO)
L               (*... um lógico — tudo correto*)

. ? TYPE(FONY)
U               (*Não há "FONY", então é indefinido*)
```

Lembre-se de que o valor do tipo de dado retornado é um caractere e pode ser tratado como tal:

```
. ? TYPE(TYPE(PAGAMENTO))
C
```

Como nas outras funções, TYPE( ) pode ser usado com uma expressão

```
. ? TYPE (VAL(" "))     (*Qual é o tipo da função VAL( )? *)
N                       (*VAL( ) retorna um número, é claro*)
```

VAL(< expressão string >) – String para Numéricos

VAL( ) é o operador oposto de STR( ). Ele é usado para extrair um valor numérico de uma string que contém dígitos. Nunca teremos de dividir um código postal por dois, mas os programadores logo aprendem que "nunca" é uma palavra muito definitiva.

VAL( ) lê uma string a partir da esquerda, ignorando os brancos e aceitando os sinais de mais ou menos. A função lerá os dígitos incluindo um ponto decimal, continuando até alcançar um espaço ou um caractere alfa. VAL( ) retorna com 0 se encontrar um caractere não numérico antes de um numérico 0.

```
. STORE "138 paginas revisadas" to string3
138 paginas revisadas

. ? VAL(string3)
           138     (*Até o espaço*)


. DISPLAY

00010 Henrique  Hipolito     R.Doze, 21        Sao Paulo  SP 92111  21.00 .T.
. ? VAL(CEP)/3              (*Nunca diga nunca*)
 30703

. ? VAL(NOME)
     0
```

Manipulação de String – Os Operadores de Concatenação

As **strings** são coleções de caracteres que formam uma unidade de dados que pode ser manipulada como um item. Em SOCIOS, tanto NOME e CIDADE são strings diferentes em cada registro.

Além das funções de string tal como TRIM( ) e LEN( ), o dBase possui vários operadores reservados especificamente para a manipulação de strings. Existem apenas três operações básicas de strings: as strings podem ser analisadas em substrings, ajuntadas nos finais (concatenadas) e também podem ser comparadas.

A análise de string em dBase é completamente possível, mas geralmente tem de ser executada usando pequenos programas ou segmentos de programas. Estes programas normalmente examinam caractere por caractere de uma string usando a função substring, até que encontre algum caractere-chave. A string pesquisada é então separada naquele ponto. Alguns exemplos de análise aparecem nas seções de programação.

O dBase possui dois operadores de concatenação para a união de strings no final, os operadores + e –. Eles funcionam como segue:

1. O concatenador "+":

```
|ABC          |   +  |XYZ   |  (*Duas strings*)
<10 carac>            <-12 carac ->
|ABC          XYX              |  (*Uma string*)
      <-22 carac ->
```

2. O concatenador "–":

```
| ABC         |   –  |XYZ   |  (*Duas strings*)
<10 carac>          <-12 carac ->
|ABCXYZ                       |  *Uma string*)
      < -22 carac ->
```

Note que nos dois casos, a string resultante é igual em extensão à soma das duas strings. O efeito do concatenador "–" não é para eliminar os brancos da string resultante (embora TRIM( ) possa ser usado na string resultante), mas para mover o conteúdo da string 2 para a esquerda através dos espaços da string 1.

Vamos ver um exemplo:

```
. GOTO TOP
. DISPLAY
00001  Mila      Moreira        R.Elma, 111          Sao Paulo  SP 92109   5.00 .T.

. STORE NOME TO mNOME
Mila

. STORE SOBRENOME TO mSOBR
Moreira
```

```
. ? mNOME + mSOBR
Mila      Moreira              (*Soma do campo — entre dois espaços*)

. ? LEN(mNOME+mSOBR)
  25                           (*O dBase nos diz o comprimento*)

. ? mNOME-mSOBR
MilaMoreira                    (*Apertado*)

. ? LEN(mNOME-mSOBR)
  25                           (*... mas o mesmo comprimento*)


. ? TRIM(mNOME)+mSOBR
MilaMoreira                    (*A mesma aparência*)

. ? LEN(TRIM(mNOME)+mSOBR)
  19                           (*... sem nenhum espaço*)
.
. ? TRIM(mNOME),mSOBR
Mila Moreira                   (*Temos um espaço entre duas strings separadas*)
```

Na próxima seção trataremos sobre as comparações de string.

Operadores Booleanos – Verdadeiro e Lógico nas Cláusulas FOR

Até este ponto, apresentamos cláusulas de condição em muitas das demonstrações feitas dos comandos sem explicarmos como elas funcionam.

Já vimos esta forma:

```
. <faça alguma coisa – verbo dBase> FOR <esta ou aquela – condição>
```

como em:

```
. LIST FOR CIDADE = "Sao Paulo"
```

Obviamente, o truque aqui é tudo que segue o "FOR".

Esta parte de um comando dBase, chamada condição, deve ser uma declaração booleana sintaticamente correta; isto é, ela deve ser uma declaração que o dBase possa avaliar como tendo um valor verdadeiro ou falso para cada registro. Um valor "talvez" não funcionará.

Isto é óbvio. Consideremos este comando incorreto:

```
. LIST FOR "A"          (*Não funcionará*)
```

O que listaríamos se usássemos este comando? Podemos tomar o primeiro registro e observá-lo, mas já que não existe nenhuma maneira de avaliar qual registro é o "A" (?), o processo não vai além disto.

Isto nos leva a um princípio das condições do dBase: a declaração que segue o FOR em um comando dBase deve ser resolvida em um valor verdadeiro ou falso para cada registro considerado.

Tanto quanto parece simples, existem algumas dicas sobre os erros comuns que ocorrem quando o princípio é violado. Mais freqüentemente, os problemas surgem por causa das misturas de tipos de dados em declarações booleanas:

```
. LIST FOR PAGAMENTO = T     (*Não funcionará – PAGAMENTO já é T/F*)

. LIST FOR CEP = 92109       (*Não, CEP é um campo caractere.*)
```

Poderíamos dar mais exemplos, mas a idéia deve estar clara. Não podemos comparar números com strings, lógicos com caracteres, nem bananas com laranjas.

Comparadores Booleanos – Transformando Strings ou Números em Verdadeiro/Falso

Consideremos a declaração "1 = 2". O que podemos dizer sobre ela? Apenas uma coisa, um não é igual a dois, assim podemos dizer que ela está errada. Mas ela não tem sentido? Não em computação. Uma declaração como esta é perfeitamente válida como uma condição do dBase, já que carrega um valor booleano legítimo-falso. O comando ".LIST FOR 1 = 2" nos forneceria uma listagem em branco, qualquer que seja o arquivo de dados, mas ele não nos forneceria uma mensagem de "erro de sintaxe".

Apesar de as cláusulas FOR somente aceitarem argumentos lógicos ou booleanos, pelo uso criterioso dos "operadores lógicos", que testa a relação entre expressões diferentes, podemos fazer uma declaração verdadeira/falsa de qualquer tipo de dado. A seguinte declaração sempre possui um valor sintaticamente correto de verdadeiro ou falso:

```
<item 1 do tipo de dado X>=<item 2 do tipo de dado X> (*.T. ou .F.*)
```

A próxima declaração nunca está sintaticamente correta:

```
<item 1 do tipo de dado X >=<item 2 do tipo de dado Y >  (*Choque de tipo*)
```

Quando as seguintes funções são usadas como comparadores nas declarações dBase, resultam em uma expressão lógica.

```
 =   : igual
       1 = 1 (.T.); "A" = "a" (.F.)

 >   : maior que
       "a" > "A" (.T.); 3 > 5 (.F.)

 <   : menor que
       21/7 < 4 (.T.); "Zorzi" < "Zorzi" (.F.)

>=   : maior que ou igual a
       1.5 >= 0 (.T.); 3 >= 3 (também .T.)

<=   : menor que ou igual a
       "A" <= "a" (.T.); "a" <= (também .T.)

<>   ou # : não igual a
       1.000001 <> 1 (.T.) ; 3 <> 3 (.F.)
```

Com esta explicação a mais, o comando

```
. LIST FOR CIDADE = "Sao Paulo"
```

pode ser entendido melhor. O campo "CIDADE" foi definido em SOCIOS como um campo de caractere e as aspas em "Sao Paulo" define-o como uma string. Portanto, estamos comparando uma string a uma string e a condição terá um valor booleano para cada registro.

Os Operadores Lógicos – Misturando Tipos de Dados em Declarações Booleanas

O que faremos se a condição para a nossa seleção (ou SORT ou APPEND) for muito complicada para expressar numa simples comparação? Podemos, por exemplo, querer todos os registros do arquivo *exceto* os São Paulos, ou poderíamos querer os registros de sócios de Belém que possuam contribuições maiores que $1,00.

O dBase possui três operadores lógicos que podem ser usados em conjunto com os comparadores, as funções e os valores lógicos para construir condições longas e expressivas para os comandos. Eles são as palavras: .NOT., .AND., e .OR., acompanhadas de pontos para que o dBase (e o programador) saiba que elas não estão sendo usadas como nomes de variáveis.

```
. LIST FOR .NOT. CIDADE = "Sao Paulo"          (*Ou CIDADE <> "Sao Paulo"*)

. LIST FOR CIDADE = "Curitiba" .AND. CONTRIB > 1
```

Como os operadores matemáticos, estes operadores lógicos possuem uma ordem de procedência pela qual eles são resolvidos.

Para o registro, é:

.NOT.
.AND.
.OR.
Funções booleanas

Dissemos "para o registro" porque, como com os operadores matemáticos, os operadores lógicos podem e são agrupados com parênteses para controlar suas ordens de evolução. Adivinhe por quê?

```
.NOT. CIDADE = "Aracaju" .AND. CONTRIB > 0  (*O que significa?*)
(.NOT. CIDADE = "Aracaju") .AND. (CONTRIB > 0)  (*O mesmo, porém melhor*)
```

Os operadores booleanos possuem os seguintes efeitos:

1. .NOT. inverte o valor verdadeiro de seu operando.

2. .AND. transforma as declarações combinadas em verdadeiras se e somente se as duas forem verdadeiras. A Figura 5-1 mostra a tabela verdadeira para "A.AND.B".

3. .OR. transforma a declaração em verdadeira se qualquer dos componentes for verdadeiro. A declaração completa será avaliada antes que o valor seja retornado. A Figura 5-2 mostra a tabela verdadeira para "A.OR.B".

4. O dBase não possui .XOR. ("ou exclusivo") por si, porém, o equivalente lógico pode ser construído facilmente usando .AND. e .OR.. A forma .XOR. pode ser simplificada através de redução, mas possui a vantagem da clareza. A tabela de verdadeiro para "(A.OR.B) .AND. (.NOT. (A.AND.B))", que é o equivalente a .XOR., é mostrada na Figura 5-3.

Em resumo, os operadores lógicos .NOT., .AND. e .OR. só podem ser usados para combinar as declarações booleanas e expressões booleanas mais complexas. Outros tipos de dados podem ser combinados para formar as declarações booleanas usando os comparadores lógicos para comparar dois valores do mesmo tipo de dado.

Os programadores mais avançados podem notar que o falso é designado ao valor numérico 0 e o verdadeiro é para o valor de qualquer número que não seja zero. Assim "LIST FOR 0" está sintaticamente correto (não listamos nada, naturalmente) como "LIST FOR 23/6*0.01" (tudo seria listado).

A: \ B:	V	F
V	VERDADEIRO	FALSO
F	FALSO	FALSO

Figura 5-1 A Tabela para "A .AND. B".

A: \ B:	V	F
V	VERDADEIRO	VERDADEIRO
F	VERDADEIRO	FALSO

Figura 5-2 A Tabela para "A .OR. B". .

A: \ B:	V	F
V	FALSO	VERDADEIRO
F	VERDADEIRO	FALSO

Figura 5-3 Tabela verdadeira para (A. OR. B) .AND (.NOT. (A .AND. B)), o equivalente lógico de .XOR. do dBase.

CONCLUSÃO

Este capítulo completa as colocações sobre comandos, funções, variáveis de sistema, operadores matemáticos e operadores booleanos do dBase. Se o dBase não oferecesse mais nada, ainda assim seria um instrumento poderoso para o gerenciamento, manipulação e o armazenamento de dados. O dBase no modo interativo é para o dado o que um processador de texto é para o texto: fornece-nos controle direto, rápido e quase milagroso de volumes de dados que seriam impossíveis de manipular.

Mas o dBase tem muito mais. Na segunda metade deste livro, aprenderemos os comandos, controle de estruturas, e técnicas para usar o dBase como uma linguagem de programação tão poderosa quanto Pascal ou Fortran. E mais facilmente.

CAPÍTULO

6

O Controle das Estruturas

No mais simples conceito, programar significa projetar uma seqüência ordenada de operações pela qual o computador pode fazer um trabalho útil e gravar esta seqüência para que possa ser usada quando desejarmos. Com esta visão, não há mágica para programar. Precisamos conhecer primeiro, uma maneira para fazermos o nosso computador trabalhar por etapas, e em segundo, uma maneira para gravarmos estas etapas.

Neste capítulo, apresentaremos as idéias fundamentais da programação dBase. Aprenderemos como o interpretador do dBase funciona e obteremos uma imagem melhor da estrutura lógica interna. Discutiremos a construção de blocos fundamentais dos programas – arquivos de comandos. Todos os controles de estruturas que governam os arquivos de comandos serão apresentados. Finalmente, colocaremos algumas boas práticas de programação e algumas de minhas filosofias sobre o desenho de programas.

COMPILADORES E INTERPRETADORES

Na maioria das linguagens de programação, o programador grava sua criação na forma de um arquivo de texto, escritas com um processador de texto ou outros programas de edição. Cada linha do arquivo de texto representa um ou mais passos que o computador executará.

Quando o programa está desenvolvido, o arquivo de texto é tratado por um outro programa chamado **compilador** ou **assembler**, que traduz o código textual para um arquivo de linguagem de máquina que o computador possa executar. O desafio para o programador é escrever um programa de arquivo de texto – chamado **código-fonte** – que irá funcionar, na verdade, quando for compilado.

Há um sistema alternativo para as linguagens de computador, porém, que não traduz o código-fonte em um arquivo executável. Neste método alternativo, um programa chamado

interpretador foi desenvolvido para ler linha por linha do código-fonte e executar cada linha enquanto vai lendo. Muitas formas do BASIC funcionam desta maneira e o dBase II também.

Os compiladores e assemblers possuem algumas vantagens em relação aos interpretadores. Uma vez que o compilador tenha trabalhado, não precisamos mantê-lo na memória para operar o programa, nem mantê-lo no disco, o que economiza espaço. Além disto, é natureza dos compiladores (um pouco) e assemblers (muito) escrever programas mais rápidos e compactos.

Por outro lado, em uma linguagem interpretada, se uma linha de código está num circuito e tem de ser executada 100 vezes, o interpretador tem de analisar 100 vezes, o que torna consideravelmente mais lenta a execução. Além disso, o interpretador e o programa devem estar presentes ao mesmo tempo para funcionar, o que pode causar problemas de espaço.

Não queremos dizer que os interpretadores não possuam vantagens. Quando um compilador realizou seu trabalho, só podemos usá-lo depois de escrevermos outro programa. Sempre que um interpretador vê uma linha de código, ele é capaz de ler e decidir o que precisa ser feito.

Para nós, assim como para os programadores em dBase, isto significa que podemos armazenar linhas de código (ou qualquer outra coisa a mais) na memória na ocasião da operação e então alimentá-las para o interpretador como macros. Não importa o que estas linhas de código possam dizer durante a operação, sabemos quando escrevemos o programa que podemos confiar no interpretador para executá-las propriamente.

A ORGANIZAÇÃO DO dBASE

Na Seção 6 do modo interativo, aprendemos a controlar a memória do dBase e o arquivo de dados digitando os comandos a partir do prompt. Quando digitávamos os comandos e pressionávamos return, era o interpretador do dBase que aceitava os comandos, decifrava seus significados e realizava o trabalho real.

Com isto em mente, podemos agora imaginar a maneira que o dBase é organizado – um diagrama lógico, ou um "mapa auxiliar", do interior de um computador como é usado pelo dBase (veja a Figura 6-1).

Sabemos que uma área de memória é um buffer de um arquivo de dados no qual os registros do arquivo são carregados. Por enquanto, não complicaremos nossa imagem com nenhum pensamento de como uma indexação funciona – é o bastante ver o arquivo de dados como uma longa série de registros, sendo alimentada do disco através do buffer. Uma seta no buffer, chamada indicador de registro, indica o único registro que seria mostrado na tela se fizéssemos um DISPLAY. Este buffer de arquivo de dados é chamado **área primária**.

Quando fechamos o arquivo com outro USE, um QUIT ou um CLEAR, sabemos que esta série de registros é completamente devolvida ao disco antes que a área primária seja fechada. Colocaremos o interpretador em cima, do lado esquerdo da área primária. Sempre imagino o interpretador como uma caixa preta com uma manivela ao lado. Colocamos um comando dentro do interpretador, viramos a manivela e os resultados surgem.

Acima da área primária, fica a memória do dBase. Aqui podemos colocar até 64 itens de dados de qualquer tipo, manipulá-los, gravá-los no disco ou fazer qualquer uma das outras operações da Seção 5. Este é o nosso bloco de rascunhos. À direita da área primária nesta figura

aparece um novo recurso do dBase e que não usaremos até o Capítulo 8 – a "área secundária". Por enquanto, é suficiente saber que o dBase permite abrir, não apenas um, mas dois arquivos de dados por vez, completos e com indexações.

Os comandos que manipulam o arquivo de dados na área secundária são, não surpreendentemente, exatamente os mesmos que os da área primária, assim tudo que aprendemos podemos aplicar. Por enquanto, podemos esquecer a área secundária.

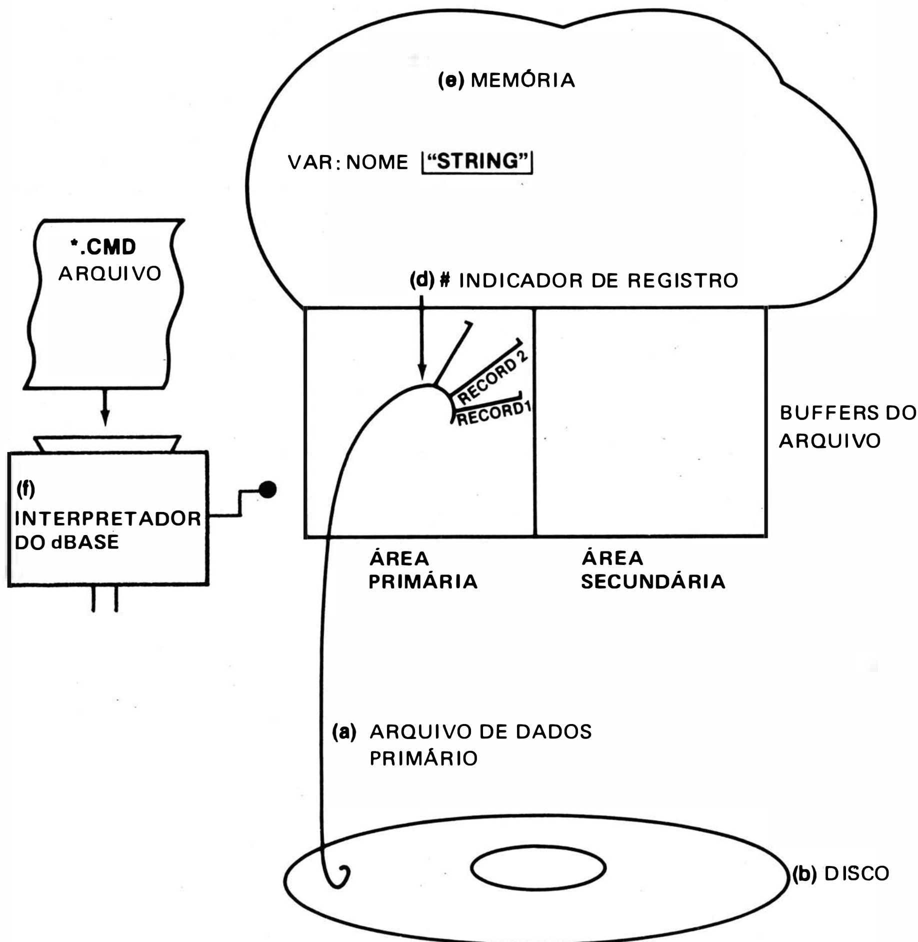


Figura 6-1 A distribuição lógica da organização do dBase. Um arquivo de dados (a) é carregado do disco (b) e montado na "área primária" da memória (c). Mostramos também os indicadores de registro (d), a área da memória reservada para variáveis (e) e o interpretador do dBase (f).

ARQUIVOS DE COMANDOS

A maioria das programações dBase é feita com os comandos já aprendidos nos cinco primeiros capítulos, acoplados com alguns novos comandos que direcionam suas execuções. Como outra linguagem de computação, o programador compõe o programa escrevendo os comandos em um arquivo de texto, usando um processador de texto ou outro editor. Estes arquivos são então gravados em um disco como **arquivos de comandos** dBase.

Estes arquivos de comandos podem então ser **chamados** ou levados a uma execução, a partir do prompt do dBase pelo verbo "DO". Por exemplo, se tivéssemos escrito um arquivo de comandos chamado GRANDE.XYZ, poderíamos conduzir o dBase a abrir o arquivo e executar seus comandos um por um dizendo:

```
. DO GRANDE.XYZ <cr>
```

Visto que o tipo de arquivo ".XYZ" não nos diz muita coisa, o dBase possui para um tipo especial de arquivo para arquivos de comandos. Originariamente no sistema operacional CP/M, o tipo era ".CMD". Porém, quando o MS-DOS foi escrito, o tipo de arquivo foi trocado (causando uma completa e desnecessária inconsistência entre os dois sistemas operacionais), e assim no IBM e outros computadores de MS-DOS, os programas dBase terminam com ".PRG". Aqui iremos aderir ao final .CMD.

Como é normal em dBase, se usarmos o tipo default para o arquivo, não precisaremos escrevê-lo nas linhas de comando. Um programa chamado GRANDE.CMD poderia ser chamado com o comando:

```
. DO GRANDE <cr>
```

Quando o interpretador do dBase recebe um comando DO, ele procura o arquivo de comandos no drive default (a menos que, é lógico, o DO tenha indicado outro drive como em "DO C:JOB"). Se ele não puder encontrar o arquivo, envia uma mensagem de erro "DO CANCELED" e volta ao prompt. Se ele encontrar o arquivo de comandos, abre-o e inicia a alimentação das linhas de comandos para o interpretador, um de cada vez.

Cada linha de comandos é executada precisamente como se o usuário a tivesse digitado. Assumindo que não foi detectado nenhum erro, quando a última linha for executada o dBase voltará ao prompt. Já que as linhas de comando são gravadas num arquivo de texto, é permitido ao programador alguma amplitude extra. Por exemplo, todos os espaços em branco e as tabulações em uma linha de comando serão ignoradas. Isto permite parágrafos que ajudam na clareza. E também, as linhas que iniciam com um asterisco serão ignoradas pelo interpretador, o que permite ao programador usar tais linhas para comentários e explicações.

Um programador também pode continuar uma linha de comando em uma segunda linha no arquivo de comandos, usando o ponto e vírgula para dizer ao interpretador do dBase para

ignorar a tecla return. Assim, as três linhas no exemplo que segue seriam executadas como sendo um comando:

```
*3-linhas de comando
DISPLAY ALL FOR;          <cr>
CIDADE = "Belem";         <cr>
.AND. CONTRIB > 0         <cr>
```

Como nos comandos digitados no modo interativo, as letras maiúsculas e minúsculas nos arquivos de comandos serão todas consideradas da mesma forma. As linhas vazias em um arquivo serão ignoradas.

MODIFY COMMAND – O Editor do dBase

Como uma conveniência ao programador, o dBase possui um editor construído dentro do programa, que pode ser usado no modo interativo. O comando para editar um arquivo é:

```
. MODIFY COMMAND <nome de arquivo>
```

Se o arquivo nomeado existe, o dBase abrirá para as mudanças. Se o dBase não puder encontrar o arquivo pelo nome, ele assumirá que este é um arquivo novo, nos dirá que este é o caso com a mensagem "NEW FILE" e abrirá um arquivo novo para trabalharmos.

É melhor desativar a intensidade antes de usar MODIFY COMMAND. Durante a sessão de edição, o ^T pode ser usado para apagar e o ^N para inserir uma linha, como com MODIFY STRUCTURE. O ^W grava o arquivo e ^Q sai sem salvar. A maioria dos outros caracteres de controle usados dentro do MODIFY COMMAND é padrão, com a exceção de ^R para "sobe uma página" e ^C "desce uma página".

Se especificarmos um nome de arquivo sem extensão para o MODIFY COMMAND, o editor automaticamente assume o ".CMD" ou ".PRG" final. Porém, qualquer arquivo de texto pode ser aberto para a edição, com especificação de seu nome de arquivo completo e tipo. O MODIFY COMMAND é um utilitário excelente para corrigir letras que faltam nos cabeçalhos de campo em um arquivo .FRM do REPORT FORM, por exemplo. Se o arquivo não possuir tipo em seu nome, como em "C:NOEND", use um ponto para fechar o seu nome quando chamá-lo:

```
. MODIFY COMMAND c:noend. <cr>
```

O MODIFY COMMAND possui uma reputação pobre entre os programadores em dBase, com alguma justificação. O seu maior defeito é que o editor truncará as linhas do programa editado que excederem a 79 caracteres.

Desde que os programadores escolham escrever seus programas usando outros editores com mais características que o MODIFY COMMAND, sempre há uma grande probabilidade de que estes programas sejam mutilados, se eles se sujeitarem a uma edição posterior com MODIFY COMMAND. Fique atento – o MODIFY COMMAND anulará o final direito de nossos programas.

Mas isto não significa que o editor do dBase seja inútil. Muitas tarefas de programação do dBase tais como escrever pequenos utilitários de auxílio para realizarmos um trabalho rápido, que são feitas em nove ou dez linhas de codificação, são sempre abandonadas após o uso. Por que sair do dBase, trocar o disco e chamar algum processador de texto potente e voltar ao dBase?

Eu recomendo MODIFY COMMAND para isto.

Processadores de Texto como Editores de Programa

Com certeza, todos os programadores viverão sem o MODIFY COMMAND, mesmo porque um bom trabalho exige bons instrumentos. Algumas características como pesquisa – e – reposição, procura rápida e movimentação de blocos, nenhuma das quais estão disponíveis em MODIFY COMMAND, são muito úteis para serem desprezadas.

Se estamos usando nosso processador de texto para a edição de um programa pela primeira vez, agora é tempo para revermos a sua documentação. A maioria destes programas utiliza alguns caracteres de controle, como marcas para justificar palavras dentro das linhas. Estas marcas são invisíveis na tela, mas serão alcançadas pelo interpretador do dBase.

Para prevenirmos este problema, a maioria dos processadores de texto possui um modo "não documento" ou um "editor de programas" que pode ser selecionado. Alguns processadores de texto também permitirão a reinstalação do programa para que ele entre automaticamente no modo não documento, quando chamado. Em WordStar, por exemplo, um arquivo de comandos dBase deveria ser carregado utilizando o comando "N" (não documento) no menu.

Seja qual for o editor ou processador de texto, descubra quais os comandos necessários para esconder ou desconectar seus sinais automáticos.

O CONTROLE DE ESTRUTURAS

Vamos escrever um arquivo de comandos que apagará todos os registros no arquivo em uso. Chamamos MODIFY COMMAND ou nosso processador de texto favorito e abrimos um arquivo chamado "DELETUDO.CMD" no qual colocaremos:

```
*  DELETUDO.CMD  Desenhado para apagar todos os registros
*                do arquivo em uso
 DELETE
 SKIP
 DELETE
 SKIP
 DELETE
 SKIP
 DELETE
 SKIP
```

Neste ponto, começamos a compreender que temos um problema com o desenho deste programa. DELETUDO não será totalmente útil, já que nunca saberemos, até abrirmos o arquivo, quantos DELETE/SKIPs colocaremos. O que precisamos de verdade, é de um comando que controle nossos pares de DELETE/SKIP até que todo o arquivo seja eliminado.

Inserimos o *controle das estruturas* do dBase. Estes são comandos que aparecem somente em arquivos de comandos e têm a finalidade de controlar a execução de outros comandos. Em nosso exemplo acima, poderíamos ter escrito DELETUDO assim:

```
* DELETUDO.CMD - Segunda tentativa, com um controle de circuito
 DO WHILE t
    DELETE
    SKIP
 ENDDO
```

Embora ocupe muito espaço para mostrar operações simples em todos os arquivos de comandos deste livro, este é o nosso primeiro, então vamos dar uma olhada. Aqui está uma operação de DELETUDO feita em SOCIOS. Mostraremos SOCIOS antes e depois para mostrar os efeitos do arquivo de comando:

```
. LIST
00001  Mila      Moreira   R.Elma, 111           Sao Paulo  SP 92109   5.00 .T.
00002  Ana       Amorim    R.Tres, 333           Natal      RN 10010   0.00 .T.
00003  Sonia     Simoes    R.Para, 831           Belem      PA 21105   0.00 .F.
00004  Monica    Medeiros  R. Diva, 2219         Curitiba   PR 44110   5.99 .T.
00005  Pedro     Penedo    R.Sao Luiz, 22        Sao Paulo  SP 92109   0.00 .F.
00006  Zilda     Zorzi     Rua Pilar, 44         Aracaju    SE         8.77 .T.
00007  Carlos    Covas     R. "A", 88            Sao Paulo  SP 92101   0.25 .T.
00008  Davi      Dunas     Rua das Flores, 123   Aracaju    SE 36222   1.00 .T.
00009  Ires      Imare     R.Neusa, 99           Belem      PA 21233   7.50 .F.
00010  Henrique  Hipolito  R.Doze, 21            Sao Paulo  SP 92111  21.00 .T.
```

```
. GOTO TOP   (*LIST nos deixou na base, lembra-se? *)
. DO DELETUDO
00001 DELETION(S)      (*aqui o DELETE *)
RECORD: 00002          (*aqui o SKIP *)
00001 DELETION(S)
RECORD: 00003
00001 DELETION(S)
RECORD: 00004
00001 DELETION(S)
RECORD: 00005
00001 DELETION(S)       (*Está funcionando *)
RECORD: 00006
00001 DELETION(S)
RECORD: 00007
00001 DELETION(S)
RECORD: 00008
00001 DELETION(S)
RECORD: 00009
00001 DELETION(S)
RECORD: 00010
00000 DELETION(S)
RECORD: 00010           (*Espere, ele não vai parar! *)
00000 DELETION(S)
RECORD: 00010
00000 DELETION(S)     <esc>
RECORD: 00010         (*Um "escape" para qualquer coisa *)

. LIST
00001 *Mila      Moreira    R.Elma, 111             Sao Paulo  SP 92109   5.00 .T.
00002 *Ana       Amorim     R.Tres, 333             Natal      RN 10010   0.00 .T.
00003 *Sonia     Simoes     R.Para, 831             Belem      PA 21105   0.00 .F.
00004 *Monica    Medeiros   R. Diva, 2219           Curitiba   PR 44110   5.99 .T.
00005 *Pedro     Penedo     R.Sao Luiz, 22          Sao Paulo  SP 92109   0.00 .F.
00006 *Zilda     Zorzi      Rua Pilar, 44           Aracaju    SE         8.77 .T.
00007 *Carlos    Covas      R. "A", 88              Sao Paulo  SP 92101   0.25 .T.
00008 *Davi      Dunas      Rua das Flores, 123     Aracaju    SE 36222   1.00 .T.
00009 *Ires      Imare      R.Neusa, 99             Belem      PA 21233   7.50 .F.
00010 *Henrique  Hipolito   R.Doze, 21              Sao Paulo  SP 92111  21.00 .T.
```

De certa forma, DELETUDO é uma alegoria para muitos dos temas que serão desenvolvidos no restante deste livro. Nosso pequeno problema funcionou (ou pareceu funcionar) perfeitamente até uma "condição limitada", o último registro. Então, em um circuito eterno de mensagens de apagamento repetidas, vimos o erro em nossa lógica – nosso circuito não possuía ponto de saída.

Antes de prosseguirmos, vamos simplesmente dizer aqui, que os controles de estruturas tomarão decisões para nós de dentro de nossos programas, mas devemos ter muito cuidado para considerar todas as possibilidades quando escolhermos as condições que escrevemos em nosso controle de estruturas.

DO WHILE – Circuitos

O DO WHILE do dBase, quando usado em um arquivo de comandos, ativará um circuito que será executado até que uma expressão booleana seja falsa:

```
DO WHILE <exp. booleana>  (*Se for verdadeira . . . *)
   <executa este comando>
   <executa este comando>
   <executa este comando>
ENDDO  (*Voltaremos ao DO WHILE e checaremos o booleano de novo *)
```

Uma declaração DO WHILE deve sempre ter uma declaração booleana para o teste do circuito e um ENDDO (é uma palavra, não duas) para fechar o circuito. Se a expressão booleana for falsa quando o DO WHILE for encontrado, o conteúdo inteiro da estrutura será pulado e a linha do programa depois do ENDDO será executada como se o DO WHILE não existisse.

Uma implicação desta estrutura é que se nada acontecer dentro do circuito DO WHILE para alterar o valor verdadeiro/falso da expressão booleana testada, então o circuito a executará sempre. Isto não é necessariamente um erro. Há casos em que desejamos colocar uma rotina especial em um "circuito para sempre"; porém, geralmente quando fazemos isto, deixamos a porta de saída fora do circuito, perdida em algum lugar no programa:

```
DO WHILE t  (*O "t" significa verdadeiro *)
   .
   <algum comando de saida>
   .
ENDDO
```

Não é necessário deixar parágrafos nas linhas de comandos entre DO WHILE e ENDDO – os espaços, como lembramos, são ignorados pelo interpretador – mas a entrada de parágrafo dentro de um circuito é uma boa prática e ajuda a legibilidade.

IF ELSE – Pontos de Bifurcação

O comando DO WHILE é perfeito para controlar os circuitos necessários para processamentos repetidos, mas muitas tarefas de programação precisam de decisões para alguém em dificuldades

sobre o que fazer depois, sem circuitos. Estes momentos de decisão são chamados **pontos de bifurcação** e são controlados em dBase pela estrutura IF:

```
IF <exp. booleana>
   <comando>
   <comando>
   <comando>
ENDIF
```

Como DO WHILE, uma declaração IF avalia a expressão booleana para decidir se executa ou não as linhas entre ele e seu ENDIF. Se o booleano possuir um valor verdadeiro, as linhas serão executadas. Se for falso, serão saltadas e a primeira linha depois do ENDIF será executada. Nos dois casos, verdadeiro ou falso, não haverá circuito ou reinteração de nenhuma das declarações — IF é uma tentativa única.

Vamos ver um pequeno arquivo de comandos que limpa a tela, obtém uma letra a partir do teclado e imprime uma mensagem se a tecla for um "H" maiúsculo:

```
*  HELLO.CMD - Um programa com IF
*
*  limpa a tela
   ERASE
*  Aguarda um caractere do teclado
   WAIT TO tecla
   IF tecla = "H"
      * imprima nossa mensagem
      ? "     Hello, mundo!"
   ENDIF
   * A operacao nao termina ate que a limpeza esteja completa
   RELEASE tecla
```

As construções IF também permitem o uso de um ELSE, como em:

```
IF <exp.booleana>
   .
   <comandos para booleano verdadeiro>
   .
ELSE
   .
   <comandos para booleano falso>
   .
ENDIF
```

Em uma construção IF-ELSE, um conjunto de declarações sempre será executado. Ao elaborar nosso programa acima, poderíamos ter feito com que o computador tocasse o alarme e reclamar se qualquer tecla exceto a "H" fosse digitada:

```
*  HELLO2.CMD - Um programa com IF  e ELSE
*
*  limpa a tela
   ERASE
   Aguarda um caractere do teclado
   WAIT TO tecla
   IF tecla = "H"
      * imprima nossa mensagem
      ? "         Hello, mundo!"
   ELSE
      * ASCII 7 (^G) e o alarme
      ? CHR(7)
      ? " Nao era esta tecla!"
   ENDIF

   * A operacao nao termina ate que a limpeza esteja completa
   RELEASE tecla
```

Note que o ELSE não precisa de nenhum teste booleano em sua linha. As declarações ELSE serão executadas se a expressão booleana na linha IF for falsa.

Novamente, a entrada de parágrafo é para clareza.

DO CASE – Tabelas de Salto

Freqüentemente, os programas contêm menus que oferecem ao usuário uma seleção de coisas para fazer em seguida. O usuário digita uma tecla simples e o programa vai na direção selecionada.

Em dBase, como em outras linguagens, estes pontos de bifurcação de múltipla direção são manipulados pelas declarações CASE. O controle de estruturas CASE pode comparar uma expressão com vários caracteres e executar os comandos apropriados quando uma igualdade é encontrada. Naturalmente as expressões comparadas devem ser do tipo caractere, mas será aceita qualquer expressão que retorna um caractere. As estruturas das expressões também permitem um OTHERWISE, que será executado pelo interpretador, se nenhuma igualdade for encontrada. Se não houver OTHERWISE e igualdade, o controle "pula" a estrutura CASE e a primeira linha do programa depois de ENDCASE é executada:

```
DO CASE
   CASE <exp.carac 1> = "<carac 1>"
      .
      <comandos 1>
      .
   CASE <exp.carac 2> = "<carac 2>"
      .
      <comandos 2>
      .
   CASE <exp.carac 3> = "<carac 3>"
      .
      <comandos 3>
      .
   OTHERWISE
      .
      <comandos default>
      .
ENDCASE
```

Note que a expressão de caractere pode ser qualquer coisa que retorna um caractere, incluindo uma string ou campo de caractere em um arquivo, dos quais retornaria o primeiro caractere na string. Note também que as expressões de caracteres não precisam estar relacionadas (embora seja uma boa idéia).

Apresentaremos aqui uma estrutura típica CASE. Ela compara uma expressão simples com uma variedade de caracteres para vermos, neste caso, qual escolha entrou no teclado.

```
* Nosso programa comeca aqui
 .
 .
 .
* Ativamos o menu (este programa nao funcionaria)
 1 = Opcao 1
 2 = Opcao 2
 3 = Opcao 3
*
 WAIT TO escolha
* agora o CASE
 DO CASE
    CASE escolha = "1"
       * opcao 1
    CASE escolha = "2"
       * opcao 2
```

```
   CASE escolha = "3"
      * opcao 3
ENDCASE
* e o resto do programa vem aqui
```

Esta combinação-padrão de CASE com WAIT aparece muitas vezes nos programas dBase, por isso é bom observá-la bem. Ela não seria tão usada se não funcionasse. Uma estrutura não muito vista, porém útil, é usada para comparar uma variedade de expressões com um simples caractere:

```
DO CASE
   CASE varmem1 = "A"
      * opcao 1
   CASE varmem2 = "A"
      * opcao 2
   CASE varmem3 = "A"
      * opcao 3
ENDCASE
```

Compare as estruturas CASE acima com esta, igualmente corretas, porém com alguma coisa menos esclarecida:

```
DO CASE
   CASE varmem = "A"
      * opcao 1
   CASE $(nome,3,1) = "c"
      * opcao 2
   CASE  STR(VAL(endereco)) <> "0"
      * opcao 3
ENDCASE
```

FOR – A Falta de Controle de Estrutura em dBase

A maioria das linguagens de alto nível possui um controle de estrutura chamado "circuito FOR", usado para executar um segmento do programa por um número finito e conhecido de vezes. Entretantc em BASIC poderíamos ver:

```
10 FOR X=1 TO 8
20 ...
30 ...
40 NEXT X
```

Ou em Pascal teríamos:

```
for I := 1 to Numero de Dependentes do Deducoes(RecEmpreg);
```

Nos dois casos, o valor do circuito FOR será aquele que informarmos quando soubermos previamente quantas vezes queremos operar.

O dBase não possui um circuito FOR. Além disto, temos que construir nossos próprios circuitos FOR fora da combinação de um circuito DO WHILE, e um contador de variável de memória sempre que uma destas tarefas apareça:

```
. STORE 1 TO count

  DO WHILE count < num:dep
     .
     .
     .
     STORE count + 1 TO count
  ENDDO
```

Note que a última etapa do circuito DO WHILE decrementa o contador. Sempre esqueço esta etapa quando escrevo programas rapidamente, e, como podemos imaginar, meus circuitos tendem a continuar por um tempo maior que o necessário.

Encaixe

Quando um controle de estrutura é colocado dentro de outro, é chamado encaixe. Programas mais complexos são impossíveis de serem escritos sem encaixe, mas eles também apresentam riscos consideráveis de erros no programa. As regras básicas para o encaixe são:

1. Para cada DO WHILE deve ter um ENDDO; para cada IF, um ENDIF; e para cada DO CASE, um ENDCASE.
2. O controle das estruturas pode ser encaixado, mas eles nunca podem cruzar!

A Figura 6-2 mostra alguns exemplos de um programa correto e um com erros de encaixe. Com estes exemplos, podemos ver como é sensata a inclusão de parágrafos nas linhas dentro do controle das estruturas. Se o END** que fecha a declaração do controle de estrutura estiver alinhado com sua abertura, as chances de cruzar dois controles de estrutura reduzem consideravelmente.

As duas construções de encaixe usadas mostrarão a ação desta técnica. A primeira, coloca uma declaração CASE dentro de um circuito DO WHILE para assegurar que o programa não abandonará o menu até que uma das opções tenha sido selecionada. Depois, os usuários podem acidentalmente pressionar uma tecla que não consta do menu — com certeza eles merecem uma segunda chance!

CODIFICAÇÃO CORRETA	**ERROS**
(*Um IF em um DO WHILE*)	(*Estruturas cruzadas*)

```
DO WHILE                    DO WHILE
   IF                          .
     .                         IF
   ELSE                          .
     .                         ELSE
   ENDIF                         .
   .                        ENDDO
   .                             .
ENDDO                       ENDIF
```

(*IFs encaixados*)	(*IFs Abertos*)

```
IF                          IF
   .                           .
ELSE                        ELSE
   IF                          IF
     .                           .
   ELSE                     ENDIF
     .
   ENDIF
ENDIF
```

Figura 6-2 Exemplos de uma codificação correta e uma codificação com erros de encaixe.

```
DO WHILE t
   * encaixar o menu aqui
   * consiga a entrada com um WAIT
   WAIT TO escolha
   DO CASE
      CASE !(escolha) = "X"
         * DO rotina X
      CASE !(escolha) = "Y"
         * DO rotina Y
      CASE !(escolha) = "Z"
         * DO rotina Z
      * nao precisa do OTHERWISE
   ENDCASE
   * toque o sino quando a entrada for errada
   ? CHR(7)
ENDDO
```

Agora se nosso usuário pressionar uma tecla "A" em vez de "x", "y" ou "z", o circuito DO WHILE manterá o programa no menu até que uma entrada válida seja feita em um dos casos. Note que a função (!) de letra maiúscula é usada para tornar válida a entrada de letras minúsculas assim o usuário não precisará lembrar de colocar as maiúsculas.

Note também que em cada caso, X, Y e Z devem assegurar que o circuito DO permaneça de maneira apropriada. Veremos as formas de fazer isto na próxima seção.

No próximo exemplo, usaremos IFs encaixados para construirmos uma alternativa para uma tabela de um CASE. A estrutura CASE possui a desvantagem de permitir somente a comparação de expressões com caracteres. Daquela vez, quando tínhamos de escolher entre caminhos múltiplos baseados na importância relativa de uma quantidade numérica, por exemplo, necessitamos usar IFs encaixados.

```
IF contrib < 5
   * DO cretino
ELSE
   IF contrib < 25
      * DO paoduro
   ELSE
      IF contrib < 100
         * DO camarada
      ELSE
         * DO vivaosocio
      ENDIF
   ENDIF
ENDIF
```

Encaixando Arquivos de Comando

Até agora, consideramos simples arquivos de comandos como programas. Revendo: um arquivo de comandos é um arquivo de texto no disco que o interpretador do dBase pode abrir e executar, uma linha de cada vez. O controle das estruturas são linhas de código dentro dos arquivos de comando que direcionam a execução do programa.

Mas o que aconteceria se estivéssemos em um arquivo de comandos nomeado "BOM TRAB .CMD", e o interpretador do dBase encontrasse uma linha que diz "DO OUTRO"? O comando DO dentro de um arquivo de comando é válido como outro comando qualquer. No caso acima, o dBase suspenderia a execução de BOMTRAB neste ponto e iria procurar OUTRO.CMD no drive default. Assumindo que ele encontrasse OUTRO, executaria então uma linha de cada vez e voltaria ao BOMTRAB na linha abaixo de "DO OUTRO". Em outras palavras, ele voltaria para seu lugar e continuaria. Este processo é mostrado no diagrama da Figura 6-3.

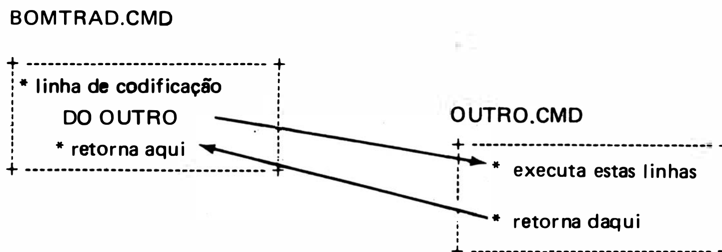


Figura 6-3 Diagrama representativo da execução de um comando DO.

É claro que OUTRO.CMD pode chamar outro programa com um comando DO, mas este processo não pode continuar para sempre. O dBase não pode ter mais do que 16 arquivos abertos de uma vez e isto representa *todos* os arquivos – arquivos de dados, arquivos de índice, arquivos de comandos e outros arquivos que vamos conhecer mais tarde, como arquivos formatados.

A habilidade de um arquivo de comandos em chamar outro afeta profundamente o desenho do programa e a filosofia de programar. Isto significa que em vez de esticar nosso programa em um longo arquivo de texto, somos encorajados pelo sistema a pensar em termos do que um programador em BASIC poderia chamar de "sub-rotinas", ou em Pascal "procedures".

De fato, um programa dBase, às vezes, pode consistir em 20 ou 30 arquivos pequenos em um disco, a maioria deles não possuindo mais que uma ou duas páginas. Um arquivo de comandos dBase de tal programa poderia ser tão pequeno a ponto de ter oito linhas de comandos; e este mesmo arquivo de comandos poderia ser chamado por outros tantos arquivos de comandos. A programação em dBase é extremamente modular, o que proporciona a escrita de programas livres de erros.

CANCEL, RETURN e LOOP – Movimentando os Arquivos de Comando

O dBase possui dois comandos especiais disponíveis para desvios entre arquivos de comandos encadeados e um terceiro para desvios dentro de um circuito DO WHILE.

CANCEL possui o efeito de abortar o arquivo de comandos em que ele aparece e voltar o usuário para o dBase no modo interativo, não importa qual seja o nível que o programa se encontra na cadeia de programas. CANCEL não faz nada com a configuração – os arquivos não são fechados nem há preocupação com as variáveis de memória – assim o usuário volta ao prompt com a configuração no estado em que estava quando CANCEL foi encontrado pelo interpretador. Ele é sempre usado dentro de um menu principal com a seleção:

```
" X = Saida para o dBASE"
      .
      .
   CASE !(escolha) = "X"
        CANCEL
```

RETURN possui de certa forma o mesmo efeito que CANCEL, exceto que ele volta ao controle do programa para qualquer programa que o tenha chamado. Em outras palavras, se o arquivo de comandos fosse chamado de outro arquivo de comandos (encadeado), o controle voltaria para o arquivo que o chamou. Se o arquivo de comandos fosse chamado do modo interativo, porém, RETURN retornaria o usuário ao prompt. RETURN é usado em submenus como uma indicação para voltar ao menu principal ou pode ser usado dentro de uma declaração IF para voltar ao controle ao programa chamador depois de completar algum processo. RETURN não altera nada na configuração do dBase.

LOOP é usado dentro de um circuito DO WHILE para voltar o fluxo do programa para o topo do circuito sem a execução dos comandos que estiverem entre o LOOP e o ENDDO.

```
DO WHILE <condicao>
   * processo comum
   IF  registro errado
       SKIP
       LOOP
   ENDIF
       * processo de registro correto
ENDDO
```

Note que o exemplo acima poderia ser reescrito como:

```
DO WHILE <condicao>
   * processo comum
   IF  registro errado
       SKIP
   ELSE
       * processo do registro correto
   ENDIF
ENDDO
```

Este é provavelmente o desejável na maioria dos casos. O LOOP (especialmente) e RETURN (para certas ocasiões) seria usado criteriosamente. Os dois comandos são "GOTOs ocultos", não no sentido de GOTO do dBase, que faz movimentação dentro de um arquivo de dados, mas no sentido do GOTO do BASIC.

Não queremos nos aprofundar muito em uma controvérsia, mas notaríamos que muitos acadêmicos e programadores profissionais são contra os programas em linguagens de alto nível que usam saltos livres. LOOP representa exatamente este tipo de salto – ele vai diretamente ao DO WHILE mais perto acima dele. Embora estes saltos aparentem certa inocência, eles podem apresentar erros desastrosos, mesmo em uma rotina pequena. Consideremos um módulo de

pesquisa para SOCIOS que deverá encontrar um registro pelo SOBRENOME ou CIDADE. Depois que o usuário disse ao dBase qual o caminho da pesquisa, ele poderia ter acionado os índices – cidade necessita de um índice de cidade e o sobrenome, um índice de sobrenome. O que aconteceria:

```
DO WHILE <condicao>
   * perguntamos se a pesquisa e por cidade
   * ativamos o indice de cidades
   * obtemos a cidade
   FIND &pesqcidade
   * o que acontece se a cidade nao estiver aqui?
   IF # = 0
        ? CHR(7)
        ? " NAO ENCONTREI"
        LOOP                  (*E aqui está o problema *)
   ENDIF
   *
   *  Rotinas para registro nao encontrado
   *
   *  ESTE E O LUGAR DE REATIVAR O INDICE
ENDDO
```

O problema com este circuito é óbvio. Vamos chegar novamente ao topo do DO WHILE com nossas indexações trocadas e não há nenhuma maneira de ativarmos o correto. Este é um erro comum quando usamos o LOOP e isto é perigoso. Vamos refletir: a rotina funcionaria, talvez por semanas, até que houvesse um NO FIND na pesquisa da cidade (e teríamos de assumir que esta é a pesquisa menos comum, caso contrário sua indexação teria sido a primeira). Além disso, quando ocorre um circuito, o usuário do programa não recebe nenhum aviso se as indexações não foram rearmazenadas – ao contrário, os registros apareceriam como se estivessem perdidos porque as pesquisas do sobrenome pararam de funcionar. E se o usuário interrompesse e indicasse novamente, o programa de repente estaria funcionando – até o próximo NO FIND na pesquisa da cidade.

Se usarmos LOOPs (e em alguns casos RETURNs), sempre devemos parar depois de o digitarmos e pensar: "O que já aconteceu do DO WHILE até aqui?" E se usarmos LOOPs, devemos usá-los bem no alto, perto de seus DO WHILE, para minimizar as chances de erros.

O ESTILO DE PROGRAMAÇÃO I

Escrever um programa pode ser uma das experiências mais gratificantes e criativas. Como qualquer ato de criação, há instintos internos de entendimento que representam uma própria recompensa – e também um ótimo momento quando o programa funciona.

Como em qualquer processo criativo, existem estilos em relação aos programas. Estes pontos estilísticos originam um interesse imediato, que é fazer o programa o mais fácil possível de entender, depurar e manter. A não ser programas pequenos utilitários, nenhum outro programa é feito para se usar e em seguida jogar fora. Eles são terminados, modificados, usados por um tempo, modificados novamente, e assim por diante durante a existência de um programa. Um bom programa, como algo vivo, deve tender ao crescimento.

O bom estilo em programação é desenhar programas de tal forma a facilitar a sua depuração e manutenção. Uma maneira de manter o bom estilo é termos consciência, quando escrevemos os programas, de que um estranho os entenderia sem precisarmos explicar. Lembre-se sempre de que você é a pessoa mais indicada para representar este estranho, porque quando voltar a um projeto depois de alguns meses ficará satisfeito em aceitar qualquer sugestão sobre como o seu próprio programa funciona. Apresentamos alguns princípios estilísticos que oferecemos como um guia:

1. Comente o seu programa:

Uma tela de programa toma até 24 linhas. Se não pudermos ver três ou quatro linhas de comentários nela, não estaremos comentando o suficiente. Cada arquivo de comandos dBase teria comentários no ponto de entrada fornecendo o nome do arquivo de comandos, de onde ele foi chamado, o que ele deve fazer e sob qual configuração espera desenvolver o trabalho.

Outros lugares para comentários inclui o início do controle das estruturas (explicar qual é o booleano para os testes de IF ou DO WHILE e por que), e qualquer outras mudanças no arquivo de dados ou de index. Geralmente uma ou duas palavras são o bastante.

Coloque os comentários em lugares onde achar que seu programa está falho ou onde puder prever a necessidade eventual de modificações. Podemos colocar séries de caracteres em nossos comentários, tais como "# sem previsões para um NO–FIND", assim mais tarde podemos pesquisar pela série "##" usando a característica rápida de pesquisa do processador de texto. Isto é especialmente importante se houver características em nosso programa que necessitem de modificações no caso de ser transferido de uma versão do dBase para outra.

2. Use letras maiúsculas para melhorar a legibilidade:

Lembre-se de que o dBase não faz diferença entre maiúsculas e minúsculas (exceto em dados, é lógico) e considera-as igualmente em todos os nomes de arquivo, nomes de campo, nomes de variáveis de memória e comandos. Por exemplo, os três dos seguintes comandos são equivalentes:

```
. list for cidade       = "Sao Paulo"   (*Minúsculas*)
. LIST FOR CIDADE       = "Sao Paulo"   (*Maiúsculas*)
. list fOr CiDaDe       = "Sao Paulo"   (*Misturadas*)
```

Como usar esta característica para melhorar a legibilidade? Esta é a questão! É muito mais uma questão de estilo individual. Alguns programadores defendem a teoria de usar todos os comandos dBase e palavras em maiúsculas e todos os campos, varmem e nomes de arquivo em minúsculas. Outros fazem o oposto. Algumas proteções do Pascal como usar minúsculas e maiúsculas misturadas em suas variáveis de nome, argumentando que "Taxa Estadual" é mais fácil de ler do que "TAXA ESTADUAL" (sim, mas e "taxa: estadual"?). Eu, particularmente, experimentei cada sistema usando maiúsculas e não encontrei uma regra simples que me satisfaça em todos os casos. Porém, acredito que minúscula é geralmente mais legível do que maiúscula, assim a minha tendência é usar maiúsculas mais para controle de estruturas e os verbos dentro das linhas de comando. Desta forma, estabeleça seu próprio estilo, mas esteja ciente de que esta opção foi avaliada.

3. **Use o parágrafo para mostrar a estrutura do programa:**

O parágrafo é muito mais importante para o estilo do dBase do que uma questão de estilo pessoal, como são os comentários e maiúsculas. As duas regras de uso de parágrafos são: a) todos os comandos entre as linhas de início e término de um controle de estrutura seriam entradas tabuladas; e b) todas as linhas de comandos de encaixes iguais teriam a mesma tabulação. A única exceção seria que uma linha de continuação deve ser deslocada com dois espaços para a melhor legibilidade.

```
* comentario explicando por que  xyz
DO WHILE  xyz
   * comentario explicando por que listar
   LIST ALL sobrenome,contrib;
     pagamento FOR cidade = "&var:mem"
   * comentario explicando por que  abc
   IF abc
      CLEAR
      ? CHR(7)
      CANCEL
   ELSE
      DO processo
   ENDIF  abc
ENDDO  xyz
```

O segmento de programa acima mostra o uso de parágrafos, maiúsculas e comentários, como descrito. Note que um ponto e vírgula foi usado para quebrar a linha de comando LIST em duas linhas, e que a segunda linha foi entrada em dois espaços para mostrar sua ligação com a linha de cima. Uma nova característica foi apresentada aqui nas linhas ENDDO e ENDIF – comentários na linha de fechamento do controle das estruturas. O interpretador do dBase ignorará tudo que aparecer na mesma linha depois de ENDDO, ENDIF, ENDCASE e ENDTEXT, que conheceremos

mais tarde. Isto nos permite rotular estes END-qualquer declaração para que saibamos exatamente a qual controle de estrutura eles pertencem; muito útil quando eles estão empilhados, especialmente nas pequenas rotinas de preparo inicial:

```
            .
            .
            ENDIF
            SKIP  (*Este SKIP esta no lugar certo?*)
        ENDIF
    ENDIF
ENDDO
```

4. Seja racional e coerente ao nomear as variáveis:

Já vimos que arquivos de dados podem ser acrescentados uns aos outros muito mais facilmente quando compartilham os mesmos nomes de campo para os mesmos itens de dados. Com uma característica poderosa como esta, não faz sentido dar nomes coerentes para nossos campos? Muitas informações podem ser comprimidas nos nove caracteres de um nome de campo (sabemos que as regras para nomes de campo dizem dez caracteres). ST:TAX, FED:TAX, PROD:CODE são todos nomes de campo com significado e sem erros. Além disso, um pouco de imaginação pode economizar letras. Se necessitarmos de um nome de campo para a pessoa de contato em uma empresa, por que não chamá-lo CONT? As datas são itens importantes de dados que ocorrem em lugares diferentes nos bancos de dados e não é raro nomear campos de datas como: PAG:DATA ou REC:DATA ou algo mais descritivo do que apenas "DATA". Vamos finalizar a nomeação de variável mencionando a questão dos nove caracteres/dez caracteres. Tomou-se um costume em dBase limitar os nomes de campos para nove caracteres. A razão é: se os nomes de campos têm nove ou menos caracteres, então pode ser seguida a convenção que qualquer variável de memória associada com o campo pode ser nomeada acrescentando-se um "M" a ele. O "M" é geralmente incluído ao contrário do resto do nome da variável de memória para a legibilidade, como em "mCONTRIB" ou "Mcontrib". Então, se uma variável de memória estiver sendo criada para guardar REC:DATA para um registro particular, o comando será:

```
STORE rec:data TO Mrec:data
```

A simplicidade e racionalidade desta convenção transformou-a em universal e eu a recomendo. (A convenção "M" que apareceu na mais nova documentação do dBase, foi originalmente uma inovação de Adam Green, do Software Banc, o mais recente *expert* em dBase, cujo livro de exercícios e seminários ensinou-me muito do que sei sobre o software.)

A PROGRAMAÇÃO "ESTRUTURADA"

Programar pode tornar-se mais fácil e os programas tendem a diminuir o número de erros quando usamos o método chamado **programação estruturada**. Neste método, aconselhamos

o truque de isolar uma parte do programa, escrever alguma rotina que pareça ser um bom passo para eliminar o problema e então tentar encaixá-lo de volta no programa.

Utilizando este método, primeiro definimos nosso sistema em termos qualitativos e então o dividimos gradualmente em módulos. Durante esta evolução, nunca sabemos como iremos eliminar os problemas – quando nos depararmos com um problema específico, simplesmente escrevemos a linha DO SOLUÇÃO, para que em um outro nível, possamos escrever um arquivo de comandos (SOLUÇÃO) que solucionará o problema.

Um dos instrumentos mais importantes da programação estruturada é o **pseudocódigo**. O pseudocódigo não é propriamente um programa, mas uma descrição do que queremos que o programa execute, em algumas sentenças estruturadas.

Geralmente quando refinamos nosso pseudocódigo, estamos muito perto de um verdadeiro programa dBase funcionando. Quando chega o momento de digitá-lo no computador, apenas algumas perguntas permanecerão, geralmente aquelas como: "Eu adoraria que o dBase me deixasse fazer *isto!*" Uma prova rápida no modo interativo testará a etapa ou etapas fundamentais e o programa terá uma chance muito maior de funcionar corretamente.

EXEMPLO 1 – UM PROGRAMA UTILITÁRIO PARA TRATAR DATAS

Vamos escrever um programa para tratar campos de data. As datas são geralmente gravadas com oito caracteres, dois dígitos para o mês, uma barra, dois dígitos para o dia, outra barra e dois dígitos para o ano. É muito importante na maioria dos programas que este formato seja seguido. O que aconteceria se um evento registrado no dia 20 de janeiro fosse armazenado como "1/20/aa"? Um erro inocente seria o programa tentar encontrá-lo com "FIND 01/20"; obteremos um NO-FIND. E os delimitadores – "01-20-aa" seriam OK? Não, na verdade. Consideremos o comando "LIST FOR data > '01/00/aa' ", considerando "mostre-me os registros a partir do primeiro dia do ano". A condição desta listagem depende da comparação da string, o que significa que os caracteres serão comparados, da esquerda para a direita, com base na sua seqüência de intercalação. Se checarmos a seqüência de intercalação de ASCII, veremos que "–" é "menor" do que " / " e além disto as datas armazenadas com hífens não aparecerão como resposta ao nosso LIST.

Mais tarde aprenderemos uma maneira de proteger nossos programas contra entrada de datas incorretas, mas por enquanto, como podemos consertar datas que entraram no formato errado?

Definição do problema: Às vezes, as datas precisam de zeros à esquerda no mês e no dia, ou ainda precisam de algum delimitador que não seja a barra.

Pseudocódigo

```
veja a data
se estiver errada, conserte e grave
vá para o próximo registro
faça isto até o final do arquivo
```

Muito bom. As duas últimas linhas parecem muito com um SKIP e um DO WHILE.NOT.EOF, mas e as duas primeiras?

```
Primeiro refinamento
se a data for: (*Esta é a "procura" *)
        m/dd/aa  ou
      mm/d/aa   ou
      mm?dd?aa

então conserte:
      incluindo um '0' na frente ou
      incluindo um '0' na posição 4 ou
      caractere diferente de " / " no carac 3 e 6

      substitua

vá para o próximo registro
faça isto até o fim do arquivo
```

Estamos muito perto de um programa. Precisamos montar um controle de estrutura IF e um mecanismo para consertar e substituir. O substituir será provavelmente um REPLACE – esta é a maneira padrão de atualizar um campo. O consertar pede uma manipulação de string – cortar a string de data e colocar as coisas no meio ou no final – então provavelmente usaremos a função substring "$( )" e o operador de concatenação "+".

E o controle da estrutura? A primeira idéia pode ser:

```
IF  caso1
    conserto 1
ELSE
    IF CASO 2
       conserto 2
    ELSE
       IF caso 3
          conserto 3
       ENDIF 3
    ENDIF 2
ENDIF 1
```

Comumente, isto funcionaria bem, mas um IF encaixado não é como a declaração CASE de endereço construído? De certo é! Estamos testando os caracteres em determinadas posições.

Usaremos um CASE, mesmo que um IF funcione. A única pergunta que permanece é se o dBase nos permitirá fazer um REPLACE em um campo depois de mexer com algumas substrings em seus conteúdos. Iremos ao modo interativo e tentaremos alguns comandos diretos como:

```
. REPLACE nome  WITH  $(nome,1,3)+ "???" + $(nome,4,6)

 000001 REPLACEMENT(S)

. DISPLAY nome

Art???ur
```

Isto funciona. Vamos prosseguir com a programação:

```
************************************************
* FIXDATE.CMD   codificado como um exemplo
Ë               d u utilitari dBASE
************************************************
* Este programa espera encontrar um arquivo
* em uso com um campo data com o formato
* mm/dd/aa. Ele esta preparado para consertar:
* (1) m/dd/aa ou (2) mm/d/aa ou (3) mm-dd-aa ,
* nesta ordem.
* Erros em potencial sao datas em qualquer outro
* formato, como m/d/yy
************************************************
*
*  Normalmente desativariamos o talk
*
*  Espera-se que o usuario use  GOTO  TOP
*
*  Inicio do circuito
DO WHILE .NOT. EOF
   * usaremos um case para ver a data
   DO CASE
      * 1o. caso -  m/dd/aa
      CASE $(data,2,1) = "/"
           REPLACE data WITH "0"+ data
      * 2o. caso - mm/d/aa
      CASE $(data,5,1) = "/"
           REPLACE data WITH $(data,1,3)+"0"+$(data,4,4)
```

```
        * 3o. caso - mm-dd-aa
        CASE $(data,3,1) <> "/"
             REPLACE data WITH $(data,1,2)+"/"+$(data,4,2)+"/";
                +$(data,7,2)
     ENDCASE
     * tudo pronto - proximo registro e volta
     SKIP
ENDDO
*
* fim do arquivo FIXDATE.CMD
```

Observe a ordem dos três casos que estamos testando. O utilitário iria funcionar se estivesse invertido? Não, não poderíamos contar com ele. Uma declaração CASE executa o bloco de comandos abaixo do *primeiro* caso verdadeiro. Existem datas tais como "1-5-83" que completam os três casos, mas que poderiam ser eliminadas pelo conserto 2 ou 3 se o primeiro conserto não fosse realizado primeiro.

Vamos testar:

```
. USE DATAS   (*Criamos um pequeno .DBF com nada mais do que um campo de data*)
. DISP STRUCTURE
STRUCTURE FOR FILE:  B:DATAS   .DBF
NUMBER OF RECORDS:   00000
DATE OF LAST UPDATE: 01/01/80
PRIMARY USE DATABASE
FLD        NAME      TYPE WIDTH   DEC
001      DATA          C    008
** TOTAL **               00009


. LIST


00001  11/11/18   (*OK  - Datas*)
00002  1/23/45      (*Algumas datas
00003  4/12/84          erradas
00004  12/4/56           precisando
00005  10/3/56              conserto*)
00006  1/2/33


. GOTO TOP
. DO FIXDATE
RECORD: 00002
```

```
00001 REPLACEMENT(S)    (*Um conjunto
RECORD: 00003              de SKIPs
00001 REPLACEMENT(S)         e
RECORD: 00004                  REPLACEs*)
00001 REPLACEMENT(S)
RECORD: 00005
00001 REPLACEMENT(S)
RECORD: 00006
00001 REPLACEMENT(S)
RECORD: 00006

. LIST
00001  11/11/18
00002  01/23/45
00003  04/12/84
00004  12/04/56
00005  10/03/56
00006  01/2/33    (*Uma segunda passagem consertaria isto? *)

. GOTO TOP
. DO FIXDATE
RECORD: 00002     (*SKIPs de
RECORD: 00003         registros
RECORD: 00004          com
RECORD: 00005            datas
RECORD: 00006               corretas*)
00001 REPLACEMENT(S)
RECORD: 00006

. LIST
00001  11/11/18
00002  01/23/45
00003  04/12/84
00004  12/04/56
00005  10/03/56
00006  01/02/33
```

FIXDATE parece tão bom quanto poderia. Ele consertou todas as datas testadas, precisando somente de uma segunda passagem, em uma data de duplos erros e, talvez o mais importante, ele não apresentou nenhum erro.

Vamos usar o utilitário novamente depois de incluir dois registros com delimitadores incorretos ao nosso banco de dados, um dos quais possui também o mês com somente um dígito:

```
. LIST
00001  11/11/18
00002  01/23/45
00003  04/12/84
00004  12/04/56
00005  10/03/56
00006  01/02/33
00007  01.02.84
00008  1-23-71


. GOTO TOP
. DO FIXDATE
RECORD: 00002
RECORD: 00003
RECORD: 00004
RECORD: 00005
RECORD: 00006
RECORD: 00007
00001 REPLACEMENT(S)
RECORD: 00008
00001 REPLACEMENT(S)
RECORD: 00008


. LIST
00001  11/11/18
00002  01/23/45
00003  04/12/84
00004  12/04/56
00005  10/03/56
00006  01/02/33
00007  01/02/84
00008  1-/3-/1
```

Apareceu um erro. E grande. FIXDATE não pode pegar nada na peneira dos dois primeiros casos que não possuem barra fora do lugar nem as datas com delimitadores incorretos que não possuam barras. Mas o caso 3 só funciona com datas que possuem dois dígitos para o dia e mês. Qualquer data que possua delimitadores incorretos e um único dígito para mês (ou dia) vão ser deixados de lado pelo nosso utilitário bem intencionado.

E não sei o que recomendar, exceto que "COPY TO TEMP" sempre é uma boa providência antes de tentar um novo programa utilitário.

EXEMPLO 2 – UTILITÁRIO DE CHECAGEM DE DUPLICATA

Diferente de outros sistemas gerenciadores de bancos de dados, o dBase não possui característica automática para prevenir entradas duplas no arquivo de dados. Como programadores, esperamos projetar nosso sistema para rejeitar duplicatas ou escrever utilitários que chequem os arquivos de dados para estes casos. Como nosso segundo exemplo de programação, escreveremos um utilitário para checar as duplicatas de SOCIOS.

Declaração do Problema: Através de erros de entrada não verificados por APPEND FROMs, registros duplicados podem entrar em um arquivo de dados. Precisamos de um utilitário que encontrará tais duplicatas e oferecerá ao usuário a opção de apagar um ou outro, ou reter os dois.

Pseudocódigo

pegue todas as duplicatas que são adjacentes
inicie no topo
compare 1 e 2
se eles estiverem duplicados, deixe o usuário apagar um
compare 2 e 3
a mesma operação
faça isto até o final

Como no último exemplo, algumas partes do pseudocódigo parecem comuns – "inicie no topo" é GOTO TOP, por exemplo. E seria fácil obter os registros duplicados perto dos outros pela indexação com uma tecla que combina alguns campos que pegaria as duplicatas – nome e cidade, por exemplo.

E as comparações? São: 1 e 2, então 2 e 3, depois 3 e 4 – provavelmente isto chamaria para um salto dentro do circuito. O único problema de verdade é precisamente como compará-los.

Primeiro Refinamento

indexar em nome + cidade
ir para o topo
iniciar o circuito
    armazenar a 1ª cidade na memória
    saltar
    comparar a 2ª cidade com a 1ª cidade na memória
    se as cidades são as mesmas, deixar o usuário apagar
faça isto até o fim

Tudo que foi dito em nosso pseudocódigo foi a declaração "se as cidades são as mesmas..." Manipularemos isto através da visualização dos registros para as duas duplicatas e usaremos um WAIT para uma variável de memória, que empregaremos para determinar qual registro apagar.

Quando olhamos programa DUPCHECK.CMD, percebemos duas coisas. A primeira, os SKIPs e SKIP-1 foram manipulados cuidadosamente para ter-se certeza de que o programa nunca vai parar ou continuar de um ponto desconhecido. A segunda coisa a notar é o teste booleano no IF para uma checagem de duplicata. Podemos ver por que a condição IF completa é:

IF (cidade = Mcidade) .AND. (.NOT.EOF)"?

Tentaremos DUPCHECK sem o ".AND. (.NOT.EOF)" para descobrir.

```
*********************************************
* DUPCHECK.CMD - Codificado para encontrar
* registros duplicados em qualquer arquivo de
* dados que possuam campos: nome,sobrenome  e
* cidade
*
* ESTA ROTINA FORMA UM INDICE TEMPORARIO
* E PODE APAGAR REGISTROS
*
* Sera preciso salvar o indicador e restarurar
* os indices (com REINDEX) apos o uso
*********************************************
*
* forme um indice para pegar as duplicatas
INDEX ON sobrenome + $(nome,1,1) + cidade TO temp
*
* Vamos comecar no topo
GOTO TOP
* Ativar o circuito
DO WHILE .NOT. EOF
   * coloque a cidade na memoria
   STORE cidade TO Mcidade
   * o proximo, por favor
   SKIP
   * sao cidades duplicadas (e nao e o ultimo registro?)
   IF (cidade = Mcidade) .AND. (.NOT. EOF)
      * mostre a operacao
      ERASE
      SKIP -1
      DISPLAY
      SKIP
      DISPLAY
```

```
      * mostre o caminho
      ?
      ?
      ? " 'T' apaga o de cima, 'B' apaga o de baixo"
      ? " qualquer outra nao apaga"
      * segure a tela e pegue a instrucao
      WAIT TO tira:ele
      * veja a instrucao e obedeca
      IF !(tira:ele) = "T"
         *volte 1 e apage o do topo
         SKIP -1
         DELETE
         SKIP
      ELSE
         IF !(tira:ele) = "B"
            * apagar onde estamos
            DELETE
         ENDIF  apaga o de baixo
      ENDIF  apaga o de cima ou o de baixo
   ENDIF  (da cidade)
ENDDO
*
* limpar tudo e um bom habito
RELEASE tira:ele
* fim do arquivo dupcheck.cmd
```

Aqui está uma rodagem do utilitário. Propositadamente carregamos SOCIOS com algumas duplicatas para tornar interessante. Note através da operação que às vezes apagamos o primeiro dos registros duplicados e algumas vezes o segundo. Uma duplicata, Pedro Penedo, foi deixada através de uma chave que não um "T" ou "B", apenas para provar que isto pode ser feito.

```
. LIST  (*Listagem dos SOCIOS, carregado com duplicatas*)
00001  Mila      Moreira    R.Elma, 111          Sao Paulo  SP 92109   5.00 .T.
00002  Ana       Amorim     R.Tres, 333          Natal      RN 10010   0.00 .T.
00003  Sonia     Simoes     R.Para, 831          Belem      PA 21105   0.00 .F.
00004  Monica    Medeiros   R. Diva, 2219        Curitiba   PR 44110   5.99 .T.
00005  Pedro     Penedo     R.Sao Luiz, 22       Sao Paulo  SP 92109   0.00 .F.
00006  Zilda     Zorzi      Rua Pilar, 44        Aracaju    SE         8.77 .T.
00007  Carlos    Covas      R. "A", 88           Sao Paulo  SP 92101   0.25 .T.
00008  Davi      Dunas      Rua das Flores, 123  Aracaju    SE 36222   1.00 .T.
00009  Ires      Imare      R.Neusa, 99          Belem      PA 21233   7.50 .F.
00010  Henrique  Hipolito   R.Doze, 21           Sao Paulo  SP 92111  21.00 .T.
00011  Ana       Amorim     R.Tres, 333          Natal      RN 10010   0.00 .F.
```

```
00012  Sonia      Simoes      R.Para, 831        Belem      PA 21201   0.00 .F.
00013  Ires       Imare       R.Neusa, 99        Belem      PA 21233   7.50 .F.
00014  Pedro      Penedo      R.Sao Luiz, 22     Sao Paulo  SP 92109   0.00 .F.
00015  Zilda      Zorzi       Rua Pilar, 44      Aracaju    SE         8.77 .T.

. DO DUPCHECK   (*Nosso programa começa aqui *)
00015 RECORDS INDEXED

Natal            (*O comando STORE funcionando *)
RECORD: 00011

( aqui limpa a tela)
RECORD: 00002
00002  Ana        Amorim      R.Tres, 333        Natal      RN 10010   0.00 .T.
RECORD: 00011
00011  Ana        Amorim      R.Tres, 333        Natal      RN 10010   0.00 .F.

  'T' apaga o de cima 'B' apaga o de baixo
   qualquer outra nao apaga
WAITING t
RECORD: 00002
00001 DELETION(S)
RECORD: 00011      (*Pulando
Natal                 os registros
RECORD: 00007           nao
Sao Paulo                repetidos*)
RECORD: 00008
Aracaju
RECORD: 00010
Sao Paulo
RECORD: 00009
Belem
RECORD: 00013

(limpa a tela aqui)
RECORD: 00009
00009  Ires       Imare       R.Neusa, 99        Belem      PA 21233   7.50 .F.
RECORD: 00013
00013  Ires       Imare       R.Neusa, 99        Belem      PA 21233   7.50 .F.


  'T' apaga o de cima 'B' apaga o de baixo
   qualquer outra nao apaga
WAITING b
00001 DELETION(S)   (*Não salte, somente apague *)
```

```
Belem
RECORD: 00001
Sao Paulo
RECORD: 00004
Curitiba
RECORD: 00005
Sao Paulo
RECORD: 00014

(limpa a tela aqui)
RECORD: 00005
00005  Pedro      Penedo          R.Sao Luiz, 22       Sao Paulo  SP 92109   0.00 .F.
RECORD: 00014
00014  Pedro      Penedo          R.Sao Luiz, 22       Sao Paulo  SP 92109   0.00 .F.


 'T' apaga o de cima 'B' apaga o de baixo
  qualquer outra nao apaga
WAITING x   (*Não apaga e vamos em frente *)
Sao Paulo
RECORD: 00003
Belem
RECORD: 00012

(aqui limpa a tela)
RECORD: 00003
00003  Sonia      Simoes          R.Para, 831          Belem      PA 21105   0.00 .F.
RECORD: 00012
00012  Sonia      Simoes          R.Para, 831          Belem      PA 21201   0.00 .F.


 'T' apaga o de cima 'B' apaga o de baixo
  qualquer outra nao apaga
WAITING t
RECORD: 00003
00001 DELETION(S)
RECORD: 00012
Belem
RECORD: 00006
Aracaju
RECORD: 00015

(aqui limpa a tela)
RECORD: 00006
00006  Zilda      Zorzi           Rua Pilar, 44        Aracaju    SE         8.77 .T.
```

```
RECORD: 00015
00015 Zilda     Zorzi        Rua Pilar, 44        Aracaju   SE          8.77 .T.


 'T' apaga o de cima 'B' apaga o de baixo
  qualquer outra nao apaga
WAITING t
RECORD: 00006
00001 DELETION(S)
RECORD: 00015
Aracaju
RECORD: 00015
```

(*Operação completa - de volta ao prompt *)

```
. LIST          (*Vamos ver *)
00002 *Ana      Amorim       R.Tres, 333          Natal     RN 10010  0.00 .T.
00011  Ana      Amorim       R.Tres, 333          Natal     RN 10010  0.00 .F.
00007  Carlos   Covas        R. "A", 88           Sao Paulo SP 92101  0.25 .T.
00008  Davi     Dunas        Rua das Flores, 123  Aracaju   SE 36222  1.00 .T.
00010  Henrique Hipolito     R.Doze, 21           Sao Paulo SP 92111 21.00 .T.
00009  Ires     Imare        R.Neusa, 99          Belem     PA 21233  7.50 .F.
00013 *Ires     Imare        Rua Neusa, 99        Belem     PA 21233  7.50 .F.
00004  Monica   Medeiros     R. Diva, 2219        Curitiba  PR 44110  5.99 .T.
00001  Mila     Moreira      R.Elma, 111          Sao Paulo SP 92109  5.00 .T.
00005  Pedro    Penedo       R.Sao Luiz, 22       Sao Paulo SP 92109  0.00 .F.
00014  Pedro    Penedo       Rua Sao Luiz, 22     Sao Paulo SP 92109  0.00 .F.
00003 *Sonia    Simoes       R.Para, 831          Belem     PA 21105  0.00 .F.
00012  Sonia    Simoes       Av. Para, 831        Belem     PA 21201  0.00 .F.
00006 *Zilda    Zorzi        Rua Pilar, 44        Aracaju   SE          8.77 .T.
00015  Zilda    Zorzi        Rua Pilar, 44        Aracaju   SE          8.77 .T.
```

Podemos observar a chave no índice temporário que foi usado em DUPCHECK.CMD. Selecionei uma combinação do sobrenome, primeira inicial e cidade, o racional visto que isto seria o suficiente para conseguir duplicatas seguintes uma da outra. A escolha da chave, em parte, foi que as "duplicatas" não podem ser digitadas exatamente iguais.

É possível que um registro poderia ter a pessoa como "Sônia Simões", e outra como "Samuel Simões". Ou poderíamos ter duas Zildas Zorzi com o código postal em um registro e nenhum CEP (ou um errado) em outro. Por esta razão, a chave do índice temporário colocaria junto os registros que estão fechados, mas não necessariamente as comparações exatas.

DUPCHECK é um utilitário generalizado; isto é, poderia ser chamado de vários lugares por programas que manipulem arquivos de dados como SOCIOS. O único requisito para que o utilitário trabalhe em um arquivo é a presença dos campos NOME, SOBRENOME e CIDADE.

Já que lemos a seção sobre selecionar nomes de campos coerentes (e concordamos com isto), provavelmente reconheceremos que um utilitário como DUPCHECK pode passar de programa para programa dentro da esfera do dBase.

EXEMPLO 3 – UM UTILITÁRIO PARA ETIQUETAS DE CORRESPONDÊNCIA

Fazer etiquetas é uma operação fundamental em computação. Possuímos um arquivo, SOCIOS, com os dados de endereços e nomes e seria um crime ter de endereçar cartas para aqueles sócios manualmente, agora que possuímos os dados.

Com o que já aprendemos nos dois exemplos anteriores podemos ir direto ao pseudo-código:

Pseudocódigo

pegue um registro
imprima o nome e sobrenome em uma linha
imprima o endereço
imprima a cidade, uma vírgula, o estado, e o cep
imprima linhas em branco o bastante para fazer uma etiqueta de 6 linhas
circule até que obtenha todos os registros

Isto é muito conhecido. A manipulação da string pode levar a alguns erros. A função TRIM( ) parece uma boa candidata para conseguir nome e cidade sem os brancos à direita. Mas queremos ter certeza de que deixamos espaço entre nome e sobrenome, mas não entre cidade e a vírgula. Obviamente terminaremos com três linhas impressas em uma etiqueta de seis linhas quando usarmos SOCIOS. Vamos fazer algumas alterações: escrever um utilitário que possa manipular também um campo chamado compl. Já que SOCIOS não possui um campo compl (complemento do endereço), nosso utilitário terá de ser inteligente o bastante para primeiro checar sua existência a fim de prevenir quebras.

Primeiro Refinamento

pegue um registro
imprima TRIM(nome) <vírgula> sobrenome
imprima o endereço
se o campo compl existir e se não estiver em branco,
    imprima-o
imprima TRIM(cidade) + "," + estado <vírgula> cep
imprima linhas brancas o bastante para fazer uma etiqueta de 6 linhas
circule até que obtenha todos os registros

Certifique-se de que entendeu as operações diferentes de concatenação na linha do nome e na da cidade-estado-CEP antes de abandonar o exemplo. Podemos rever a manipulação da string

no capítulo anterior, testando os vários operadores no modo interativo. Apresentaremos aqui com o que ETQ.CMD parece:

```
************************************************
* ETQ.CMD  - Imprime etiquetas usando
*            formulario de etiqueta unica
************************************************
*
* Faremos este trabalho usando um campo  compl
* (complemento de endereco), embora nao exista
* este campo em SOCIOS
* -- Queremos que o programa funcione para
* qualquer arquivo que tenha nome,sobrenome,etc
*
* O ARQUIVO DE DADOS DEVE ESTAR  ABERTO QUANDO
* ESTE UTILITARIO FOR CHAMADO, COM O INDICADOR
* DE REGISTRO NO LUGAR ONDE SE QUISER COMECAR
*
* SE HOUVER ORDEM ESPECIAL PARA AS ETIQUETAS,
* INDEXAR O ARQUIVO ANTES DE CHAMAR A ROTINA
*
************************************************
*
* nao use goto top - iniciar de qualquer lugar
* ative a configuracao - TALK imprimira as etiquetas
* a menos que seja desativado
SET TALK OFF
*
* Hora do circuito
DO WHILE .NOT. EOF
   * linha do nome
   ? TRIM(nome),sobrenome
   * linha do endereco
   ? endereco
   * linha do complemento - tem um campo complemento?
   IF TYPE(compl) <> "U"
      * OK mas tem alguma coisa em compl?
      IF compl <> " "
         * vamos imprimir
         ? compl
         * vamos cuidar desta linha
         STORE t TO compl:1
      ENDIF (compl em branco)
   ELSE
```

```
      * precisamos de um valor para compl:1
      STORE f TO compl:1
   ENDIF
   * note a concatenacao
   ? TRIM(cidade)+","+estado,cep
   * neste ponto imprimimos 3 ou 4 linha
   * dependendo se havia um campo compl
   * todas as etiquetas tem 6 linhas
   IF compl:1
      *  4+2 = 6
      ?
      ?
   ELSE
      *  3+3 = 6
      ?
      ?
      ?
   ENDIF compl:1
   * proxima etiqueta
   SKIP
ENDDO
SET TALK ON
*
* fim do arquivo ETQ.CMD
```

O primeiro exemplo está com o arquivo de dados original SOCIOS. Não há campo COMPL aqui.

```
. USE SOCIOS
. DO ETQ
Mila Moreira
R.Elma, 111
Sao Paulo,SP 92109

Ana Amorim
R.Tres, 333
Natal,RN 10010

Sonia Simoes
R.Para, 831
Belem,PA 21105

Monica Medeiros
R. Diva, 2219
Curitiba,PR 44110
```

```
Pedro Penedo
R.Sao Luiz, 22
Sao Paulo,SP 92109

Zilda Zorzi
Rua Pilar, 44
Aracaju,SE

Carlos Covas
R. "A", 88
Sao Paulo,SP 92101

Davi Dunas
Rua das Flores, 123
Aracaju,SE 36222

Ires Imare
R.Neusa, 99
Belem,PA 21233

Henrique Hipolito
R.Doze, 21
Sao Paulo,SP 92111
```

Para o segundo passo, criamos um arquivo de dados temporário igual a SOCIOS, exceto que tem o campo COMPL. Para a demonstração, quatro registros possuem dados dentro dos campos compl. ETQ.CMD nos forneceria três linhas de texto em uma etiqueta de seis linhas para todos os compl em branco e quatro linhas de texto em uma etiqueta de seis linhas para os quatro registros com COMPL:

```
. LIST SOBRENOME,ENDERECO,COMPL,CIDADE
00001  Moreira       R.Elma, 111          Suite 1    Sao Paulo
00002  Amorim        R.Tres, 333          Conj. A    Natal
00003  Simoes        R.Para, 831                     Belem
00004  Medeiros      R. Diva, 2219                   Curitiba
00005  Penedo        R.Sao Luiz, 22       Apt. 232   Sao Paulo
00006  Zorzi         Rua Pilar, 44                   Aracaju
00007  Covas         R. "A", 88                      Sao Paulo
00008  Dunas         Rua das Flores, 123  Sala 17    Aracaju
00009  Imare         R.Neusa, 99                     Belem
00010  Hipolito      R.Doze, 21                      Sao Paulo
```

```
. DO ETQ
. GOTO TOP
. DO ETQ
Mila Moreira
R.Elma, 111
Suite 1
Sao Paulo,SP 92109

Ana Amorim
R.Tres, 333
Conj. A
Natal,RN 10010

Sonia Simoes
R.Para, 831
Belem,PA 21105

Monica Medeiros
R. Diva, 2219
Curitiba,PR 44110

Pedro Penedo
R.Sao Luiz, 22
Apt. 232
Sao Paulo,SP 92109

Zilda Zorzi
Rua Pilar, 44
Aracaju,SE

Carlos Covas
R. "A", 88
Sao Paulo,SP 92101

Davi Dunas
Rua das Flores, 123
Sala 17
Aracaju,SE 36222

Ires Imare
R.Neusa, 99
Belem,PA 21233

Henrique Hipolito
R.Doze, 21
Sao Paulo,SP 92111
```

RESUMO

Neste capítulo, fomos apresentados ao controle de estruturas do dBase e vimos programas práticos em dBase construídos fora dele. A estrutura de um sistema dBase foi esquematicamente desenhada e discutimos um pouco sobre estilo de programação.

Os dois próximos capítulos concentram-se mais em técnicas fundamentais para programar

CAPÍTULO

7

Programação de Entrada/Saída

Como os exemplos de programação no capítulo anterior mostraram, os comandos e controles de estruturas podem ser usados para escrever programas efetivos e elaborados que, quando chamados, executam a tarefa e voltam o controle para o usuário. Este tipo de programação é mais apropriado para escrever utilitários que limpam e mantêm os arquivos de dados de uma maneira específica. Porém, a maioria dos programas necessita de um maior direcionamento do que os utilitários. Se nosso objetivo for confeccionar etiquetas para correspondência de um arquivo de dados completo, o ETQ.CMD é tudo que precisamos, mas o que acontecerá se desejarmos realizar algo mais trabalhoso:

```
. DO etq FOR cidade = "Sao Paulo"   (*Não vai funcionar nunca! *)
```

O comando acima não possui nenhuma chance de funcionar – tente se quiser – mas a idéia dele é importante. Muitas vezes queremos especificar condições de operação para os comandos em nossos programas, assim como uma maneira de conversarmos com eles.

Da mesma forma, os programas respondem ao usuário. Eles fornecem todos os tipos de mensagens auxiliares sobre o que está acontecendo, qual a entrada que esperam e quais opções usar. Os bons programas ocasionalmente fazem barulho. O alarme toca quando estão sendo operados com uma tarefa mais longa ou estão com problemas.

ENTRADA/SAÍDA PELO TECLADO E VÍDEO

Todas as características mencionadas acima relacionam-se ao **Console de Entrada/Saída**, isto é, entrar através do teclado e sair na tela. Neste capítulo, trataremos destas Entradas/Saídas de dentro dos programas dBase.

Novamente "WAIT TO" – Entrada de um Único Caractere

Não há melhor oportunidade de iniciarmos a discussão de E/S do que com "WAIT TO", um comando que já usamos extensivamente no Capítulo 6. Como uma conclusão, o comando WAIT fornece a mensagem "WAITING" na tela e cessa a execução do programa até que uma tecla — qualquer uma — seja pressionada. Se quisermos cessar o programa temporariamente por qualquer razão, o comando WAIT é o mais recomendável. Por outro lado, WAIT TO é um comando de E/S verdadeiro. Quando o interpretador do dBase encontra "WAIT TO < nome da varmem >" ele pára a tela e grava o próximo caractere digitado na variável de memória nomeada no comando. Isto é tudo que ele faz, mas é o suficiente.

Dentro de nossos programas podemos realizar o que quisermos com o caractere gravado. Freqüentemente o comparamos com a lista de caracteres gravada numa tabela CASE, para verificar qual o próximo passo do usuário, mas certamente não estaremos limitados a isto. Consideremos o seguinte fragmento de programa:

```
* limpar a tela e mostrar uma mensagem
ERASE
? "Qual o arquivo que podemos usar?"
WAIT TO nome
*Se o caractere digitado for X, usaremos "X.DBF"
USE &nome
RELEASE nome
```

A codificação acima é perfeitamente aceitável. O programador está propenso a limitar a entrada para o nome de arquivo às 26 letras do teclado. Talvez imaginário, mas WAIT TO aceita um caractere único como entrada, que pode ser usado onde quer que os dados de caracteres sejam apropriados.

Alguns programadores não gostam da mensagem WAITING que o dBase coloca na última linha da tela antes de continuar a operação. É verdade que esta mensagem é invariável e seria melhor se fosse centralizada na tela em vez de aparecer do lado esquerdo. De qualquer maneira, aqueles que não apreciam a mensagem podem fechar a tela com um comando SET CONSOLE OFF acima do WAIT, e um SET CONSOLE ON abaixo dele. Isto causa o mesmo efeito de um WAIT "não comprovado" sem a mensagem WAITING.

Não há nada de errado com esta habilidade, mas eu a omiti dos meus programas por uma razão: o dBase tem de ler o comando WAIT do arquivo .OVR, levando a um acesso ao disco e a uma demora. Quando desativarmos o console, o usuário quase sempre digitará a tecla para a seleção do menu antes que WAIT tenha tempo de aparecer, e, já que não há mensagem, o usuário terá a impressão de que o programa se perdeu.

Pelo fato de não gostar que as pessoas pensem que meus programas são mais lentos do que na verdade são, deixo os WAITINGs aparecerem. Desta maneira, o usuário possui um aviso visual que é a hora de dar a entrada.

Novamente ? e ?? – Saída de Console

Se WAIT é comando fundamental de entrada, o "?" é o de saída. Revendo, "?" pode ser lido como "imprimir" e "??" pode ser lido como "imprimir na mesma linha". Já que é importante que o material impresso na tela não desapareça antes que o usuário possa ler, os comandos de impressão são freqüentemente usados junto com WAIT, como em:

```
? "        alguma mensagem"
?         (*espacejamento estetico*)
WAIT      (* para a mensagem*)
```

TEXT e ENDTEXT – Saída de Telas

Ocasionalmente, um programador desejará apresentar uma ou mais telas de mensagens digitadas para um usuário durante o curso do programa. Na versão 2.4 do dBase esta tarefa é realizada muito mais facilmente pelos comandos TEXT e ENDTEXT, que podem ser usados para instruir o interpretador para enviar uma saída textual na tela:

```
TEXT

    Toda esta mensagem sera apresentada sem modificacoes
    na tela, pelo dBase. Ela sera emitida sem interrupcoes,
    a menos que um comando WAIT seja usado junto

ENDTEXT
```

Estas telas de texto poderiam ser criadas por um processador de texto, usando-se o modo de não documento, especialmente em MS-DOS, já que os sinais internos deixados por alguns processadores de texto como o WordStar podem possuir efeitos inconvenientes em uma tela de texto. Às vezes, por exemplo, uma tela de texto escrita no modo de documento trará, de repente, caracteres do alfabeto grego quando o dBase está em MS-DOS.

Telas de Auxílio

Uma das aplicações mais práticas dos comandos de texto e impressão está nos arquivos de auxílio na tela. Não estou me referindo às telas do comando HELP, disponíveis na versão 2.4 do dBase, e sim às telas de auxílio construídas – aquelas que escrevemos como programas – para um cliente, o usuário, a fim de tornar nossos sistemas mais fáceis de usar. Geralmente este processo consiste na entrada de um menu (ou entradas, se houver menus encaixados) oferecendo auxílio, como em:

```
DO WHILE (condicao)
   (*A impressao do menu aparece aqui*)
           .
           .
   ?
   ? "     H=Help     "  (*No menu*)
           .
           .
   WAIT TO fazer
   DO CASE
           .
           .
      CASE !(fazer)= "H" (*Na estrutura CASE*)
           DO help
           .
           .
   ENDCASE
ENDDO
```

De fato, este fragmento não funcionaria como foi apresentado aqui. Desta forma, podemos encaixar algumas seleções no menu e estrutura do case, a fim de que sejam oferecidas mais opções onde os pontos aparecem na codificação. Uma estrutura como esta ofereceria um menu pelo qual uma tela de auxílio poderia ser chamada.

Obviamente esta tela de auxílio é – ou melhor, teria sido – um arquivo de comandos localizados no drive default, nomeado HELP.CMD. Do contrário, quando o dBase procurar por ele, nós cairemos fora do programa com um DO CANCELED, COMMAND FILE help.cmd CALLED FROM xyz.cmd NOT FOUND.

Um acontecimento como este quase nunca ocorre em um programa testado, a menos que o programa se tenha tornado sofisticado a ponto de o disco default sofrer modificações em determinadas condições durante a execução do programa.

Assumindo que HELP.CMD foi chamado apropriadamente e que está lá no disco default, o restante torna-se um problema de bom senso. A idéia é apresentar uma ou mais telas impressas com informações úteis para o operador, segurá-las o bastante para serem lidas e então voltar ao programa no ponto da chamada. Habituei-me a escrever um ou dois parágrafos sobre o que realizei no final de cada seção de programação (por isso tornei-me tão disciplinado em meus projetos!) e estes arquivos de texto, mais tarde, tornaram-se o suporte de minhas documentações e a fonte de minhas telas de auxílio. Usei um processador de texto para isto, é claro, já que MODIFY COMMAND é inadequado.

Uma vez possuído o programa pronto, corto e colo (figurativamente) uma cópia dessas notas de programas até conseguir um arranjo de 80 colunas em espaço simples das informações mais importantes sobre o programa. Sempre isto pode ser feito em 40 linhas e raramente ultrapassam 100 linhas.

Num caso mais simples, com a versão 2.4, o arquivo de tela pode então ser dividido em telas separadas usando a rotina:

```
ERASE       (*Levar o cursor para o alto a esquerda*)
TEXT        (*Marcar o top*)
      O preenchimento da tela vai aqui, escrito no modo
      nao documento do WordStar.
ENDTEXT
WAIT        (*Mostra a tela para o leitor*)
ERASE       (*Iniciar o ciclo da tela 2*)
TEXT
      Mais telas de auxilio
        .
        .
        etc.
ENDTEXT
WAIT              (*E assim vai...*)
```

Telas de Auxílio sem "TEXT"

A técnica descrita anteriormente para telas de auxílio é muito boa, mas muitas pessoas não possuem a versão 2.4 ainda e, TEXT e ENDTEXT são irrealizáveis na 2.3. Além disto, o que acontece se um programa está à venda? Neste caso a compatibilidade é necessária.

Nestas situações, ainda uso WordStar para obter as telas de auxílio prontas, e então incluo uma etapa antes de inserir todos os ERASEs e WAITs. A etapa complementar é fazer uma pesquisa global e substituir cada tecla return com teclas returns combinadas com as declarações de impressão e delimitadores. Simbolicamente, é:

Pesquisar por		Substituir por
<cr>	→	]<cr>? [

Note que usei o delimitador colchete, porque me cansei de erros quando usava apóstrofos ou aspas dentro do texto de auxílio, obtendo linhas de impressão desastrosas como:

```
? '    Quero um copo d'agua'   (*Três apóstrofes *)
```

Qualquer processador de texto pode ser usado.

Em WordStar a pesquisa e substituição é:

```
1. ^QA
2. FIND? ^N <cr> (*Inserido com um ^P ^N *)
3. REPLACE WITH? ] ^N? [<cr>
```

Note também que os WAITS entre as telas podem ser considerados como pontos de bifurcação. Às vezes, um pequeno auxílio já é o bastante e não é preciso avançar sete telas. Não faz muita diferença manipular nossas telas em ordem decrescente de importância e colocar o seguinte fragmento de codificação entre elas:

```
ERASE
TEXT
        .
        .
        .
    ... fim da tela de auxilio um
    (Digite "R" para voltar,qualquer outra tecla
     para mais telas
ENDTEXT
WAIT TO mais
IF !(mais) = "R"
     RETURN
ENDIF
ERASE
TEXT
    A tela de auxilio dois comeca aqui
        .
        .
        .
```

Uma observação final: se todas as nossas telas de auxílio possuem valor em potencial iguais, e se há muitas delas é mais rápido para o usuário oferecer um menu de tela de auxílio do que tentar resolver com uma distribuição "melhor" (veja a Fig. 6-4).

ACCEPT – Entrada de String

Esta colocação sobre executar E/S de dentro dos programas foi limitada para entrada de caracteres único e saída de texto. Necessitamos também de um comando de entrada que ultrapasse um caractere em comprimento. ACCEPT é este comando permitindo a entrada de string de até 254 caracteres através do teclado.

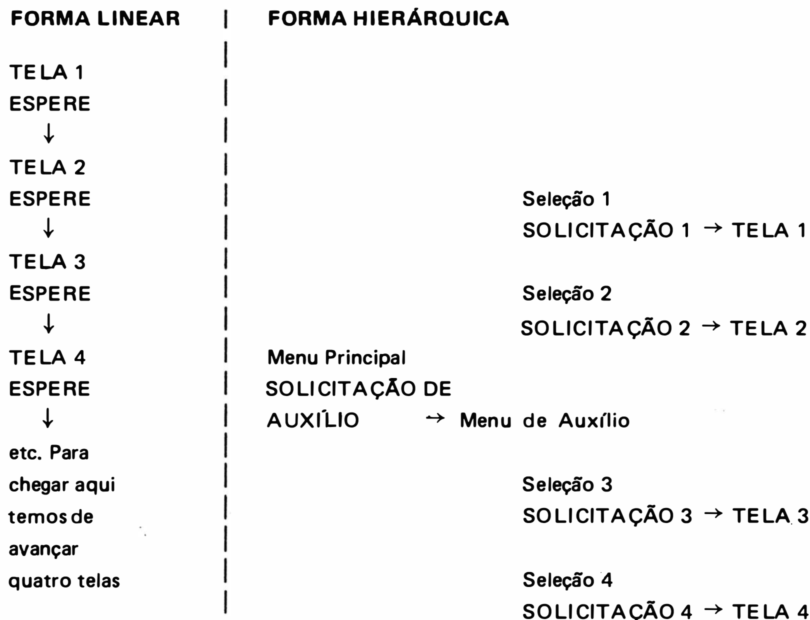


Figura 6-4 Distribuição linear *versus* hierárquica de telas de auxílio.

ACCEPT é um comando altamente versátil. A string de entrada é repetida na tela quando entra e pode ser editada com a tecla backspace. Ao pressionar return a entrada termina e a string é armazenada como um dado de caractere pelo nome da variável de memória designada. ACCEPT também permite a seleção de uma linha de mensagem para ser visualizada na tela à frente da resposta.

```
. ACCEPT <linha de indicação> TO <nome de varmem>
```

Vamos fazer uma experiência com ACCEPT usando um arquivo de comando de uma linha, ACC.CMD, que é:

```
ACCEPT "Esta é uma linha de mensagem" TO testar
```

Aqui, a nossa linha de mensagem é a frase, "Esta é uma linha de mensagem", que ao executar ACCEPT será colocada na tela. A string que o usuário inseriu será gravada na memória com o nome "TESTAR". Agora realizaremos a operação com ACC.CMD:

```
.DO ACC

Esta e uma linha de mensagem : Testar linha 1
          (* Digitamos "Testar linha 1"*)

  ? testar

 Testar linha 1   (*O dBase responde*)

? LEN(testar)  (*LEN() mostra o comprimento da string*)

 14    (*Nenhum espaco foi incluido pelo dBase*)
```

Embora ACCEPT seja reservado para a entrada de strings, se tivermos habilidade, podemos usá-lo para números. As macros funcionam para substituir string – onde quer que coloquemos uma macro válida, é como se tivéssemos digitado caracteres. Se usarmos ACCEPT para string de dígitos, embora eles sejam de tipo de dado caractere, podemos colocar a macro em um cálculo:

```
.DO ACC

Esta e uma linha de mensagem : 123
          (* Digitamos uma string numerica*)

. ? TYPE(testar)

 C        (*O dBase diz que nao e um numero*)

? &testar/2          (*Mas uma macro de uma string de numero*)

 61    (*e considerada como um numero*)
```

Definitivamente esta é uma proeza, mas não é para ser usada por pessoas precavidas nem em programas comerciais. Algo contra o que precisamos nos prevenir em relação a uma programação, é que boa parte de nossos usuários digitará bobagens em nossos programas e então nos culpará quando ele não funcionar:

```
.DO ACC  (*Agora vamos fazer um truque com o numero*)

Esta e uma linha de mensagem : 123 Elm Street
          (* O usuario se confundiu*)
```

```
. ? testar

 123 Elm Street      (*O dBase nao viu o problema ....*)

? &testar/2
*** SYNTAX ERROR ***  (*enquanto tentarmos aritmetica*)
       ?
? 123 Elm Street/2
CORRECT AND RETRY (Y/N)? N
```

Não é bom acusarmos nossos usuários de obstinados. Como programadores, é nosso dever prever entradas que poderiam confundir nosso programa e construir defesas às nossas codificações. Nunca abandone um comando ACCEPT, por exemplo, sem considerar o que poderia acontecer se o usuário digitasse uma linha de continuação ou espaços em branco. Se qualquer uma destas situações precipitar um erro, deveremos construir alguma proteção para o programa, talvez usando um teste de substring para brancos e a função TRIM( ) para eliminar os espaços.

ACCEPT e Condições de Processamento

Lembram-se de nosso comando "DO ETQ FOR <condições>"? Estamos prontos agora para usar ACCEPT a fim de escrever um arquivo de comando que produzirá este efeito.

Declaração do Problema: Queremos escrever um arquivo que confeccionará etiquetas de correspondência – para qualquer arquivo de dados e em registros selecionados naquele arquivo e em ordem especial. E queremos especificar tudo isto em tempo de processamento.

Pseudocódigo:
peça ao usuário o nome do arquivo de dados
peça as condições para a impressão das etiquetas
peça a ordem da impressão
coloque o arquivo de dados nomeado na ordem apropriada
copie para um arquivo de dados temporário usando a condição
use o arquivo de dados temporário
chame ETQ

O pseudocódigo está tão ligado ao programa que podemos avançar sem a etapa do refinamento:

```
***********************************************
* GERETQ.CMD
***********************************************
* Codificado para a impressao de etiquetas de
* correspondencia para registros selecionados
* sob uma condicao especifica
*
* O ARQUIVO DE DADOS DEVE TER UMA ESTRUTURA
* PARECIDA COM A DE SOCIOS.DBF
***********************************************
*
*  limpa a tela
ERASE
* consiga o arquivo de dados
ACCEPT " Entre com o nome do arquivo de dados"  TO arquivo
?
?
* pergunte a ordem
ACCEPT " Informe a ordem de classificacao " TO chave
?
?
* Obtenha as condicoes
ACCEPT " Quais os registros a imprimir? (Condicao) TO condicao
*
* escreva alguma coisa para ele ficar lendo
ERASE
?
?
? "        TRABALHANDO ...
* ao trabalho
USE &arquivo
INDEX ON &chave TO temp
COPY TO temp FOR &condicao
USE temp
DO etq
*
RELEASE ALL
* fim de arquivo GERETQ.CMD
```

Observou o potencial de erros na última parte do arquivo de comandos? O que aconteceria se não digitássemos nenhuma condição? Ou nenhuma chave? O dBase não pode realizar uma indexação em " ", e será um erro de sintaxe se o usuário apenas pressionar return. Para o nosso refinamento, entretanto, tentaremos:

```
(*Mesmos passos ate aqui*)
* ao trabalho
USE &arquivo
IF chave <> " "
   INDEX ON &chave TO temp
ENDIF  chave em branco
IF condicao = " "
   * pode ser, mas precisa ser um caractere para
   * ser uma macro
   STORE "t" TO condicao
ENDIF   condicao em branco
COPY TO temp FOR &condicao
USE temp
DO etq
```

O leitor agora seria capaz de incluir três ou mais linhas que ligariam a impressora em tempo de processamento se o operador solicitasse etiquetas impressas.

O Endereçamento do Cursor @yy,xx – Desenhando as Telas E/S

Provavelmente o método mais versátil de controlar a saída do dBase para a tela ou impressora é usar a declaração arroba ("@") para endereçar o cursor ou cabeça da impressora para as coordenadas exatas de partida para cada linha. Além disso, usar o comando @ devidamente requer estudo e responsabilidade.

A tela do dBase está dividida em um painel coordenado de 24 linhas por 80 colunas, com sua origem no canto mais alto esquerdo sendo designada como ponto 0,0. Pela convenção, quando fornecemos pontos na tela, a linha é especificada primeiro, em seguida a coluna.

Na Tela	*Fora da Margem*	
@ 0,66	@ 29,4	(*Linha > 23*)
@ 23,79	@ 22,84	(*Coluna > 79*)
@ 11,20	@ 11,80	(*Trunca depois de 79*)

Expressões podem ser usadas para fornecer coordenadas, desde que elas expressem valores numéricos. Quando usamos uma expressão, um "$" pode ser usado para simbolizar "da última coordenada". Porém, se uma expressão calcular uma posição de uma coordenada anterior, as posições negativas são proibidas:

Coordenadas Corretas	*Coordenadas Incorretas*	
@ num + 2,5	@ carac + 2,5	(*Somente números*)
@ $, $ + 5	@ $ -2,5	(*Nenhum negativo*)

Os dois verbos que podem ser usados com a @ são SAY e GET. SAY é usado apenas para saída, podendo ser enviado para a tela ou impressora e usado para texto subordinado ou conteúdos de campos ou variáveis. GET só pode ser usado com o console. Apresentaremos alguns exemplos do comando @ com SAY:

```
@ 5,5 SAY "O sobrenome do registro é"
@ 5, $ + 2 SAY 1:sobrenome
```

Se o indicador em SOCIOS estivesse no registro Amorim, ele sairia iniciando na linha 5, coluna 5:

```
O sobrenome do registro é Amorim
```

Os comandos @ podem aparecer em qualquer lugar dentro de um programa dBase – por exemplo, quase sempre eles representam a maneira mais conveniente para enviar rapidamente uma mensagem de linha única tal como "TRABALHANDO" ou "LIGUE A IMPRESSORA" para o centro da tela.

Porém, o dBase também possui precauções para um tipo de arquivo especial, chamado **arquivo de formatação** e terminando com o tipo de arquivo ".FMT", que contém os comandos arroba (e comentários). Estes arquivos de formatação podem ser chamados de qualquer lugar do programa para ativar a tela assim como qualquer outro tipo de módulo de programa.

O comando @ com um GET é usado para formatar a posição da tela pela qual o usuário editará um valor variável em tela-cheia, podendo ser uma variável de memória ou conteúdo de campo. Quando a tela surgir, GET mostrará o valor atual da variável. Quando fornecemos o comando READ, ativamos o GET para que o usuário possa trocar a variável. Sob o comando READ, GET fará uma substituição automática da variável pela entrada do usuário. O registro envolvido é automaticamente reindexado em todos os índices em uso, depois do READ. Se o campo que é sujeito ao GET não estiver envolvido na chave de indexação, o programador pode omitir a execução escondendo a indexação com o argumento "READ NOUPDATE". A tela pode ser organizada de qualquer maneira que agrade ao programador (incluindo sobre-impressão), e pode conter apenas um SAY ou GET por linhas de codificação, com esta exceção: um único GET pode ser colocado após um SAY como um deslocamento automático:

```
@ 11,5 SAY "Digite sobrenome" GET sobrenome (*Não há vírgula *)
```

Quando a saída for enviada à impressora, o comando usado será "SET FORMAT TO PRINT". Para desfazer isto, o comando é "SET FORMAT TO SCREEN". Se o formato for para a impressora, todas as coordenadas devem estar em ordem, da direita para a esquerda e de cima para baixo. GET não pode ser enviado à impressora. Os comandos SAY e GET não podem criar uma variável. Se a variável ainda não estiver criada pelo comando STORE, ou recuperada pelo RECALL FROM <arquivo mem>, ou ainda apresentada como um nome de campo de um arquivo de dados em uso, então SAY e GET provocarão um erro de sintaxe de uma variável indefinida.

Quando SAY e GET são usados sem um READ, a tela formatada escapa do monitor e o programa avança para a próxima etapa, sem pausas. Se não possuímos nenhum GET e queremos manter uma tela repleta de SAYs, usamos WAIT. Os GETs podem receber dados de qualquer tipo, porém, o tipo será o mesmo da variável que está sendo preenchida. Entretanto quando as variáveis são criadas para um GET, o programador deve ter certeza de que elas são do tipo e largura corretas, e se elas são numéricas, com o número correto de casas decimais. Quando usamos SAY e GET devemos sempre lembrar das três etapas do processo:

1. As variáveis de memória para os GETs devem ser criadas com um STORE.
2. Os SAYs e GETs podem ser formatados com um "SET FORMAT TO <nome do arquivo de formato>" (mencionados com mais detalhes neste capítulo).
3. Ele não será terminado sem um READ.

As Cláusulas USING e PICTURE

Incluindo a capacidade de endereçar o cursor para qualquer parte da tela com um comando @, a E/S com os comandos GET e SAY possuem uma outra grande vantagem sobre ACCEPT e a declaração de impressão ?. A entrada e saída de GET/SAY podem ser formatadas por cláusulas nas linhas de comando que irão colocar os dados na tela.

As cláusulas de saída usadas com SAY são as USING:

```
SAY <variável> USING "<string de carc>"
```

Os caracteres de controles USING são:

Caractere	Efeito
9 ou #	Próximo número (somente números)
A ou X	Próximo caractere
!	Sem efeito
$ ou *	Próximo dígito exceto para zeros à esquerda, que são substituídos por um $ ou *. Muito útil para proteção de escritas.

As cláusulas de entrada de telas usadas com GET são geralmente muito mais úteis. Sua palavra-chave é PICTURE:

```
GET <variável> PICTURE "<string de caracteres de tela>"
```

Os caracteres de controles são:

Caractere	Efeito
9 ou #	Aceita somente dígitos, um $ ou —, um ponto decimal e um espaço
X	Aceita qualquer caractere
A	Aceita somente caracteres alfa
!	Aceita qualquer caractere mas coloca em maiúscula todas as entradas alfa
$ ou *	Sem efeito — sai como está

Quando as cláusulas picture contêm outros caracteres que não sejam estes, eles atuam como texto subordinado e o cursor pula-os. Isto facilita a padronização de tais coisas como barras em datas:

```
@ yy, xx SAY "Digite a data" GET reg:data PICTURE"99/99/99"
```

No momento da entrada, o dBase exibirá ao usuário a figura mostrada na cláusula e colocará na tela o campo, protegido contra qualquer entrada exceto dígitos de 0 a 9. As barras serão saltadas pelo cursor mas entrarão no campo de data no arquivo automaticamente. A tela de entrada aparecerá com:

```
Digite a data: / / :
```

A SAÍDA NA IMPRESSORA USANDO @

Quando usamos o comando @ para enviar uma saída à impressora, o sistema de coordenadas é o mesmo que o da tela, exceto que o número de linhas de saída excede 24. O papel normal de 81/2 × 11 polegadas possui 66 linhas e 80 colunas em cada página. O comando @ pode endereçar todas elas. O comando para ativar a impressora é "SET FORMAT TO PRINT". Como em REPORT FORM, uma forma automática de alimentação é enviada à impressora tendo cada chamada do arquivo formatado, para limpar o restante do relatório anterior da impressora.

Também como com REPORT FORM, o comando "SET MARGIN TO <num>" pode ser usado para movimentar a margem do relatório para a esquerda ou direita de 0 a 254 — o default é 0. Uma margem de uma polegada possui oito espaços.

Exemplifiquemos enviando duas linhas de relatório para a impressora. O arquivo formatado quando preparado para a saída na tela seria:

```
*****************************************
* EXEMPLO.FMT
*****************************************
*
* apenas para mostrar como pode ser feito
@ 4,5 SAY sobrenome
@ $+2,5 SAY contrib
* mais saidas podem vir aqui
*
* fim do arquivo EXEMPLO.FMT
```

O que faremos agora, se desejarmos enviar para a impressora? Nosso relatório necessitará de uma nova folha de papel para cada registro, porque o primeiro endereço (4, 5) é absoluto. Há apenas um (4, 5) em cada folha de papel.

Queremos três registros em cada folha. Podemos movimentar o esqueleto do formato para um arquivo de comandos de relatório e incluir um deslocamento de 20 linhas para a coordenada-Y dentro de nosso programa. O deslocamento fará nossos registros imprimirem em Y = 4, Y = 24 e Y = 44. Uma segunda variável, "prnt", será 1 ou 0, dependendo se desejarmos que o deslocamento possua algum valor na expressão.

Agora, não podemos mais usar a convenção do "$" para indicar que uma linha será impressa após a última. A tela pode tolerar isto, mas um "$" enviado para a impressora pode possuir efeitos desfavoráveis. Na Epson, ele age como um tipo de superalimentação, pulando duas páginas entre as linhas.

Agora escreveremos uma rotina de chamada que ativa prnt e circula o deslocamento:

```
*****************************************
* EX2CHAM.CMD - imprime relatorio
*****************************************
* cada registro sairia numa pagina, a menos que ...
SET EJECT OFF
SET TALK OFF
* veja se o usuario quer imprimir
ERASE
@ 11,8 SAY "QUER IMPRIMIR O RELATORIO? S/N "
```

```
WAIT TO hard:copy
IF !(hard:copy) = "S"
   * imprima-o
   SET FORMAT TO PRINT
   STORE 1 TO prnt
ELSE
   * prnt precisa de algum valor
   STORE 0 TO prnt
ENDIF
ERASE
* comece o circuito de registros
DO WHILE .NOT. EOF
   *inicie a contagem
   STORE 0 TO offset
   * aqui vai o cabecalho da pagina
   @ 2,2 SAY "Relatorio Anual"
   * comece o circuito de contagem
   DO WHILE (offset <= 2) .AND. (.NOT. EOF)
      @ prnt*(20*offset) + 4,5 SAY sobrenome
                  (*@ pode usar expressoes*)
      @ prnt*(20*offset) + 6,5 SAY contrib
      * mais saidas pode vir aqui
      STORE offset + 1 TO offset
      * proximo registro
      SKIP
      * se nao quiser imprimir
      IF prnt = 0
         * ... entao e melhor parar a tela
         WAIT
         * limpar para a proxima tela
         ERASE
      ENDIF  prnt = 0
   ENDDO  (offset)
   * salta a pagina
   EJECT
ENDDO  saida
* logico, temos que limpar tudo
SET FORMAT TO SCREEN
SET TALK ON
* fim do arquivo  EX2CHAM.CMD
```

O ponto crítico deste programa está em duas linhas:

```
@ prnt* (20* offset) + 4,5 SAY sobrenome
@ prnt* (20* offset) + 6,5 SAY contrib
```

O fato de prnt ser 1 ou 0, indica se o deslocamento está ativado ou não. Quando o deslocamento for 0, 1 ou 2, os registros serão colocados no papel em blocos de três. Os princípios neste relatório podem ser completamente adaptados.

Criando e Usando Arquivos Formatados

Vamos fazer endereçamento do cursor funcionar, usando @, SAYs e GETs e criar um arquivo formatado adequado para usar com SOCIOS.

```
******************************************
* SOCIFORM.FMT - Para usar com a extrutura
*  do arquivo de cados SOCIOS.DBF
******************************************
@ 3,5 SAY "NOME " GET nome
@ 3,$+5 SAY "SOBRENOME " GET sobrenome
@ 5,5 SAY "ENDERECO " GET endereco
@ 7,5 SAY "CIDADE " GET cidade
@ 7,$+6 SAY "ESTADO " GET estado
@ 7,$+6 SAY "CEP " GET cep
* vamos deixar contribuicao e pagamento de lado
*
* usamos o comando @ para colocar avisos aqui,
* lembrando o usuario do controle do cursor, etc.
*
* fim do arquivo SOCIFORM.FMT
```

Este arquivo formatado ativará a tela parecendo com:

```
0    (*Números de linha, que não aparecerão*)
1
2
3    NOME :          :       SOBRENOME:          :
4
5    ENDEREÇO :          :
6
7    CIDADE :        :     ESTADO :   :      CEP : :
8
```

Uma vez que o arquivo formatado tenha sido colocado em uso com o comando "SET FORMAT TO <nome do arquivo formatado>" este formato será a figura da tela que o dBase apresentará para os APPENDs e EDITs. De fato, se um outro arquivo de dados é colocado em uso e não partilha os mesmos nomes de campos, o programador deve lembrar de trocar os formatos ou desativá-lo com o comando "SET FORMAT TO". Do contrário, o dBase procurará variáveis indefinidas e apresentará muitos erros de sintaxe.

ESTILO DE PROGRAMAÇÃO II – CAMINHO E ERRO, E SUPERPROGRAMAÇÃO

Compare os dois seguintes módulos de codificação – eles realizam a mesma coisa:

```
*******************************************
*  MODULO1 - Para appends com SOCIFORM
*******************************************
*
* prepare a memoria
STORE "         " TO Mnome
STORE "             " TO Msobrenome
STORE "                  " TO Mendereco
STORE "           " TO Mcidade
STORE "  " TO Mestado
STORE "     " TO Mcep
*
*  grave no disco
SAVE TO branco
* ative a configuracao
USE socios
SET FORMAT TO sociform
* inicio do circuito
DO WHILE t
   * preencha a memoria
   RESTORE FROM branco
   APPEND BLANK
   * pegue os dados de entrada do usuario
   READ
   * abandonar ou tem mais registros?
   IF Mnome = " "
      * esta em branco = vamos parar
      RETURN
   ENDIF (nome em branco)
   * coloque tudo no registro
   REPLACE nome WITH Mnome, sobrenome WITH Msobrenome
   REPLACE endereco WITH Mendereco, cidade WITH Mcidade
```

```
   REPLACE estado WITH Mestado, cep WITH Mcep
* este esta pronto - pegue outro
ENDDO
* restaure a configuracao
RELEASE ALL
*
* fim do arquivo MODULO1
```

Percebeu a figura? Agora observe isto:

```
***********************************************
*   MODULO2 - tambem para appends com SOCIFORM
***********************************************
*
*  ative a configuracao
USE socios
SET FORMAT TO sociform
* faca-o
APPEND
*
* fim do arquivo MODULO2
```

Alguém adivinharia qual módulo eu prefiro? Se há uma maneira simples para fazer algo complicado, eu geralmente a procuro.

Isto nos conduz a um ponto estilístico neste capítulo. Não superprograme! Sempre pergunte se há uma maneira mais simples. De fato, se pensamos em uma programação como desafio e apenas nos surgiu uma idéia, não estamos prontos para codificar. E aqui está o nosso segundo ponto: seja um experimentador.

Quando iniciar a escrita de um programa e quiser fazer algo em três etapas, tais como:

```
STORE contrib TO Mcontrib
STORE 1.06 * Mcontrib TO ctrb:n:tx
REPLACE contrib WITH ctrb:n:tx
```

vá ao modo interativo primeiro e não faça rodeios:

```
REPLACE CONTRIB WITH 1.06 * CONTRIB
```

O que você tem a perder? Evite a superprogramação.

A CONFIGURAÇÃO DO dBASE

O dBase tornou-se a linguagem escolhida em muitas das empresas por causa da rapidez com que um programa pode ser desenvolvido. O tempo de desenvolvimento de um programa é o aspecto mais caro do software hoje, e o dBase paga por si só este tempo, pois permite aos clientes realizarem o programa em semanas em vez de meses. Mas o sistema possui limitações especialmente em velocidade de execução, uma vez que os programas são operacionais. Linguagens interpretadas operam mais devagar comparadas com as linguagens compiladas. Além disso, a codificação de um programa-fonte geralmente é espalhada pelo disco em arquivos de texto pequenos. Durante uma programação, um quarto ou metade da operação é reservada para o acesso ao disco — abrindo e fechando arquivos de texto, arquivos de dados, arquivos de índice e outros.

A resposta mais simples para este problema é usar um drive de disco rígido com o dBase. A velocidade de acesso dos discos rígidos humilha os discos flexíveis mais rápidos e suas tremendas capacidades de armazenamento são também ideais para aplicações comerciais. Mas não são todas as pequenas operações que podem contar com um disco rígido. As limitações de velocidade, quando usamos discos flexíveis em vez de rígidos, tornam programas elaborados ou complexos impraticáveis. A cada nova entrada, o usuário cansa de esperar pelo programa enquanto ele está atualizando índices múltiplos ou trazendo uma dentre várias telas. O tempo de resposta é o mais prejudicado.

Tentar conseguir todas as operações comerciais no modo interativo também não é a solução. O modo interativo do dBase possui desvantagens de ser difícil de aprender e um tanto perigoso de manipular. Um "DELETE ALL FOR <condição>", por exemplo, pode levar a resultados não desejáveis quando um operador inexperiente escolhe os parâmetros. Mesmo um operador experiente em dBase pode eliminar todo um banco de dados se não tomar o cuidado necessário.

Tenho uma solução a recomendar, que chega a ser quase uma filosofia. Entre o modo interativo e a programação elaborada há um meio-termo no qual o dBase se situa como mais poderoso e produtivo.

Encontrando um termo melhor, eu chamo isto de configuração alternada, e significa simplesmente que, para a maioria dos trabalhos diários, o dBase pode ser operado no meio do caminho entre o modo interativo e um programa prestes a rodar. Para aqueles que estão tentando operar comercialmente usando o dBase com discos flexíveis, aqui estão algumas normas:

1. A organização inicia ao nível do disco; isto é, tendo um disco por programação/período. Exemplo: para processamentos mensais, um disco para cada mês; a mais, um disco para cada ano.

2. Criar programas simples para ajudar a transcrição de informações na entrada de dados e outras rotinas, usando lotes orientados. Quando menos habilidoso o pessoal, mais elaborado o programa precisa ser. Realizando digitações em lotes, podemos ter certeza de que as pessoas podem criticar seus próprios discos.

3. Criar rotinas de utilitários bem nomeadas e generalizadas para ajudar o pessoal de nível médio na manipulação e uso de bancos de dados. Estes utilitários podem servir para

qualquer objetivo, mas algumas necessidades óbvias serão programas que limparão e verificarão os arquivos de lote e então os incluirão ao banco de dados.

4. Criar ou assumir a posição de processador de dados ou de diretor de sistemas.
5. Criar relatórios periódicos e auditorar relatórios constantes em uma forma padronizada com revisão regular pelo diretor do sistema. Isto estende-se aos backups de cópias de discos rígidos, desde que fornecidos com oportunidade para intervenção. Revisões periódicas de bancos de dados, relatórios e utilitários pelo Diretor de Sistemas.
6. Nenhuma outra pessoa, mas somente o diretor de sistemas,

   SEMPRE: Alterará os relatórios
   Alterará ou escreverá novos utilitários
   Usará o comando PACK.

   Há uma sétima norma que se refere mais a uma filosofia pessoal e é argumentável:
7. Um esforço constante deve ser feito para melhorar todas as habilidades dos usuários, movimentando-os da entrada de lotes para um nível médio e assim por diante. Esta norma é baseada por acreditar que os usuários se tornam mais úteis ao sistema (e menos perigosos ao banco de dados) quando melhoramos suas habilidades.

Obviamente, já que cada configuração de um programa solicita seus próprios utilitários a serem desenhados, uma configuração ideal necessita de planejamento e desenvolvimento. Mas eu fico cético quando vejo um cliente contratar um programador para resolver todos os problemas com os seus dados. O que faz o programador ser tão inteligente? E tenho visto grandes sucessos quando um comerciante decide fazer do dBase um instrumento para o seu pessoal.

EXEMPLO DE UTILITÁRIOS COM CONFIGURAÇÃO dBASE

Os pequenos utilitários deveriam ser usados onde fosse possível reduzir chances de entrada de dados sem índices corretos em uso, e assim por diante.

Apresentaremos alguns exemplos.

GO.CMD – Para Inserir o dBase

Este pequeno utilitário apenas inicia o dBase com data correta e o drive do default:

```
****************************************
* GO.CMD  -  Iniciando o dBase
****************************************
STORE [        ] TO mDATE
ERASE
@ 11,20 SAY [DATE()? ] GET mDATE PICTURE [99/99/99]
READ
```

```
IF $(mDATE,1,1) <> [ ]
   SET DATE TO &mDATE
ENDIF
ERASE
@ 11,20 SAY [DISCO DEFAULT?]
?

WAIT TO mDISCO
IF mDISCO <> [ ]
   SET DEFAULT TO &mDISCO
ENDIF
RELEASE ALL
SET INTENSITY OFF
* fim do arquivo GO.CMD
```

Eliminando Campos pela Consistência

Um segundo tema que recorremos em relação aos utilitários é a correção de campos depois da entrada em lotes para TEMP.DBFs e antes dos APPENDs. Consideremos um arquivo de dados de contabilidade, OUTGO, no qual os campos DATA e CATEGORIA devem estar devidamente formatados para serem usados nos índices e classificações. CATEGORIA teria letras maiúsculas e DATA teria barras (' / ') como delimitadores. Porém, em uma entrada em lote (ou corrida), é mais rápido ignorar estas exatidões. Do contrário, quem pode garantir a entrada perfeita de dados?

```
. USE OUTGO
. DISP STRU
STRUCTURE FOR FILE:  B:OUTGO    .DBF
NUMBER OF RECORDS:   00000
DATE OF LAST UPDATE: 00/00/00
PRIMARY USE DATABASE
FLD        NAME      TYPE WIDTH   DEC
001      DATA         C    008
002      VALOR        N    007    002
003      CATEGORIA    C    002
004      OBSERV       C    030
005      FATURA       L    001
006      CLIENTE      C    012
007      HISTORICO    C    010
** TOTAL **             00071

. LIST
00001  11 22 33     9.00 bc Custos Faturados              .T. PSM
```

```
00002  04*08(82     4.00 ex Despesas                    .F.
00003  12#13#82    12.00 bc Mais faturas                .F.
```

Para consertarmos os delimitadores de DATA e as entradas de categoria, escrevemos um utilitário chamado COMB.CMD.

```
*********************************************
*  COMB.CMD
*  codificado para consertar categoria e datas
*********************************************
GOTO TOP
DO WHILE .NOT. EOF
   REPLACE categoria with !(categoria)
   REPLACE data WITH $(data,1,2)+"/"+$(data,4,2)+"/"+$(data,7,2)
   SKIP
ENDDO
```

Com o OUTGO em uso, vamos usar DO COM. Note que o comando talk não foi desativado neste utilitário. De certa maneira, ele fornece proteção contra erros, fazendo comentários durante a operação:

```
. DO COMB
00001 REPLACEMENT(S)
00001 REPLACEMENT(S)
11/22/33
RECORD: 00002
00001 REPLACEMENT(S)
00001 REPLACEMENT(S)
04/08/82
RECORD: 00003
00001 REPLACEMENT(S)
00001 REPLACEMENT(S)
12/13/82
RECORD: 00003

. LIST
00001  11/22/33     9.00 BC Custos Faturados            .T. PSM
00002  04/08/82     4.00 EX Despesas                    .F.
00003  12/13/82    12.00 BC Mais faturas                .F.
```

Poderíamos demonstrar mais exemplos. Em geral, os utilitários podem:

1. Ativar a configuração (defaults, intensidade, alarme, comunicação, ligar impressora etc).
2. Limpar ou checar um arquivo de lotes antes de usar um APPEND para um .DBF principal.
3. Desenvolver uma rotina que ligue dois .DBFs.
4. Pesquisar um .DBF para entradas variáveis ou incorretas.
5. Pedir e escrever dados para usar com outros programas CP/M ou MS-DOS (WordStar, SuperCalc, 1-2-3 etc.).

CAPÍTULO

8

Relacionando Arquivos de Dados

No Capítulo 6, apresentamos a imagem do interior da memória de um computador operando o dBase, que incluía a área primária e a secundária, reserva para variáveis de memória, e um interpretador que abria e processava os arquivos de comandos. Até agora, todos os aquivos de comandos fornecidos nos exemplos operaram somente em arquivos de dados na área primária.

Embora os programas dBase que trabalham somente com um arquivo de dados possam ser muito úteis, para uma grande maioria representaria apenas a metade de um sistema gerenciador de banco de dados, se não existisse uma área secundária para colocar outro arquivo de dados. De fato, com dois arquivos, simultaneamente na memória, novos horizontes de manipulação de dados abrem-se para o programador.

ENDEREÇANDO A ÁREA SECUNDÁRIA

Todos os comandos para usarmos na área secundária são exatamente os mesmos da área primária. As duas áreas podem ser consideradas como imagens de espelho uma da outra, ou até, mais precisamente, como almas gêmeas. De fato, a única coisa "primária" sobre a área primária é que, quando chamamos o dBase (e depois de um CLEAR), o sistema leva o default para a área primária.

A movimentação entre as duas áreas é realizada pelos comandos:

```
. SELECT SECONDARY  (*Iremos à secundária... *)
. SELECT PRIMARY    (*e voltaremos. *)
```

Como mencionamos acima, o comando CLEAR também traz o sistema de volta à área primária, onde quer que esteja.

Nenhum outro comando troca a área que o dBase está, por isso o programador deve lembrar-se disso quando usar CANCEL ou RETURN. A causa de erros freqüentes em dBase ocorre quando um arquivo de comando se desloca para a área secundária e volta ao arquivo chamador sem deslocar-se de volta. Imagine os efeitos de um SKIP ou DELETE se estivermos na área errada.

Na verdade, quando observamos os exemplos dos programas dBase que utilizam as duas áreas, podemos notar que há uma tendência nos programas para retornarem o controle à área primária. Eu recomendo esta prática. Não há nada de errado com a área secundária – ela funciona tão bem quanto a primária – mas o hábito de voltar para a área primária é o mesmo que conservar-se à direita em uma estrada quando não pretendemos realizar ultrapassagem. Sempre saberemos onde estamos em caso de emergência.

REGRAS PRIMÁRIAS E SECUNDÁRIAS

As regras para manipularmos as áreas primárias e secundárias são simples:

1. Podemos ler as duas áreas bem como a memória, a partir de qualquer área, mas só podemos alterar, movimentar ou manipular o arquivo na área em que estamos. Se desejarmos alterar o arquivo na outra área, teremos de usar um "SELECT <outra área>" para irmos lá. A memória pode ser alterada a partir de qualquer área.
2. Se dois campos de arquivos de dados em áreas diferentes possuírem o mesmo nome, eles podem ser distinguidos colocando um "p." (primária) ou "s." (secundária) quando solicitamos o seu valor. O prefixo é incluído sem espaços, como em "p.SOBRENOME". Quando não usarmos os prefixos p. ou s. para distinguir o mesmo nome de campo, o valor obtido será o do arquivo de dados na área em que nos encontramos. Esta convenção vale apenas para acessar valores e não pode ser usada para trocar os conteúdos dos campos (veja o item 1).

Note que é perfeitamente possível gravar uma variável à memória a partir da outra área – é um ato de leitura da outra área, que não apresenta problemas. O que *não* podemos fazer nas outras áreas são as linhas de comando INDEX, SKIP, LOCATE, FIND, REPLACE, DELETE ou nada que comande movimentos ou escritas ao arquivo de dados.

UMA OPERAÇÃO PRIMÁRIA/SECUNDÁRIA

Como exercício, vamos montar dois arquivos de dados, um na área primária e outro na secundária e realizaremos algumas manipulações. Apresentaremos um novo arquivo de dados aqui, chamado "BICHOS.DBF", que lista os animais de estimação de nossos registros em SOCIOS.

```
A>DBASE
. USE SOCIOS   (*estamos na primária por default *)
. SELECT SECONDARY
. USE BICHOS
. SELECT PRIMARY
. DISPLAY STATUS   (*preste atenção no status *)

DATABASE SELECTED - B:SOCIOS  .DBF
PRIMARY USE DATABASE


UNSELECTED DATABASE - B:BICHOS  .DBF
SECONDARY USE DATABASE

(*vamos pular os SETs da segunda parte do status *)

. DISPLAY STRUCTURE  (*ainda na primaria*)
STRUCTURE FOR FILE:  B:SOCIOS  .DBF
NUMBER OF RECORDS:   00005
DATE OF LAST UPDATE: 00/00/00
PRIMARY USE DATABASE
FLD       NAME       TYPE WIDTH   DEC
001       NOME        C    010
002       SOBRENOME   C    015
003       ENDERECO    C    020
004       COMPL       C    010
005       CIDADE      C    010
006       ESTADO      C    002
007       CEP         C    005
008       CONTRIB     N    005     002
009       PAGAMENTO   L    001
** TOTAL **              00079

. GOTO TOP  (*top de quem? — Da primária *)
. LIST NEXT 3
00001 Mila    Moreira   R.Elma, 111       Suite 1    Sao Paulo  SP 92109   5.00 .T.
00002 Monica  Medeiros  R. Diva, 2219                Curitiba   PR 44110   5.99 .T.
00003 Pedro   Penedo    R.Sao Luiz, 22    Apt. 232   Sao Paulo  SP 92109   0.00 .F.

. DISPLAY    (*Quem está apontado? *)
00003 Pedro   Penedo    R.Sao Luiz, 22    Apt. 232   Sao Paulo  SP 92109   0.00 .F.

. SELECT SECONDARY
```

```
. DISPLAY STATUS
(* Note que o selecionado agora é BICHOS *)

DATABASE SELECTED - B:BICHOS  .DBF
SECONDARY USE DATABASE


UNSELECTED DATABASE - B:SOCIOS  .DBF
PRIMARY USE DATABASE

(*outra vez pulamos os SETs *)



. DISPLAY STRUCTURE
STRUCTURE FOR FILE:  B:BICHOS  .DBF
NUMBER OF RECORDS:  00004
DATE OF LAST UPDATE: 00/00/00
SECONDARY USE DATABASE
FLD       NAME      TYPE WIDTH   DEC
001     SOBRENOME    C    010
002     TIPO         C    008
003     NOME         C    010
** TOTAL **             00029

. LIST NEXT 3
00001 Penedo    cachorro Fido
00002 Zorzi     gato     Arthur
00003 Simoes    pombo    Apollo

. SKIP   (*Este LIST ou SKIP afetam a primária? *)
RECORD: 00004

. DISPLAY  (*Ainda na secundária *)
00004 Medeiros  cachorro Bola

. SELECT PRIMARY
. DISPLAY  (*Claro que não *)
00003 Pedro     Penedo     R.Sao Luiz, 22      Apt. 232   Sao Paulo  SP 92109   0.00 .F.
```

Para demonstrarmos o uso dos prefixos p. e s., realizaremos uma minioperação. Note que o conteúdo de um campo do arquivo não selecionado pode ser armazenado na memória.

```
A>dbase

. USE SOCIOS
. DISPLAY  (* O registro apontado na primária *)

00001  Mila    Moreira    R.Elma, 111    Suite 1    Sao Paulo  SP 92109   5.00 .T.

. SELECT SECONDARY
. USE BICHOS
. DISPLAY  (* O registro apontado na secundária *)
00001  Penedo      cachorro Fido

. ? sobrenome   (* O default - sobrenome nesta área *)
Penedo

. ? p.sobrenome   (* Este vem da primária *)
Moreira

. STORE p.sobrenome to dono   (* ação da leitura *)
Moreira

. ? dono
Moreira     (* variáveis podem ser criadas a partir de área não selecionada *)
```

UNINDO DOIS ARQUIVOS

Quando combinamos vários arquivos de dados que guardam informações sobre um assunto, criamos um banco de dados verdadeiro. Pelo fato de o dBase nos fornecer as áreas primárias e secundárias nas quais carregamos os arquivos, e também permitir usar linhas de codificação nos arquivos de comando para manipular estes arquivos, somos capazes de escrever programas que construam bancos de dados relacionais.

Há três maneiras lógicas para os arquivos nas áreas primárias e secundárias serem relacionadas. Primeira: pode ser uma estrutura de um-para-um, na qual cada registro na área primária possui um e apenas um registro correspondente na secundária. Esta estrutura é raramente usada, exceto na situação especial onde o limite de 32 campos não é o bastante para um único registro e a estrutura do registro estende-se para as duas áreas. Segunda: pode haver uma situação um-para-muitos, onde cada registro na área primária possui de zero a muitos registros correspondentes na secundária (veja Figura 8-1). Um exemplo de situação para este desenho é um programa de venda de livros, onde um arquivo de clientes na primária seria relacionado com um arquivo de pedidos na secundária. Terceira: seria uma estrutura de muitos-para-um, na qual muitos registros no arquivo primário fossem relacionados a cada registro na secundária. Tal situação ocorre em **tabelas de consulta.**

AS TABELAS DE CONSULTA

Discutiremos com detalhes as tabelas de consulta porque elas representam um grande recurso do programador que pode desenvolver muito trabalho sem esforçar-se tanto, uma técnica de programação muito usada em dBase. A idéia é direta, existem situações nas quais ao se conhecer os conteúdos de um campo em um registro é o suficiente para preenchermos muitos outros campos. Por exemplo, se o registro de um cliente possui um único código de cliente, então o nome, endereço, cidade e telefone podem ser procurados através de um índice. Da mesma maneira, um número de peça em um inventário seria o suficiente para encontrarmos a sua descrição e o preço atual.

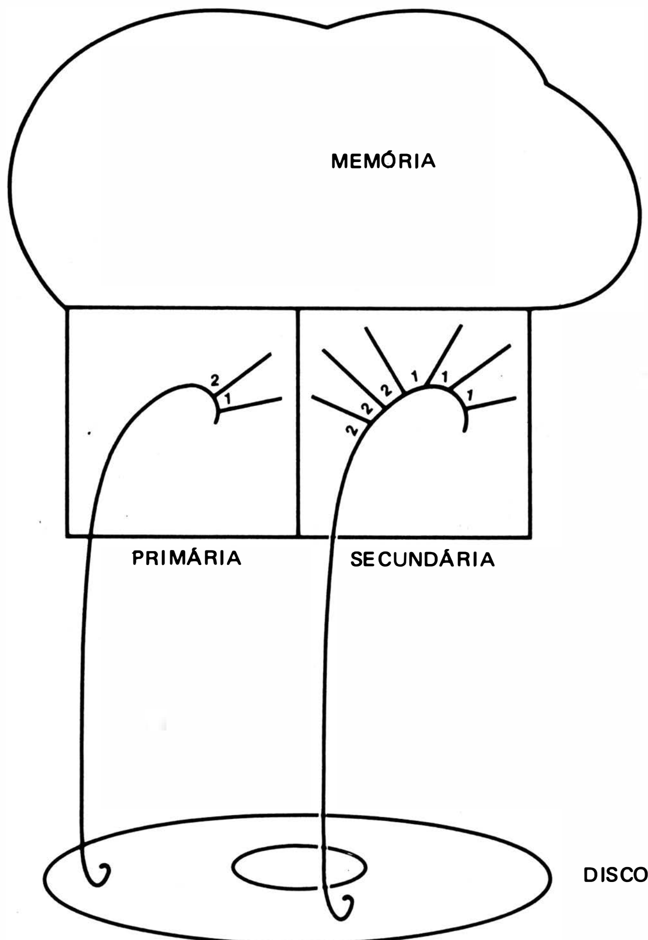


Figura 8-1 Em uma relação de um-para-muitos, cada registro do arquivo de dados na área primária é encaixado em vários registros no arquivo de dados na área secundária.

As tabelas de consulta são simplesmente arquivos de dados dBase consistindo em registros com dados tais como descrição das peças ou nomes e endereços dos clientes, indexados em alguma união, como o número da peça ou o código do cliente. Sempre que uma linha completa de informação for necessária do outro banco de dados, um programa pode ser usado para procurar os dados corretos e preencher o registro de acordo com a união.

Na Figura 8-2, o banco de dados na área primária foi preparado para ter seus campos vazios preenchidos com DAD1 e DAD2 na tabela de consulta na área secundária.

Figura 8-2 Um banco de dados na área primária com campos vazios a serem preenchidos com os dados na tabela de consulta.

As uniões podem ser qualquer coisa única associada ao dado, como números da previdência social para pessoas ou números de estoque para inventário, mas eles devem ser armazenados em campos de caracteres porque o arquivo de banco de dados de pesquisa deve ter a capacidade de realizar um FIND em uma indexação do campo da união na tabela de consulta.

EXEMPLO 1 – ENVIANDO DESCRIÇÕES A PARTIR DO NÚMERO DAS PEÇAS

Vamos considerar um banco de dados primário usado por uma fábrica para controlar as transações rejeitadas (TRs) submetidas a um quadro de revisão de materiais (QRM). Este módulo é de um programa verdadeiro, usado pela divisão do controle de qualidade de um de meus clientes.

Sem entendermos mais nada sobre este banco de dados, está claro que um campo no registro, o número da peça, pode ser usado para procurar o valor de dois campos dependentes, a descrição da peça e o seu valor unitário. Sabendo o número da peça, saberemos as outras duas. O número da peça, desta forma, representa um candidato à união.

```
STRUCTURE FOR FILE:  B:QRMLOG  .DBF
PRIMARY USE DATABASE
FLD       NAME      TYPE WIDTH   DEC
001      NO:RT       C    004
002      DATA:RT     C    008
003      NO:PECA     C    007              (*Uniao*)
004      DESCRICAO   C    020              (* Dado dependente *)
005      QTD:REC     N    005
006      QTD:REJ     N    005
```

```
007     INST:DATA    C   008
008     INST:DISP    C   008
009     VAL:UNIT     N   007    002    (* Dado dependente *)
010     VAL:TOTAL    N   008    002
011     FOLLOW:UP    C   016
012     COMP:DATA    C   008
013     MDIA         N   003
** TOTAL **             00108
```

Obviamente, uma tabela de consulta que utiliza esta união para preencher o registro teria também um número de peça como um campo, bem como campos para os dados dependentes, ambos indexados pela união. Neste caso, um campo adicional foi incluído para a data da última edição do registro na tabela de consulta. Isto ocorre porque o valor unitário em cada peça mudará com o tempo e, pela colocação das datas de edições, podemos obter dados de posições anteriores.

```
STRUCTURE FOR FILE:  B:PECAS   .DBF
SECONDARY USE DATABASE
FLD       NAME      TYPE WIDTH   DEC
001     NO:PECA      C    007
002     DESCRICAO    C    020
003     VAL:UNIT     N    007    002
004     EDIT:DATA    C    008
** TOTAL **              00043
```

Uma listagem parcial da tabela de consulta pode ajudar a compreensão.

```
. LIST
00001  01-0119 PWB                       23.18 08/23/83
00002  01-0137 Controlador              114.22 09/03/82
00003  01-0144 Terminal                 221.85 08/26/83
00004  01-0155 ICs                        0.97 10/01/83
```

O envio de dados novos pode processar um registro de cada vez durante os APPENDs ou todas as entradas podem ser gravadas por uma operação de tipo lote. As operações em lotes são mais rápidas, mas o APPEND é mais apropriado quando os dados devem estar nos registros para uso imediato.

Devemos notar que na codificação que segue, temos de construir nossos próprios circuitos para incluir registros ao arquivo na área primária, usando os comandos APPEND BLANK e REPLACE. Seria melhor ativar um formato e apenas usar um APPEND – mas se fizéssemos isto, não poderíamos abandonar o ciclo do APPEND para nossa execução na secundária.

```
****************************************************************
* APPEND Tempo de Envio -Note que varios campos foram deixa-
* dos para fora desta tela porque os dados surgem de outros
* lugares.
****************************************************************
* carregue a tabela de consulta
SELECT SECONDARY
USE PECAS INDEX PARTSP/N
SELECT PRIMARY
* Temos uma imagem de memoria para registros em branco?
IF .NOT. FILE ("blank.mem")
   * abra espaco na memoria para os gets
     STORE " " TO mNO:RT
     STORE "     " TO mDATA:RT
     STORE "     " TO mNO:PECA
     STORE 0 TO mQTD:REC
     STORE 0 TO mQTD:REJ
     STORE "     " TO mINST:DATA
     STORE "     " TO mINST:DISP
     STORE "     " TO mCOMP:DATA
     SAVE TO blank
ENDIF
* ativar um circuito append
store t to we:add
DO WHILE we:add
   * ative a tela e a memoria
     RESTORE FROM blank ADDITIVE
     ERASE
     @ 1,5 SAY " NO:RT " GET mNO:RT
     @ 2,5 SAY " DATA:RT " GET mDATA:RT
     @ 3,5 SAY " NO:PECA " GET mNO:PECA
     @ 4,5 SAY " QTD:REC " GET mQTD:REC
     @ 5,5 SAY " QTD:REJ " GET mQTD:REJ
     @ 6,5 SAY " INST:DATA " GET mINST:DATA
     @ 7,5 SAY " INST:DISP " GET mINSP:DISP
     @ 8,5 SAY " COMP:DATA " GET mCOMP:DATA
     read
* coloque um ponto de escape
   IF mNO:RT = " "
      RELEASE we:add
      RETURN
   ENDIF
   APPEND BLANK
   REPLACE NO:RT WHIT mNO:RT;
     DATA:RT WITH mDATA:RT;
```

```
      NO:PECA WHIT mNO:PECA;
      QTD:REC WHIT mQTD:REC
    REPLACE QTD:REJ WHIT mQTD:REJ;
      INST:DATA WITH mINST:DATA;
      INST:DISP WITH mINST:DISP;
      COMP:DATA WITH mCOMP:DATA
* aqui esta a rotina de consulta
    SELECT SECONDARY
    FIND &mNO:PECA
* coloque uma rotina de nao encontrado aqui
    SELECT PRIMARY
    REPLACE DESCRICAO WITH S.DESCRICAO,VAL:UNIT WITH S.VAL:UNIT
ENDDO note que entramos e saimos do circuito dentro da primaria
```

Esta é uma boa codificação, mas a rotina pode ficar lenta, especialmente se nosso sistema usar disco flexível, e duas ou mais indexações em uso no banco de dados primário. Contando todos os replaces mais as atualizações de índices, cada entrada pode gastar vários segundos depois que pressionarmos return.

Eu prefiro o arquivo de codificação em lote, porque representa um exemplo de "arquivo inteligente", isto é, uma tabela de consulta que chame o usuário para novos dados se não puder encontrar os dados certos em seus registros.

```
*************************************************************
* post.cmd Chamado por UPDATE e por outros reports
* Quando for chamado, MRB:LOG deve estar em
* use na area primary
*************************************************************

* vamos entrete-los enquanto trabalhamos

ERASE
     @ 10,20 SAY " Agora enviando descricao das pecas"
     @ 11,20 SAY " e valores unitarios"

GOTO TOP

**** ESTA E A ETAPA DE VELOCIDADE ****
set index to qrmp/n
SET TALK OFF
SELECT SECONDARY
USE PECAS INDEX PARTSP/N
SELECT PRIMARY

DO WHILE .NOT. EOF
   * nao vamos perder tempo se ja esta tudo enviado
```

```
IF descricao = "    "
  STORE NO:PECA TO mNO:PECA
  SELECT SECONDARY
  FIND &mNO:PECA
  * o que acontece se a peca nao esta na tabela de consulta?
  IF # = 0
      ERASE
      ?
      ?
      ?" POR FAVOR INCLUA A PECA NO ARQUIVO DE CONSULTA"
      ?
      ?"                NO:PECA",mNO:PECA
      ?
      ACCEPT "           DESCRICAO  => " TO mDESCRICAO
      ACCEPT "           VAL:UNIT => " TO mVAL:UNIT
      APPEND BLANK
      REPLACE no:peca WITH Mno:peca, descricao WITH Mdescricao;
         val:unit WITH Mval:unit
  ENDIF
  SELECT PRIMARY
  * vamos conserta-los de uma vez so
     DO WHILE (NO:PECA = mNO:PECA) .AND. (.NOT. EOF)
        REPLACE p.descricao WITH s.descricao;
           p.val:unit WITH s.val:unit
        SKIP
     ENDDO
   ENDIF
     SKIP
ENDDO
SET INDEX TO no:rt,qrmp/n
```

Uma nota sobre a "etapa de velocidade" na codificação: reative o banco de dados para um índice de parte de número. Fazendo isto, permitimos que todos os registros sejam enviados para uma parte particular em uma passagem, em vez de cada registro alcançar seu número de peça na consulta individualmente. Em bancos de dados grandes, isto poderia ser apressado ainda mais tendo uma indexação especial:

```
. INDEX ON (part: no) + $ (descrip, 1, 1) TO specIndx
```

Incluindo o primeiro caractere da descrição para a parte do número no índice, garantiríamos que todos aqueles registros com campos de descrições vazias surgissem no topo de seus números de peças.

EXEMPLO 2 – DADOS DE TAXAS ESTADUAIS DE VENDAS

Em nosso segundo exemplo, a taxa estadual de venda é mantida em um .DBF junto com o nome correto de cada Estado. Mais tarde, um pequeno utilitário pode enviar a um cliente .DBF cada nome de Estado e sua taxa.

Iniciaremos com o banco de dados CLIENTES. Ele possui código e nome do cliente, um Estado, abreviados e por extenso, e a taxa.

```
. USE CLIENTE

. DISP STRUCTURE

STRUCTURE FOR FILE:  B:CLIENTE .DBF
NUMBER OF RECORDS:   00008
DATE OF LAST UPDATE: 01/01/80
PRIMARY USE DATABASE
FLD       NAME      TYPE WIDTH   DEC
001      CODIGO      C    004
002      NOME        C    010
003      EST         C    002
004      ESTADO      C    015
005      TAXA        N    005    003
** TOTAL **              00037
```

Observando os registros em CLIENTES, vemos que taxa e Estado (escritos) são campos não enviados e que alguns dos Estados (abreviados) não estão em maiúscula.

```
. LIST (*Temos agora alguns dados para manipular*)

. LIST
00001  9876 Amigo      GO             0.000
00002  6123 Lina       rj             0.000
00003  1234 Apelo      MG             0.000
00004  1236 Belo       am             0.000
00005  2134 Karina     SE             0.000
00006  1237 Luna       SP             0.000
00007  4536 Franco     sp             0.000
00008  1235 Belo       PR             0.000
```

Vamos indexar CLIENTES a um índice do mesmo nome.

```
. INDEX ON !(EST) TO CLIENTE
00008 RECORDS INDEXED
. LIST
00004  1236 Belo      am          0.000
00001  9876 Amigo     GO          0.000
00003  1234 Apelo     MG          0.000
00008  1235 Belo      PR          0.000
00002  6123 Lina      rj          0.000
00005  2134 Karina    SE          0.000
00006  1237 Luna      SP          0.000
00007  4536 Franco    sp          0.000
```

Agora vamos observar ESTADOS, um .DBV com informações freqüentemente usadas sobre Estados.

```
. USE ESTADOS
. LIST
00001  CE CEARA           0.600
00002  MA MARANHAO        0.065
00003  PE PERNAMBUCO      0.040
00004  AM AMAZONAS        0.080
00005  PA PARA            0.065
00006  RN RG DO NORTE     0.035
00007  MG MINAS GERAIS    0.044
00008  SC SANTA CATARINA  0.100
00009  BA BAHIA           0.025
00010  SE SERGIPE         0.033
00011  PB PARAIBA         0.045
00012  SP SAO PAULO       0.035
00013  ES ESPIRITO SANTO  0.025
00014  PR PARANA          0.045
```

Para tornarmos ESTADO acessível aos FINDs, indexamos no campo-chave. (Os campos-chaves devem ser campos de caracteres para FIND trabalhar com comparações não perfeitas.)

```
. INDEX ON !(EST) TO ESTADOS
00014 RECORDS INDEXED
. LIST
```

```
00004  AM AMAZONAS         0.080
00009  BA BAHIA            0.025
00001  CE CEARA            0.600
00013  ES ESPIRITO SANTO   0.025
00002  MA MARANHAO         0.065
00007  MG MINAS GERAIS     0.044
00005  PA PARA             0.065
00011  PB PARAIBA          0.045
00003  PE PERNAMBUCO       0.040
00014  PR PARANA           0.045
00006  RN RG DO NORTE      0.035
00008  SC SANTA CATARINA   0.100
00010  SE SERGIPE          0.033
00012  SP SAO PAULO        0.035
```

Neste ponto, vamos apresentar FIXEST.CMD, originariamente escrito e depurado em MODIFY COMMAND, e posteriormente refinado no modo não documento (programação) do WordStar. Note que as convenções de maiúsculas e minúsculas, desapareceram da tela quando surgiu FIXEST.

```
****************************************
*  fixest.cmd
*  designado a colocar o nome do estado
*  a partir da abreviacao
*  usando o campo est como uniao
****************************************
set talk off
SELECT SECO
USE ESTADOS INDEX ESTADOS
SELE PRIMAR
GOTO TOP
do while .not. eof
   STORE EST TO MST
   SELE SECO
   FIND '&MST'
   IF # = 0
      ERASE
      ? "           ",MST," nao esta no arquivo de estados"
*     set cons off
      wait
*     set cons on
      sele prim
      skip
      loop
```

```
   ENDIF
   SELE PRIM
   REPL p.ESTADO with s.estado
   repl p.taxa with s.taxa
   skip
enddo
remark          **** TUDO PRONTO *****
set talk on
```

Agora realizaremos uma operação exemplo. Primeiro, com a abreviação misturada dos Estados em clientes.

```
. USE CLIENTE
. LIST
00001  9876 Amigo      GO                 0.000
00002  6123 Lina       rj                 0.000
00003  1234 Apelo      MG                 0.000
00004  1236 Belo       am                 0.000
00005  2134 Karina     SE                 0.000
00006  1237 Luna       SP                 0.000
00007  4536 Franco     sp                 0.000
00008  1235 Belo       PR                 0.000

. DO FIXEST
           GO  nao esta no arquivo de estados
WAITING
           rj  nao esta no arquivo de estados
WAITING
           am  nao esta no arquivo de estados
WAITING
           sp  nao esta no arquivo de estados
WAITING
         **** TUDO PRONTO *****
```

Ao usar FIXEST, devemos notar que Goiás *não* estava no arquivo de dados. FIXEST não é capaz de reconhecer que "am" e "AM" representam o mesmo Estado. Devemos possuir um critério para escrever utilitários com mensagens NO FIND para o usuário.

Agora, observando os resultados, veremos os dados atualizados:

```
. LIST
00001  9876 Amigo     GO                 0.000
00002  6123 Lina      rj                 0.000
00003  1234 Apelo     MG MINAS GERAIS    0.044
00004  1236 Belo      am                 0.000
00005  2134 Karina    SE SERGIPE         0.033
00006  1237 Luna      SP SAO PAULO       0.035
00007  4536 Franco    sp                 0.000
00008  1235 Belo      PR PARANA          0.045
```

Para melhorarmos os resultados, vamos arrumar o campo EST em CLIENTES. Note que este conserto é muito direto – não devemos escrever um programa dBase antes de perguntarmos se podemos usar um comando direto para melhorar os dados em nosso banco de dados. Note que a próxima etapa, que coloca em maiúsculas todos os Estados em CLIENTES, poderia facilmente ser acrescentada ao FIXEST.

```
. REPLACE ALL EST WITH !(EST)
00008 REPLACEMENT(S)

. LIST
00001  9876 Amigo     GO                 0.000
00002  6123 Lina      RJ                 0.000
00003  1234 Apelo     MG MINAS GERAIS    0.044
00004  1236 Belo      AM                 0.000
00005  2134 Karina    SE SERGIPE         0.033
00006  1237 Luna      SP SAO PAULO       0.035
00007  4536 Franco    SP                 0.000
00008  1235 Belo      PR PARANA          0.045
```

Agora quando realizarmos FIXEST, atualizaremos todos os registros possíveis.

```
. DO FIXEST
            GO  nao esta no arquivo de estados
WAITING
            RJ  nao esta no arquivo de estados
WAITING
          **** TUDO PRONTO *****

. LIST
00001  9876 Amigo     GO                 0.000
00002  6123 Lina      RJ                 0.000
```

```
00003  1234 Apelo     MG MINAS GERAIS     0.044
00004  1236 Belo      AM AMAZONAS         0.080
00005  2134 Karina    SE SERGIPE          0.033
00006  1237 Luna      SP SAO PAULO        0.035
00007  4536 Franco    SP SAO PAULO        0.035
00008  1235 Belo      PR PARANA           0.045
```

A única maneira para melhorar isto é digitar Goiás e Rio de Janeiro no ESTADOS.DBF. Este arquivo de banco de dados ainda está sendo utilizado na área SECUNDÁRIA.

```
. SELECT SECONDARY
. APPEND  (*aqui colocamos Goias e Rio de Janeiro*)

. SELE PRIM   (*quatro letras bastam p/comandos*)
. LIST
00001  9876 Amigo     GO                  0.000
00002  6123 Lina      RJ                  0.000
00003  1234 Apelo     MG MINAS GERAIS     0.044
00004  1236 Belo      AM AMAZONAS         0.080
00005  2134 Karina    SE SERGIPE          0.033
00006  1237 Luna      SP SAO PAULO        0.035
00007  4536 Franco    SP SAO PAULO        0.035
00008  1235 Belo      PR PARANA           0.045
```

Agora, quando rodarmos FIXEST, ele não mostrará nenhuma mensagem de erro.

```
. DO FIXEST
          **** TUDO PRONTO *****

. LIST
00001  9876 Amigo     GO GOIAS            0.090
00002  6123 Lina      RJ RIO DE JANEIRO   0.070
00003  1234 Apelo     MG MINAS GERAIS     0.044
00004  1236 Belo      AM AMAZONAS         0.080
00005  2134 Karina    SE SERGIPE          0.033
00006  1237 Luna      SP SAO PAULO        0.035
00007  4536 Franco    SP SAO PAULO        0.035
00008  1235 Belo      PR PARANA           0.045
```

EXEMPLO 3 – UM DONO, MUITOS BICHOS

Quando um registro na área primária está associado a muitos pontos na secundária isto representa uma situação tão comum, que pode ser considerada como um programa-padrão em dBase. Nosso exemplo será um módulo que nos permite inserir os bichos para cada novo registro incluído em SOCIOS, quando um novo sócio for incluído. Um utilitário separado poderia ser escrito ao longo das mesmas linhas para incluir um bicho ou bichos aos sócios já existentes.

Sei que este é um exemplo extravagante, mas imagine, quando ler o programa, que os sócios são clientes e os bichos são os pedidos de produtos. O programa seria o mesmo, embora a formatação e as estruturas dos registros não o fossem.

Pseudocódigo:
coloque SOCIOS na primária e os bichos na secundária
pegue um novo sócio e o número de bichos que ele/ela possui
se o número de bichos for > 0
    vá para a secundária e pegue um bicho
    faça isto até que tenha todos os bichos
se o usuário quiser, pegue um novo sócio e repita
faça isto até processar todos os sócios

Você pode fazer o primeiro refinamento deste pseudocódigo sozinho? Uma referência: O programa não necessitará de nenhuma declaração IF ... ENDIF.

```
********************************************
* NEWSOCIO.FMT - Para usar com SOCIOS.DBF
* quando acrescentar novos bichos
********************************************
@ 3,5 SAY "NOME " GET nome
@ 3,$+5 SAY "SOBRENOME " GET sobrenome
@ 5,5 SAY "ENDERECO " GET endereco
@ 7,5 SAY "CIDADE " GET cidade
@ 7,$+7 SAY "ESTADO " GET estado
@ 7,$+7 SAY "CEP " GET cep
* deixaremos contribuicao e pagamento de fora
*
* aqui temos que pegar o numero de bichos - nao mais do que
* dois digitos, por favor
@ 12,8 SAY "INFORME O NUMERO DE BICHOS " GET bic PICTURE "99"
*
* ajuda ao usuario - teclas de controle, etc.
*
* fim do arquivo NEWSOCIO.FMT
```

```
**************************************************
* NEWBICHO.FMT  - Para usar com BICHOS.dbf
* durante as inclusoes relacionadas com SOCIOS.DBF
**************************************************
* dizemos sobrenome porque o programa preenche o campo
@ 3,5 SAY "SOBRENOME "
@ 3$,+2 SAY s.SOBRENOME
@ 5,5 SAY "TIPO DO BICHO " GET tipo
@ 7,5 SAY "NOME DO BICHO " GET nome
*
* ajuda ao usuario - teclas de controle, etc.
*
* fim do arquivo NEWBICHO.FMT


************************************************
* ADDDONOS.CMD - Desenhado para mostrar uma
* relacao de um-para-muitos
*
* SOCIOS na primaria, BICHOS na secundaria
* Este programa incluira um dono de cada vez
* mas o dono pode ter ate 64k de bichos
************************************************
* ative a configuracao
SET TALK OFF
USE SOCIOS
* ponha o arquivo BICHOS na area secundaria
SELECT SECONDARY
USE BICHOS INDEX DONOS
SELECT PRIMARY
* abra um circuito para pegar os socios
* faca um circuito infinito. Colocaremos a saida mais tarde
DO WHILE T
   * crie uma variavel para  bic  para o numero de bichos
   * bic deve ser numerica
   STORE 0 TO BIC
   * pegue o novo socio
   SET FORMAT TO NEWSOCIO
   * o formato esta criado
   APPEND BLANK
   READ
   *
   * ate agora foi facil - agora pegaremos
   * um ou mais bichos
   *
```

```
    * vamos aonde estao os bichos
    SELECT SECONDARY
    SET FORMAT TO NEWBICHO
    * agora um circuito para mais de um bicho
    * teste a entrada de bichos
    * bic e numerico e 0 e falso por default
    DO WHILE BIC
       APPEND BLANK
       REPLACE S.SOBRENOME WITH !(P.SOBRENOME)
       * pegue o resto do registro
       READ
       * volte 1 nos bichos
       STORE BIC-1 TO BIC
    ENDDO bic
    * de volta aos donos, rapido
    SELECT PRIMARY
    * mais donos? (Esta e a nossa porta de saida)
    ERASE
    @ 11,11 SAY "MAIS NOVOS SOCIOS ? (S/N)"
    ?
    WAIT TO denovo
    IF !(denovo) <> "S"
       *  restaure a configuracao e volte
       CLEAR
       SET TALK ON
       RETURN
    ENDIF
ENDDO
*
* fim do arquivo ADDDONOS.CMD
```

Usando uma cópia de cinco registros de SÓCIOS, vamos incluir um sexto sócio que possui três animais de estimação: uma galinha chamada Poka, uma foca chamada Ester e um peru chamado Léo. Antes da inclusão possuíamos:

```
. LIST
00001 Mila    Moreira   R.Elma, 111          Suite 1   Sao Paulo SP 92109  5.00 .T.
00002 Monica  Medeiros  R. Diva, 2219                  Curitiba  PR 44110  5.99 .T.
00003 Pedro   Penedo    R.Sao Luiz, 22       Apt. 232  Sao Paulo SP 92109  0.00 .F.
00004 Carlos  Covas     R. "A", 88                     Sao Paulo SP 92101  0.25 .T.
00005 Davi    Dunas     Rua das Flores, 123  Sala 17   Aracaju   SE 36222  1.00 .T.

. USE BICHOS
```

```
. LIST
00001  PENEDO    cachorro Fido
00002  ZORZI     gato     Arthur
00003  SIMOES    pombo    Apollo
00004  MEDEIROS  cachorro Bola
```

Agora digitamos:

```
. DO ADDDONOS
```

Se voltarmos a ADDDONOS.CMD, veremos que o programa coloca os arquivos apropriados em uso – não precisamos nos preocupar com isto.

Depois da inclusão:

```
. LIST
00001 Mila    Moreira   R.Elma, 111           Suite 1   Sao Paulo SP 92109  5.00 .T.
00002 Monica  Medeiros  R. Diva, 2219                   Curitiba  PR 44110  5.99 .T.
00003 Pedro   Penedo    R.Sao Luiz, 22        Apt. 232  Sao Paulo SP 92109  0.00 .F.
00004 Carlos  Covas     R. "A", 88                      Sao Paulo SP 92101  0.25 .T.
00005 Davi    Dunas     Rua das Flores, 123   Sala 17   Aracaju   SE 36222  1.00 .T.
00006 Julio   Belo      Praia da Lua                    Recife    PE 92022  0.00 .F.

. SELECT SECONDARY

. LIST  (* Os registros estão indexados porque ADDDONOS deixou-os assim *)

00005  BELO      foca     Ester
00006  BELO      galinha  Poka
00007  BELO      peru     Leo
00004  MEDEIROS  cachorro Bola
00001  PENEDO    cachorro Fido
00003  SIMOES    pombo    Apollo
00002  ZORZI     gato     Arthur
```

REGISTROS COM FINAL DUPLO

Embora os exemplos neste capítulo mostrem os arquivos nas áreas primárias e secundárias ligados em um simples banco de dados de dois arquivos, nas aplicações reais os bancos de

dados tendem a expandir lateralmente para fora. Como isto acontece freqüentemente, o arquivo A une-se ao B e este como resposta une-se ao C:

A ←→ B ←→ C

Quando isto ocorre, significa que dois campos separados em B são campos unidos, um para A e um para C. Explicando de outra maneira, se os campos unidos de B para A e C eram os mesmos, assim os dois arquivos, B e C provavelmente estariam ligados.

Em tal desenho, B é chamado **registro com final duplo** e poderíamos manter dois índices em seus dois campos unidos. Por exemplo, um armazenamento poderia manter os registros de clientes no arquivo A, os registros de vendas no B e os de estoque em inventário no C. O A seria unido ao B (cliente para venda) pelo número do cliente, e B seria unido ao C (vendas deduzidas do inventário) através do estoque ou número de item.

SAÍDA DE ARQUIVOS DE DADOS UNIDOS

Uma tarefa constante é desenhar relatórios que possuem os conteúdos de dois arquivos relacionados em uma estrutura de um-para-muitos. As faturas, por exemplo, consistem em um endereço de um cliente seguido de uma lista com peças ou itens. Há um registro na primária para o cliente, muitos registros na secundária para as peças despachadas.

Até agora vimos que se podemos colocar um relatório na tela, podemos enviá-lo à impressora. Vamos desenhar um relatório que emitirá uma lista de todos os animais de estimação em bichos para um determinado sócio do arquivo SOCIOS.

Se fizermos este arquivo de comandos de relatório genérico o suficiente, poderemos desenhar vários controles de estruturas que o chamarão

Pseudocódigo:

coloque os donos na primária, bichos na secundária, os dois indexados adequadamente
armazene o sobrenome do dono à memória e coloque em maiúsculas para formar a união
vá à secundária e encontre os bichos
mostre todos os animais para o dono
volte à primária

Poderíamos ser tentados a colocar em alguma rotina de manipulação situações onde não houvessem bichos ou talvez fosse melhor deixar a cargo da rotina chamadora. Na verdade, esta é uma questão de preferência individual; possuindo duas não causaria danos (isto é, se o relatório não for chamado para uma situação de "não encontrado", esta codificação não importaria). O único refinamento principal é o da etapa de visualização:

Primeiro Refinamento:

visualizar significa:

imprimir um cabeçalho nomeando o dono
imprimir alguns cabeçalhos de coluna
colocar os bichos para o dono

```
**********************************************
*  REPORT.CMD  - Relatorio  de tela para
*  listar os bichos por donos
**********************************************
*
* Note que este modulo nao pode confirmar a
* existencia de um bicho ou encontrar o bicho
*
* OS DONOS DEVEM ESTAR NA PRIMARIA E OS BICHOS
* NA SECUNDARIA COM O INDICADOR NO PRIMEIRO
* BICHO
**********************************************
*
*  cabecalho
?
?
? [                  RELACAO DOS BICHOS DE ],TRIM(NOME),SOBRENOME
?  [   ------------------------------------------------------------]
? [        Num               Nome do Bicho                Tipo     ]
? [     -------           ----------------             ----------- ]
?
* pegue os bichos
SELECT SECONDARY
* mostre todos os bichos com numeros
STORE 1 TO BIC:CONT
* lembre-se o sobrenome e maiusculo na secundaria
DO WHILE !(P.SOBRENOME) = S.SOBRENOME
   * a linha principal do relatorio
   ? [  ],STR(BIC:CONT,3,0),[    ],NOME,;
       [    ],TIPO
   * uma linha de espaco
   ?
   * proximo bicho
   SKIP
   * incrementa numero da linha
   STORE BIC:CONT + 1 TO BIC:CONT
```

```
ENDDO
RELEASE BIC:CONT
SELECT PRIMARY
*
* fim do arquivo report.cmd
```

Se lermos novamente REPORT FORM na seção seis (Capítulo 4) poderemos reconhecer que o programa de clientes acima poderia ser substituído por uma das formas de relatório dBase. Vamos assumir que usamos REPORT FORM para definir um arquivo ".FRM" chamado "BICHOLST.FRM", que fornece uma lista de animais de estimação. Assim, todo o módulo do relatório poderia ficar estabelecido como:

```
******************************************
* REPORT.CMD ver 2
******************************************
*
* esclarecimentos sobre a rotina e suas
* condicoes de entrada
*
STORE TRIM(NOME),SOBRENOME TO DONO:NOME
SET HEAD TO DONO: &DONO:NOME
SELECT SECONDARY
REPORT FORM BICHOLST FOR !(P.SOBRENOME)=S.SOBRENOME
SELECT PRIMARY
*
* fim da versao 2
```

Relatórios de clientes possuem espaço. Agora apresentaremos uma rotina de chamamento.

Pseudocódigo:

```
pegue o nome de um sócio que desejamos
veja se o sócio existe
    se não, volte
veja se o sócio possui bichos
    se não, volte
chame REPORT.CMD
```

Não são necessários os refinamentos. Há apenas dois lugares onde podemos prever o surgimento de problemas. Primeiro, deveríamos ter certeza do que aconteceria se o usuário digitasse o nome dos sócios com letras maiúsculas ou com minúsculas.

Como isto acontece, o programa escrito não possui nenhuma maneira de correção para os erros de maiúsculas e minúsculas. Nosso índice original no campo do sobrenome de SOCIOS, SOBRENDX, não se encaixa na regra "!(sobrenome)". Poderíamos corrigir formando um novo índice para uma provável aplicação real. Eu não fiz isto aqui.

A segunda área na qual assistiremos nossos passos está nas duas etapas de volta. É apenas um detalhe, mas devemos lembrar-nos de voltar a acionar talk on, selecionar a primária novamente, liberar as variáveis de memória apropriadas, e assim por diante, antes de voltarmos – detalhes.

```
**********************************************
* BICHOS.CMD - Relatorio de Tela
**********************************************
* Programa de relatorio para usar com SOCIOS
* e BICHOS relacionados
*
* Fara relatorio de todos os bichos por donos
* confirmando a existencia dos bichos e chamara
* o programa REPORT.CMD
*
* REPORT espera encontrar SOCIOS indexado pelo
* sobrenome na PRIMARIA e BICHOS indexado por
* donos na SECUNDARIA
*
*********************************************
*
SET TALK OFF
* pegue o dono
ERASE
* um truque para fazer o ACCEPT aparecer na linha 11
@ 10,0 SAY " "
ACCEPT [     ENTRE O SOBRENOME DO DONO ] TO MSOBRE
* pegue o registro do socio
FIND &MSOBRE
* o que acontece se nao existe o socio?
IF # = 0
   * beep - o socio nao esta aqui
   ? CHR(7)
   ERASE
   @ 11,8 SAY MSOBRE
   @ 11,$+2 SAY "SOCIO INEXISTENTE - CONFIRA"
   @ $+2,10 SAY "        E TENTE OUTRA VEZ"
   SET TALKON
   RETURN
ENDIF
* ele esta aqui, mas ele tem bichos?
```

```
SELECT SECONDARY
* precisamos procurar nas maiusculas
STORE !(MSOBRE) TO FINDER
FIND &FINDER
IF # = 0
   * nenhum bicho
   ERASE
   @ 11,8 SAY MSOBRE
   @ 11,$+2 SAY "NAO TEM NENHUM BICHO NO ARQUIVO"
   * retorne a primaria e pronto
   SELECT PRIMARY
   SET TALK ON
   RETURN
ENDIF
* se estamos aqui e porque temos um socio com bichos
* mas primeiro ...
SELECT PRIMARY
*
*  AGORA!!
ERASE
DO REPORT
SET TALK ON
*
* fim do arquivo bichos.cmd
```

Não resta nada a não ser tentar. Primeiro temos de ativar a configuração – SOCIOS na primária e BICHOS na secundária, com os índices corretos e enviados propriamente.

```
. USE SOCIOS INDEX SOBRE

. SELECT SECONDARY

. USE BICHOS INDEX DONOS

. SELECT PRIMARY
```

Agora que estamos prontos, vamos testar nosso programa tentando um nome que não está no arquivo SOCIOS e então um sócio que não possui bichos:

```
. DO BICHOS

      ENTRE O SOBRENOME DO DONO : Fidel
```

```
          Fidel  SOCIO INEXISTENTE - CONFIRA
                           E TENTE OUTRA VEZ

. DO BICHOS


        ENTRE O SOBRENOME DO DONO : Covas
        Covas NAO TEM NENHUM BICHO NO ARQUIVO
```

Já possuímos o suficiente de falhas. Testamos as duas rotinas de volta e descobrimos que elas fazem o que imaginávamos – agora para o relatório de um sócio que possui registros de bichos no arquivo de dados na secundária:

```
. DO BICHOS

        ENTRE O SOBRENOME DO DONO : Belo

                         RELACAO DOS BICHOS DE Julio Belo
           ---------------------------------------------------

           Num              Nome do Bicho               Tipo
          -------          ----------------          -----------

           1                   Ester                   foca

           2                   Poka                    galinha

           3                   Leo                     peru
```

OS REGISTROS DE UM-POR-UM – A "OPÇÃO DE UNIÃO"

O dBase possui limitações. Há um limite máximo em cada número significativo – tamanho de um campo, número de registros, o maior número – e a seriedade destas limitações depende do que estamos tentando fazer.

Uma limitação que é completamente fácil de alcançar é a de 32 campos por registro. Os programadores que estão iniciando em dBase, especialmente, tentam definir as estruturas de registros contando todo o banco de dados em somente um arquivo (não é um objetivo ruim, se puder ser realizado, mas sempre impraticável). No próximo capítulo, discutiremos o desenho de bancos de dados e veremos as maneiras de fazer a manutenção, mas mesmo com essas técnicas, surgem as aplicações que precisam de um registro com mais de 32 campos.

Uma solução é desenhar uma estrutura de registro que se estenda por dois arquivos montados nas áreas primária e secundária. Quando desenhamos esta estrutura, depende de nós escrever os programas necessários para a entrada de dados, como fizemos para SÓCIOS e BICHOS. Na verdade, a codificação é mais simples no desenho de um-por-um, porque um circuito externo pode controlar uma operação de acréscimo:

Pseudocódigo:

```
coloque os primeiros 32 campos na primária e o resto
    na secundária
faça a indexação das duas áreas no campo unido para que eles
    fiquem na mesma ordem aparente
inicie o circuito
        inclua a metade da primária
        selecione a secundária
        inclua a metade da secundária
        selecione a primária
inicie um novo registro
pare quando desejar
```

Se quiséssemos colocar em uma rotina que nos desse a certeza da existência de apenas um registro na primária e secundária para cada união, e se esta fosse a única maneira que os registros sempre são incluídos a estes dois bancos de dados de arquivo, então a chance de erro seria pequena.

E a saída? Estes arquivos de relatório sempre requerem arquivos de clientes? Felizmente, não. Há um SET chamado LINKAGE que nos permite usar REPORT FORM (até a sua limitação de 24 campos por relatório) com arquivos de dados nas áreas primária e secundária. Primeiro vamos descrever qual é o efeito quando o comando SET LINK ON é fornecido. Quando LINK está ativado, qualquer comando que pode aceitar um escopo (como "NEXT 5") avançará os arquivos nas áreas primárias e secundárias simultaneamente através de sua ordem aparente.

Isto não se aplica aos comandos SKIP, FIND, LOCATE e GOTO. As áreas primária e secundária permanecerão independentes quando estes comandos de movimentação forem fornecidos.

Os comandos que unem os dois arquivos de dados incluem LIST, DISPLAY, DELETE e REPORT FORM. (Já que o apagamento é uma operação escrita, DELETE só apaga o registro na área selecionada. Se os dois registros unidos estão para ser apagados, o programador deve escrever uma rotina que apaga os dois.) Verifique se realmente entendeu esta aplicação – tudo o que o LINKAGE fará é avançar automaticamente os registros dos arquivos nas duas áreas. Devemos ter certeza de que eles estão na ordem certa, que os dois arquivos iniciam com os registros adequados e, o mais importante, devemos lembrar-nos de desativar o comando SET LINK-OFF quando terminarmos a operação.

Para demonstrarmos a conveniência deste comando, vamos usar dois pequenos arquivos subativados chamados LEFT.DBF e RIGHT.DBF. O LEFT manipulará os dois nomes de campos de SÓCIOS e RIGHT manipulará cidade e estado.

```
. USE LEFT
. DISPLAY STRUCTURE

STRUCTURE FOR FILE:  B:LEFT    .DBF
NUMBER OF RECORDS:   00006
DATE OF LAST UPDATE: 01/01/80
PRIMARY USE DATABASE
FLD       NAME       TYPE WIDTH   DEC
001     NOME          C    006
002     SOBRENOME     C    010
** TOTAL **              00017

. LIST
00001  Mila   Moreira
00002  Monica Medeiros
00003  Pedro  Penedo
00004  Carlos Covas
00005  Davi   Dunas
00006  Julio  Belo

. SELECT SECONDARY
. USE RIGHT
. DISPLAY STRUCTURE
STRUCTURE FOR FILE:  B:RIGHT   .DBF
NUMBER OF RECORDS:   00006
DATE OF LAST UPDATE: 01/01/80
SECONDARY USE DATABASE
FLD       NAME       TYPE WIDTH   DEC
001     CIDADE        C    013
002     ESTADO        C    002
** TOTAL **              00016

. LIST
00001  Sao Paulo     SP
00002  Curitiba      PR
00003  Sao Paulo     SP
00004  Sao Paulo     SP
00005  Aracaju       SE
00006  Recife        PE
```

Já que REPORT FORM realiza uma operação de leitura, ele pode acessar os dois arquivos de uma só vez. Não há problema em definir rapidamente "TST.FRM" que possui nomes de campos desenhados a partir dos dois arquivos:

```
                (* Linha em branco para "W = 128", etc. *)
Y
RELATORIO COMBINADO
N
N
10,NOME
>NOME
15,SOBRENOME
                    (* Linha em branco p/nenhum cabeçalho *)
16,CIDADE
<CIDADE
15,ESTADO
<ESTADO
```

Agora, da área primária, vamos operar o relatório TST sem ativar a união. Como veremos, os resultados são perfeitamente lógicos, mas não tão úteis:

```
. SELECT PRIMARY
. REPORT FORM TST     (* aqui rodando sem LINK on *)


       PAGE NO. 00001


                              RELATORIO COMBINADO

            NOME                CIDADE          ESTADO

       Mila      Moreira        Sao Paulo        SP
       Monica    Medeiros       Sao Paulo        SP
       Pedro     Penedo         Sao Paulo        SP
       Carlos    Covas          Sao Paulo        SP
       Davi      Dunas          Sao Paulo        SP
       Julio     Belo           Sao Paulo        SP


. SELECT SECONDARY
. DISPLAY
00001  Sao Paulo     SP
```

O arquivo da secundária nunca sai do primeiro registro. Vamos tentar novamente com a união ativada (depois de voltar para a primária e alcançar o topo, é claro):

```
. SET LINKAGE ON

. SELE PRIM
. REPORT FORM TST


      PAGE NO. 00001


                              RELATORIO COMBINADO

             NOME                 CIDADE           ESTADO

      Mila       Moreira          Sao Paulo        SP
      Monica     Medeiros         Curitiba         PR
      Pedro      Penedo           Sao Paulo        SP
      Carlos     Covas            Sao Paulo        SP
      Davi       Dunas            Aracaju          SE
      Julio      Belo             Recife           PE
```

Devo prevenir sobre os erros que aguardam o programador quando deixar o LINK ativado e tentar outras tarefas. Eles serão muitos e desastrosos. Consideremos, por exemplo, a tentativa de SKIP em um arquivo que possui mais registros enquanto o EOF foi alcançado em um arquivo unido. Isto não funciona. Sempre desative LINK quando terminar uma operação.

ESCOLHENDO UNIÕES

Um **campo de união** é um campo que existe em dois ou mais arquivos de dados. O conteúdo deste campo é a união. A natureza da união entre os dois arquivos é determinada pelos programas que escrevemos. Tudo isto deveria estar claro.

Mas o que faz uma boa união? Deveríamos escolher ou o programa deveria designar uma união? Podemos usar o número de registro do dBase? As respostas para algumas dessas perguntas relacionam-se ao desenho do programa e variam de aplicação para aplicação.

Primeiro vamos observar o que uma união requer. Seja o que um desenho de um banco de dados chamar para a relação dos registros de um-para-um ou um-para-muitos entre os arquivos na primária e secundária, ainda há alguns requisitos específicos.

Para que um programa funcione, a união deve ser absoluta e não ambígua. Em um sistema de controle manual, tal como uma escola (embora muitas delas usem computadores todos os dias), podem existir dois alunos com o nome "Francisco Silva". A secretária pode chamar os professores e descobrir que o aluno "A" está na 8ª série e o "B" está na 2ª.

O dBase não é tão inteligente. Se enviarmos a um programa dBase uma operação em um arquivo de dados no qual dois registros possuam conteúdos idênticos no campo de união, o dBase encontrará o primeiro e fará a operação. As uniões duplicadas não estão sempre incorretas. Mas as uniões duplicadas *acidentalmente* são perigosas, por isso temos de escolher as uniões cuidadosamente para prevenir essas situações.

Apresentaremos algumas características que tornarão uma união como ideal:

1. As uniões corretas são únicas.

2. Os campos de caracteres pequenos (quatro até dez caracteres) são os melhores, embora os campos numéricos são aceitáveis quando o programa designa uniões únicas. Se as uniões possuírem caracteres alfa, é melhor colocá-los todos em maiúsculas.

3. Se os usuários responderem a perguntas para informação dos clientes sobre os registros, pode ser útil escolher uma união que o cliente conheça. Isto pode facilitar as perguntas pelo telefone, quando podemos perguntar ao cliente por seu nº de CGC, por exemplo, em vez do número que o nosso departamento de contas designou a ele.

4. As uniões corretas são "naturais". Elas têm sentido e aproximam-se da lógica manual.

5. As uniões corretas não mudam. Se partes dos números suportam revisões constantes elas não podem ser as melhores uniões em nosso banco de dados de inventário. Se apenas os dois últimos dígitos de um número de peça com dez dígitos mudarem após uma revisão, então usaremos $(parte:num, 1,8) como união e geraremos um campo de união em cada registro para ele.

A maioria das uniões corretas espelha suas utilizações. Nossos registros estão recheados de números de empregados, números de peças, números de contas e centenas de outras uniões que já instalamos em todos os nossos bancos de dados. Estes números foram originariamente designados para unir registros e ainda representam as melhores uniões para usarmos. Use-os.

UNIÕES CALCULADAS

Haverá situações em que os dados que possuímos nos bancos de dados não contêm ou não sugerem uma boa união natural. Nestas situações, devemos designar uma união aos registros do banco de dados ou fazer com que o dBase a designe automaticamente. Uniões calculadas deveriam possuir tantas características de uniões naturais quanto possível, como esta. Queremos calcular uma união para um registro determinado que não pode ser (ou está muito diferente para ser) calculada por qualquer outro registro.

Vamos ilustrar com um exemplo. A seguir encontraremos uma parte de um registro de dados retirado de uma telefonista de uma firma de reembolso postal de um novo pedido de cliente:

Data do Pedido	Nome	Telefone	Cep	União
03/23/83	SIMOES	619/483-6222	92109	LINK

Este cliente pediu quatro itens:

LINK	Ursinho de pelúcia	$49,95
LINK	Botas de couro de cobra	$99,95
LINK	Champagne de sidra	$29,95
LINK	Sabonete perfumado	$ 4,95

Quando a telefonista anotar o pedido, o programa dBase automaticamente colocará um novo registro do cliente no arquivo de pedidos na área primária, colocará quatro novos registros no arquivo de pedidos na secundária e também preencherá os campos LINK com a mesma string nos cinco registros.

A questão, então, é qual string escolher. O sobrenome parece bom, mas não podemos garantir que este é o único "Simões". O CEP não parece prometer muito – os comerciantes tendem a concentrar-se na mesma área. O número do telefone possui uma grande chance de ser único mas pode mudar. Até pior, Simões pode não lembrar do seu telefone antigo depois de dois anos. Apesar disso, dos campos simples, ele é provavelmente o melhor.

As combinações de campos possuem vantagens sobre os campos simples. Consideremos esta expressão como uma união:

```
. REPLACE união WITH !(sobrenome) + $(data, 1, 3) +$(data, 7, 2) +$(tel, 8, 5)
```

em nosso registro exemplificado seria:

```
"SIMOES       03/83-6222"
```

Esta é uma excelente união calculada. Representa uma boa chance de ser única – quantos Simões tivemos em março de 83 do qual o número de telefone terminava em "6222"? E o dado concatenado é do tipo que um cliente poderia lembrar.

Podemos notar que não me importo em usar a função TRIM( ) no campo sobrenome na expressão para a união calculada. Na verdade, isto não faz diferença (a não ser esteticamente) para os resultados de classificação e indexação. Porém, se tivéssemos de considerar os espaços, teríamos usado arbitrariamente os primeiros cinco ou seis caracteres do campo sobrenome.

Um campo calculado também pode ser um campo numerado, embora isto apresente uma outra complicação, o conceito de um "arquivo de memória", que será discutido completamente no próximo capítulo. A idéia é simples designaremos um número para cada registro conforme o surgimento, seja um número do cliente, fatura ou outro qualquer. As uniões de campo-número possuem a vantagem de que todos os FINDs devem ser exatos, economizando algumas linhas de codificação e eliminando a necessidade de usar SET EXACT ON e SET EXACT OFF.

Não há nada de errado com este método desde que não designamos o mesmo número para dois registros. Na maioria dos casos, isto implica escrever codificações em nosso programa que (1) checa para ver se um número já está em uso, e/ou (2) mantém o maior número usado como uma variável de sistema definida pelo usuário. Vamos examinar isso. Imaginem que o cliente, antes do mencionado Simões, tivesse sido o número 18.343. Isto seria armazenado na memória como uma variável numérica chamada "X: HI: NUM", gravado no disco como "SYSVAR MEM" e trazido durante o início do programa pelo comando:

```
. RESTORE FROM sysvar ADDITIVE
```

Assumindo que nunca usamos um número de clientes duas vezes, podemos incluir o registro Simões com as linhas de codificação:

```
APPEND BLANK
STORE x: hi: num + 1 TO x: hi: num
REPLACE link WITH x: hi: num
EDIT     (* Para preencher o resto dos campos primários *)

(*o restante para colocar os pedidos na secundária etc.*)
```

No momento de desativarmos nosso programa, teríamos de gravar este número X: HI: NUM no arquivo SYSVAR; MEM, junto com todas as outras variáveis de sistema que poderíamos ter definido (forneceríamos a eles a característica de prefixo "X:"):

```
SAVE TO sysvar ALL LIKE "X: *"
```

Vimos aqui uma maneira inteligente sobre este comando; usamos o número do registro e do dBase de cada registro primário como uma união. Então poderemos usar GOTO em vez de FIND, que eliminará um arquivo de índice, e a codificação para armazenar a união na secundária será simples:

```
STORE # TO Slink
SELECT SECONDARY
REPLACE link WITH Mlink
```

Isso poderia funcionar, depois de forçar o nosso arquivo na área primária a permanecer estático, mas não é recomendável. Lembre-se de que os números dos registros representam a maneira de o dBase manter a tabulação nos registros e isto não é permanente. Eles são mudados pelo

SORT, DELETE e PACK e pelo COPY TO ou APPEND FROM quando os índices estão em uso. Além disso, a natureza exata dessas mudanças é imprevisível.

Mesmo gravar o número do registro em um campo no registro não é seguro. Vamos imaginar que Simões foi incluído com o número do registro 00404. Substituímos o campo de união com 404 e prosseguimos com nossa operação.

Posteriormente após um PACK, o número do registro de Simões é reduzido para 398, mas a união permanece 404. Seis registros mais tarde, outro 00404 será incluído. É melhor abandonar os indicadores de números dos registros quando ativamos uniões.

A LEITURA DE UM TERCEIRO ARQUIVO DE DADOS

Embora o dBase limite o usuário ou programador a dois arquivos montados em uma área primária e secundária, um terceiro arquivo pode ser aberto e lido, se necessário. O comando envolvido é o APPEND FROM, e ele aceita condições.

Consideremos o seguinte banco de dados: os clientes estão na primária com NOME, UNIÃO e um booleano chamado HAS:PEDIDOS; os pedidos para o mês estão na área secundária pela União.

Um NO-FIND no arquivo de pedidos significa que um pedido está no arquivo do mês anterior.

Cliente	*União*	*tem:pedido*	*união*	*Pedido*	*preenchido*
BAR DO JO	1	.T.	3	Biscoitos	.T.
			2	Cervejas	.F.
			6	Amêndoas	.F.

```
* agora encontramos o pedido para o Bar do Jo
* coloque "1" na varmem
STORE união TO Munião
SELECT SECONDARY
* onde está o pedido do Jo?
FIND &Munião
* talvez não esteja nos pedidos deste mês ainda
IF # = 0
   * ainda estamos no arquivo de pedidos
   APPEND FROM mês passado FOR (união = &Sunião) .AND. .NOT. preenchido
ENDIF no-find if
```

Com certeza, mesmo se gostarmos desta codificação, devemos perguntar que tipo de programa de contabilidade nos permitiria entrar em um novo mês sem automaticamente reposicionar todos os pedidos do mês anterior que não foram preenchidos.

CAPÍTULO 9

Considerações Sobre o Desenho de Banco de Dados

O desenho dos bancos de dados representa um assunto complexo e fascinante e um capítulo não é o suficiente para discuti-lo. Nenhum livro sobre uma linguagem poderia colocar todas as ramificações de como estruturar um banco de dados. Felizmente, não precisamos fazer isto. A maioria dos procedimentos complexos desenvolvidos na ciência de gerenciamento de banco de dados surgiu dos computadores de grande porte e encontram-se fora do mundo do dBase.

O dBase é designado a usuários e tarefas únicas (embora tenha sido adaptado por alguns programadores para sistemas operacionais de multi-usuários tal como o MP/M Digital Research). De fato, são as limitações do dBase que mais afetam o desenho de um sistema. O fato de o dBase limitar-se a dois arquivos na memória ao mesmo tempo e consumir tempo de operação ao atualizar arquivos e índices, indica que os sistemas tendem a ser desenhados com cuidado.

O que discutiremos, entretanto, são as considerações sobre o desenho que podem mostrar a diferença entre um bom sistema e um quase impraticável de operação lenta.

OTIMIZAÇÃO NA ESTRUTURA DO REGISTRO

Nenhuma característica de desenho simples representa diferença na qualidade de um programa do que possuir um conjunto racional e bem desenhado de estruturas de registro em um banco de dados.

Esta é a única etapa que nunca deveria ser feita sem reflexão – sentar e digitar CREATE sem planejar, representa um passo para possíveis problemas, mesmo em um programa simples. Nossos objetivos são desenhar registros nos vários arquivos no banco de dados para que:

1. A programação seja simples e direta.
2. A velocidade do programa seja aumentada, eliminando ao mínimo os movimentos do banco de dados às entradas e saídas na memória.
3. A utilização do espaço do disco seja eficiente, assim os registros não terão espaço vazio e não conterão nada além do necessário.

Para alcançar uma estrutura de registro ideal, vários instrumentos de planejamento foram desenvolvidos durante anos. Entre eles, os mais aplicados em dBase são: "o dicionário de dados", "os diagramas circulares" e o conceito de "arquivo nivelado".

Os Dicionários de Dados

Os grandes bancos de dados possuem **dicionários de dados** no computador no qual cada item de dado no banco de dados é digitado com significado preciso: quem o utiliza, quem é responsável pela entrada, e quem pode ou não vê-lo. Estes dicionários, entre outras coisas, previnem a entrada do mesmo dado com dois nomes diferentes (sinônimos) ou do mesmo nome usado para dois dados diferentes (homônimos).

Isto pode parecer um desastre para um programa simples em dBase, mas o fato de criar um dicionário de dados implica um tempo bem empregado, especialmente para consultores que desenvolvem programações de clientes. Ocasionalmente, todos os consultores passam pela experiência de destruir um programa que decepciona o cliente. Na maioria dos casos, o problema poderia ser prevenido se o consultor e os usuários finais tivessem elaborado um dicionário de dados compreensível no qual o termo levasse ao dado, quem o utilizou e como.

No caso do dBase, a etapa definitiva na determinação de uma ótima estrutura de registro implica a determinação dos nomes de campo, tipos e larguras e da formalização das associações de campos. Se fôssemos listar o histórico dos dados obteríamos: qual é o documento-fonte, quem o inseriu e quando, quem o editou e quando; e a um nível inferior, quais os relatórios em que ele aparece (porque sempre assumiremos que se podemos obter o dado correto *dentro* do banco de dados, como resultado obteremos os relatórios corretos).

Quando desenvolvemos o dicionário de dados, ele deve circular entre todos os usuários finais e supervisores que o utilizarão no banco de dados, para que cada um possa verificar as idéias e comentários dos outros, e até discordar ou discutir. É mais do que uma política. Quase sempre as pessoas não têm idéia de quais dados seus colegas estão se baseando no trabalho a ser desenvolvido.

Os cobradores em uma doca de embarque podem estar gravando as datas das entregas achando que é necessário para o estoque. Em contrapartida, o estoque pode estar lançado pelo número da fatura. E o planejamento de materiais pode controlar as novas entregas pelos números originais de requisição.

Deve haver boas razões históricas para tudo isso, mas o programador não pode escrever um programa que encontrará a aprovação dos três departamentos sem obter a concordância sobre quais itens de dados seriam os campos-chaves, quais seriam os nomes etc.

Um dicionário de dados pode ser criado através do dBase. Uma possível estrutura seria:

NOME DE CAMPO	
DEFINIÇÃO	(o que representa exatamente este dado)
TIPO DE DADO	(C, N ou L)
LARGURA DO CAMPO	(número mais alto, número mais longo etc.)
DOCUMENTO-FONTE	(pedido, ação de troca pessoal etc.)
QUEM ENTRA	(empregados do almoxarifado, cobradores etc.)
QUANDO ENTROU	(lote, distribuição, periodicidade etc.)
QUEM EDITA	(quem troca este valor)
QUANDO FOI EDITADO	(o mesmo que "quando entrou", mas para trocas)

Eventualmente quando desenvolvemos o dicionário, queremos incluir campos para REGISTRO, ARQUIVO DE DADOS e DEPARTAMENTO DE CONTROLE. Estes dicionários são instrumentos muito úteis para ajudar o programador e o cliente a definir crítica e analiticamente a natureza do problema na programação.

Os Diagramas Circulares

Os **diagramas circulares** représentam uma forma de descrever visualmente a relação de dados dentro de um banco de dados, de forma que as estruturas de registro possam ser extraídas do todo. Criar um diagrama circular é relativamente simples. Todos os tipos distintos de itens de dados – campos em dBase – são representados por círculos ou ovais rotulados. As relações entre os campos são desenhadas com linhas entre os círculos, como mostramos na Figura 9-1. As setas são usadas simbolicamente para mostrar a natureza dessas relações. Se a relação for de um-para-um, desenharemos uma ponta na seta; no caso de ser um-para-muitos, usaremos duas pontas nas setas.

Quando um diagrama circular estiver completo – e pode ter vários desenhos até que obtemos todos os desenhos sem criar desordem – quase sempre as estruturas do registro tornam-se mais fáceis de definir. Os círculos que são as origens de várias setas de uma única ponta são bons candidatos para serem campos indexados nos registros que compõem os conteúdos da seta. Os círculos tocados pelas setas de dupla ponta são possíveis campos de união associando dois arquivos de dados no desenho de um-para-muitos.

Os diagramas circulares são instrumentos limitados e eu não os convencerei a usá-los como outros autores o fazem. Se como programadores possuímos experiência e boa intuição, então nossos diagramas circulares provavelmente refletirão isto, assim como as nossas estruturas de registros. No caso de ainda estarmos adquirindo experiência ou estarmos incertos em relação a um projeto, provavelmente os diagramas circulares serão confusos e não muito úteis e eu sinto algo problemático em recomendar um guarda-chuva que só possa ser usado em dias de sol. De qualquer forma, se os diagramas nos ajudam a cristalizar o raciocínio, devemos utilizá-los.

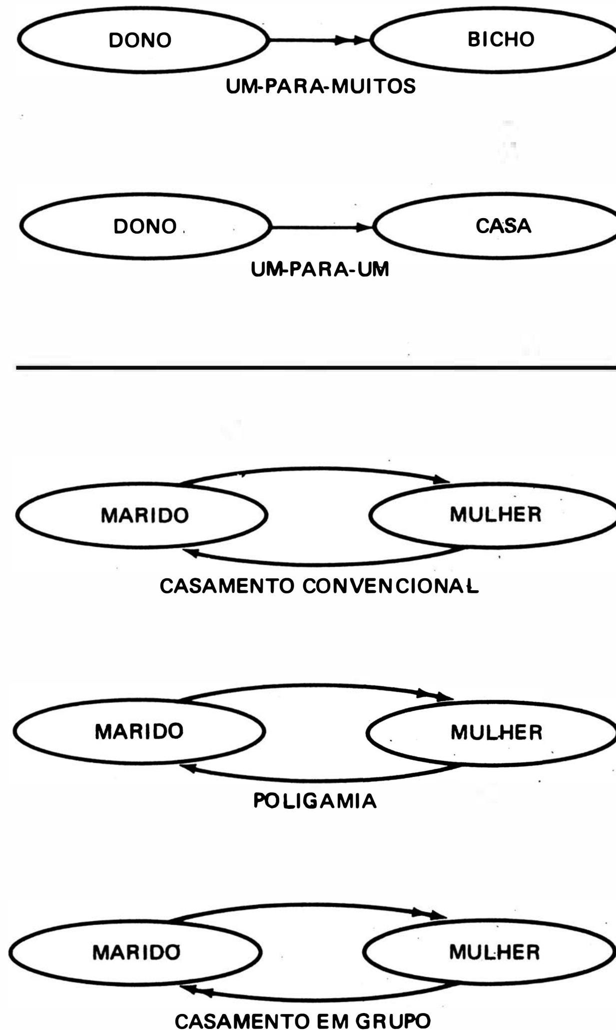


Figura 9-1 Os diagramas circulares usam pontas de setas simples e duplas para indicar as relações de um-para-um e um-para-muitos entre os itens de dados.

Eventos e Assuntos — Uma Alternativa para os Diagramas Circulares

Há uma analogia gramatical que pode ser útil ao definir a estrutura do registro. Um bom registro deve ser uma coleção de campos que representa um assunto. Os conteúdos dos campos devem definir o assunto. Se o registro for de um empregado, os campos devem defini-lo. Não queremos dizer que os registros sempre são sobre descrições de substantivos concretos. Muitos registros relacionam-se a eventos, transações financeiras, movimentos de materiais em um inventário, contratação de empregados, todos são fortes candidatos a uma estrutura simples de registro.

```
ENDDO
RELEASE BIC:CONT
SELECT PRIMARY
*
* fim do arquivo report.cmd
```

Se lermos novamente REPORT FORM na seção seis (Capítulo 4) poderemos reconhecer que o programa de clientes acima poderia ser substituído por uma das formas de relatório dBase. Vamos assumir que usamos REPORT FORM para definir um arquivo ".FRM" chamado "BICHOLST.FRM", que fornece uma lista de animais de estimação. Assim, todo o módulo do relatório poderia ficar estabelecido como:

```
*****************************************
* REPORT.CMD ver 2
*****************************************
*
* esclarecimentos sobre a rotina e suas
* condicoes de entrada
*
STORE TRIM(NOME),SOBRENOME TO DONO:NOME
SET HEAD TO DONO: &DONO:NOME
SELECT SECONDARY
REPORT FORM BICHOLST FOR !(P.SOBRENOME)=S.SOBRENOME
SELECT PRIMARY
*
* fim da versao 2
```

Relatórios de clientes possuem espaço. Agora apresentaremos uma rotina de chamamento.

Pseudocódigo:

```
pegue o nome de um sócio que desejamos
veja se o sócio existe
    se não, volte
veja se o sócio possui bichos
    se não, volte
chame REPORT.CMD
```

Não são necessários os refinamentos. Há apenas dois lugares onde podemos prever o surgimento de problemas. Primeiro, deveríamos ter certeza do que aconteceria se o usuário digitasse o nome dos sócios com letras maiúsculas ou com minúsculas.

Como isto acontece, o programa escrito não possui nenhuma maneira de correção para os erros de maiúsculas e minúsculas. Nosso índice original no campo do sobrenome de SOCIOS, SOBRENDX, não se encaixa na regra "!(sobrenome)". Poderíamos corrigir formando um novo índice para uma provável aplicação real. Eu não fiz isto aqui.

A segunda área na qual assistiremos nossos passos está nas duas etapas de volta. É apenas um detalhe, mas devemos lembrar-nos de voltar a acionar talk on, selecionar a primária novamente, liberar as variáveis de memória apropriadas, e assim por diante, antes de voltarmos – detalhes.

```
***********************************************
* BICHOS.CMD - Relatorio de Tela
***********************************************
* Programa de relatorio para usar com SOCIOS
* e BICHOS relacionados
*
* Fara relatorio de todos os bichos por donos
* confirmando a existencia dos bichos e chamara
* o programa REPORT.CMD
*
* REPORT espera encontrar SOCIOS indexado pelo
* sobrenome na PRIMARIA e BICHOS indexado por
* donos na SECUNDARIA
*
**********************************************
*
SET TALK OFF
* pegue o dono
ERASE
* um truque para fazer o ACCEPT aparecer na linha 11
@ 10,0 SAY " "
ACCEPT [     ENTRE O SOBRENOME DO DONO ] TO MSOBRE
* pegue o registro do socio
FIND &MSOBRE
* o que acontece se nao existe o socio?
IF # = 0
   * beep - o socio nao esta aqui
   ? CHR(7)
   ERASE
   @ 11,8 SAY MSOBRE
   @ 11,$+2 SAY "SOCIO INEXISTENTE - CONFIRA"
   @ $+2,10 SAY "        E TENTE OUTRA VEZ"
   SET TALKON
   RETURN
ENDIF
* ele esta aqui, mas ele tem bichos?
```

```
SELECT SECONDARY
* precisamos procurar nas maiusculas
STORE !(MSOBRE) TO FINDER
FIND &FINDER
IF # = 0
   * nenhum bicho
   ERASE
   @ 11,8 SAY MSOBRE
   @ 11,$+2 SAY "NAO TEM NENHUM BICHO NO ARQUIVO"
   * retorne a primaria e pronto
   SELECT PRIMARY
   SET TALK ON
   RETURN
ENDIF
* se estamos aqui e porque temos um socio com bichos
* mas primeiro ...
SELECT PRIMARY
*
*  AGORA!!
ERASE
DO REPORT
SET TALK ON
*
* fim do arquivo bichos.cmd
```

Não resta nada a não ser tentar. Primeiro temos de ativar a configuração – SOCIOS na primária e BICHOS na secundária, com os índices corretos e enviados propriamente.

```
. USE SOCIOS INDEX SOBRE

. SELECT SECONDARY

. USE BICHOS INDEX DONOS

. SELECT PRIMARY
```

Agora que estamos prontos, vamos testar nosso programa tentando um nome que não está no arquivo SOCIOS e então um sócio que não possui bichos:

```
. DO BICHOS

      ENTRE O SOBRENOME DO DONO : Fidel
```

```
          Fidel  SOCIO INEXISTENTE - CONFIRA
                         E TENTE OUTRA VEZ

. DO BICHOS


      ENTRE O SOBRENOME DO DONO : Covas
      Covas NAO TEM NENHUM BICHO NO ARQUIVO
```

Já possuímos o suficiente de falhas. Testamos as duas rotinas de volta e descobrimos que elas fazem o que imaginávamos – agora para o relatório de um sócio que possui registros de bichos no arquivo de dados na secundária:

```
. DO BICHOS

      ENTRE O SOBRENOME DO DONO : Belo

                  RELACAO DOS BICHOS DE Julio Belo
        -----------------------------------------------------

         Num            Nome do Bicho               Tipo
        -------        ----------------          -----------

         1                Ester                   foca

         2                Poka                    galinha

         3                Leo                     peru
```

OS REGISTROS DE UM-POR-UM – A "OPÇÃO DE UNIÃO"

O dBase possui limitações. Há um limite máximo em cada número significativo – tamanho de um campo, número de registros, o maior número – e a seriedade destas limitações depende do que estamos tentando fazer.

Uma limitação que é completamente fácil de alcançar é a de 32 campos por registro. Os programadores que estão iniciando em dBase, especialmente, tentam definir as estruturas de registros contando todo o banco de dados em somente um arquivo (não é um objetivo ruim, se puder ser realizado, mas sempre impraticável). No próximo capítulo, discutiremos o desenho de bancos de dados e veremos as maneiras de fazer a manutenção, mas mesmo com essas técnicas, surgem as aplicações que precisam de um registro com mais de 32 campos.

Uma solução é desenhar uma estrutura de registro que se estenda por dois arquivos montados nas áreas primária e secundária. Quando desenhamos esta estrutura, depende de nós escrever os programas necessários para a entrada de dados, como fizemos para SÓCIOS e BICHOS. Na verdade, a codificação é mais simples no desenho de um-por-um, porque um circuito externo pode controlar uma operação de acréscimo:

Pseudocódigo:

```
coloque os primeiros 32 campos na primária e o resto
    na secundária
faça a indexação das duas áreas no campo unido para que eles
    fiquem na mesma ordem aparente
inicie o circuito
        inclua a metade da primária
        selecione a secundária
        inclua a metade da secundária
        selecione a primária
inicie um novo registro
pare quando desejar
```

Se quiséssemos colocar em uma rotina que nos desse a certeza da existência de apenas um registro na primária e secundária para cada união, e se esta fosse a única maneira que os registros sempre são incluídos a estes dois bancos de dados de arquivo, então a chance de erro seria pequena.

E a saída? Estes arquivos de relatório sempre requerem arquivos de clientes? Felizmente, não. Há um SET chamado LINKAGE que nos permite usar REPORT FORM (até a sua limitação de 24 campos por relatório) com arquivos de dados nas áreas primária e secundária. Primeiro vamos descrever qual é o efeito quando o comando SET LINK ON é fornecido. Quando LINK está ativado, qualquer comando que pode aceitar um escopo (como "NEXT 5") avançará os arquivos nas áreas primárias e secundárias simultaneamente através de sua ordem aparente.

Isto não se aplica aos comandos SKIP, FIND, LOCATE e GOTO. As áreas primária e secundária permanecerão independentes quando estes comandos de movimentação forem fornecidos.

Os comandos que unem os dois arquivos de dados incluem LIST, DISPLAY, DELETE e REPORT FORM. (Já que o apagamento é uma operação escrita, DELETE só apaga o registro na área selecionada. Se os dois registros unidos estão para ser apagados, o programador deve escrever uma rotina que apaga os dois.) Verifique se realmente entendeu esta aplicação – tudo o que o LINKAGE fará é avançar automaticamente os registros dos arquivos nas duas áreas. Devemos ter certeza de que eles estão na ordem certa, que os dois arquivos iniciam com os registros adequados e, o mais importante, devemos lembrar-nos de desativar o comando SET LINK-OFF quando terminarmos a operação.

Para demonstrarmos a conveniência deste comando, vamos usar dois pequenos arquivos subativados chamados LEFT.DBF e RIGHT.DBF. O LEFT manipulará os dois nomes de campos de SÓCIOS e RIGHT manipulará cidade e estado.

```
. USE LEFT
. DISPLAY STRUCTURE

STRUCTURE FOR FILE:  B:LEFT    .DBF
NUMBER OF RECORDS:   00006
DATE OF LAST UPDATE: 01/01/80
PRIMARY USE DATABASE
FLD       NAME      TYPE WIDTH   DEC
001     NOME         C    006
002     SOBRENOME    C    010
** TOTAL **              00017

. LIST
00001  Mila   Moreira
00002  Monica Medeiros
00003  Pedro  Penedo
00004  Carlos Covas
00005  Davi   Dunas
00006  Julio  Belo

. SELECT SECONDARY
. USE RIGHT
. DISPLAY STRUCTURE
STRUCTURE FOR FILE:  B:RIGHT   .DBF
NUMBER OF RECORDS:   00006
DATE OF LAST UPDATE: 01/01/80
SECONDARY USE DATABASE
FLD       NAME      TYPE WIDTH   DEC
001     CIDADE       C    013
002     ESTADO       C    002
** TOTAL **              00016

. LIST
00001  Sao Paulo     SP
00002  Curitiba      PR
00003  Sao Paulo     SP
00004  Sao Paulo     SP
00005  Aracaju       SE
00006  Recife        PE
```

Já que REPORT FORM realiza uma operação de leitura, ele pode acessar os dois arquivos de uma só vez. Não há problema em definir rapidamente "TST.FRM" que possui nomes de campos desenhados a partir dos dois arquivos:

```
            (* Linha em branco para 'W = 128'', etc. *)
Y
RELATORIO COMBINADO
N
N
10,NOME
>NOME
15,SOBRENOME
                 (* Linha em branco p/nenhum cabeçalho *)
16,CIDADE
<CIDADE
15,ESTADO
<ESTADO
```

Agora, da área primária, vamos operar o relatório TST sem ativar a união. Como veremos, os resultados são perfeitamente lógicos, mas não tão úteis:

```
. SELECT PRIMARY
. REPORT FORM TST     (* aqui rodando sem LINK on *)


    PAGE NO. 00001


                         RELATORIO COMBINADO

        NOME                CIDADE          ESTADO

    Mila      Moreira       Sao Paulo       SP
    Monica    Medeiros      Sao Paulo       SP
    Pedro     Penedo        Sao Paulo       SP
    Carlos    Covas         Sao Paulo       SP
    Davi      Dunas         Sao Paulo       SP
    Julio     Belo          Sao Paulo       SP


. SELECT SECONDARY
. DISPLAY
00001  Sao Paulo     SP
```

O arquivo da secundária nunca sai do primeiro registro. Vamos tentar novamente com a união ativada (depois de voltar para a primária e alcançar o topo, é claro):

```
. SET LINKAGE ON

. SELE PRIM
. REPORT FORM TST


        PAGE NO. 00001


                              RELATORIO COMBINADO

              NOME                  CIDADE            ESTADO

        Mila       Moreira          Sao Paulo         SP
        Monica     Medeiros         Curitiba          PR
        Pedro      Penedo           Sao Paulo         SP
        Carlos     Covas            Sao Paulo         SP
        Davi       Dunas            Aracaju           SE
        Julio      Belo             Recife            PE
```

Devo prevenir sobre os erros que aguardam o programador quando deixar o LINK ativado e tentar outras tarefas. Eles serão muitos e desastrosos. Consideremos, por exemplo, a tentativa de SKIP em um arquivo que possui mais registros enquanto o EOF foi alcançado em um arquivo unido. Isto não funciona. Sempre desative LINK quando terminar uma operação.

ESCOLHENDO UNIÕES

Um **campo de união** é um campo que existe em dois ou mais arquivos de dados. O conteúdo deste campo é a união. A natureza da união entre os dois arquivos é determinada pelos programas que escrevemos. Tudo isto deveria estar claro.

Mas o que faz uma boa união? Deveríamos escolher ou o programa deveria designar uma união? Podemos usar o número de registro do dBase? As respostas para algumas dessas perguntas relacionam-se ao desenho do programa e variam de aplicação para aplicação.

Primeiro vamos observar o que uma união requer. Seja o que um desenho de um banco de dados chamar para a relação dos registros de um-para-um ou um-para-muitos entre os arquivos na primária è secundária, ainda há alguns requisitos específicos.

Para que um programa funcione, a união deve ser absoluta e não ambígua. Em um sistema de controle manual, tal como uma escola (embora muitas delas usem computadores todos os dias), podem existir dois alunos com o nome "Francisco Silva". A secretária pode chamar os professores e descobrir que o aluno "A" está na 8ª série e o "B" está na 2ª.

O dBase não é tão inteligente. Se enviarmos a um programa dBase uma operação em um arquivo de dados no qual dois registros possuam conteúdos idênticos no campo de união, o dBase encontrará o primeiro e fará a operação. As uniões duplicadas não estão sempre incorretas. Mas as uniões duplicadas *acidentalmente* são perigosas, por isso temos de escolher as uniões cuidadosamente para prevenir essas situações.

Apresentaremos algumas características que tornarão uma união como ideal:

1. As uniões corretas são únicas.

2. Os campos de caracteres pequenos (quatro até dez caracteres) são os melhores, embora os campos numéricos são aceitáveis quando o programa designa uniões únicas. Se as uniões possuírem caracteres alfa, é melhor colocá-los todos em maiúsculas.

3. Se os usuários responderem a perguntas para informação dos clientes sobre os registros, pode ser útil escolher uma união que o cliente conheça. Isto pode facilitar as perguntas pelo telefone, quando podemos perguntar ao cliente por seu nº de CGC, por exemplo, em vez do número que o nosso departamento de contas designou a ele.

4. As uniões corretas são "naturais". Elas têm sentido e aproximam-se da lógica manual.

5. As uniões corretas não mudam. Se partes dos números suportam revisões constantes elas não podem ser as melhores uniões em nosso banco de dados de inventário. Se apenas os dois últimos dígitos de um número de peça com dez dígitos mudarem após uma revisão, então usaremos $(parte:num, 1,8) como união e geraremos um campo de união em cada registro para ele.

A maioria das uniões corretas espelha suas utilizações. Nossos registros estão recheados de números de empregados, números de peças, números de contas e centenas de outras uniões que já instalamos em todos os nossos bancos de dados. Estes números foram originariamente designados para unir registros e ainda representam as melhores uniões para usarmos. Use-os.

UNIÕES CALCULADAS

Haverá situações em que os dados que possuímos nos bancos de dados não contêm ou não sugerem uma boa união natural. Nestas situações, devemos designar uma união aos registros do banco de dados ou fazer com que o dBase a designe automaticamente. Uniões calculadas deveriam possuir tantas características de uniões naturais quanto possível, como esta. Queremos calcular uma união para um registro determinado que não pode ser (ou está muito diferente para ser) calculada por qualquer outro registro.

Vamos ilustrar com um exemplo. A seguir encontraremos uma parte de um registro de dados retirado de uma telefonista de uma firma de reembolso postal de um novo pedido de cliente:

Data do Pedido	Nome	Telefone	Cep	União
03/23/83	SIMOES	619/483-6222	92109	LINK

Este cliente pediu quatro itens:

LINK	Ursinho de pelúcia	$49,95
LINK	Botas de couro de cobra	$99,95
LINK	Champagne de sidra	$29,95
LINK	Sabonete perfumado	$ 4,95

Quando a telefonista anotar o pedido, o programa dBase automaticamente colocará um novo registro do cliente no arquivo de pedidos na área primária, colocará quatro novos registros no arquivo de pedidos na secundária e também preencherá os campos LINK com a mesma string nos cinco registros.

A questão, então, é qual string escolher. O sobrenome parece bom, mas não podemos garantir que este é o único "Simões". O CEP não parece prometer muito – os comerciantes tendem a concentrar-se na mesma área. O número do telefone possui uma grande chance de ser único mas pode mudar. Até pior, Simões pode não lembrar do seu telefone antigo depois de dois anos. Apesar disso, dos campos simples, ele é provavelmente o melhor.

As combinações de campos possuem vantagens sobre os campos simples. Consideremos esta expressão como uma união:

```
. REPLACE união WITH !(sobrenome) + $(data, 1, 3) +$(data, 7, 2) +$(tel, 8, 5)
```

em nosso registro exemplificado seria:

```
"SIMOES     03/83-6222"
```

Esta é uma excelente união calculada. Representa uma boa chance de ser única – quantos Simões tivemos em março de 83 do qual o número de telefone terminava em "6222"? E o dado concatenado é do tipo que um cliente poderia lembrar.

Podemos notar que não me importo em usar a função TRIM( ) no campo sobrenome na expressão para a união calculada. Na verdade, isto não faz diferença (a não ser esteticamente) para os resultados de classificação e indexação. Porém, se tivéssemos de considerar os espaços, teríamos usado arbitrariamente os primeiros cinco ou seis caracteres do campo sobrenome.

Um campo calculado também pode ser um campo numerado, embora isto apresente uma outra complicação, o conceito de um "arquivo de memória", que será discutido completamente no próximo capítulo. A idéia é simples designaremos um número para cada registro conforme o surgimento, seja um número do cliente, fatura ou outro qualquer. As uniões de campo-número possuem a vantagem de que todos os FINDs devem ser exatos, economizando algumas linhas de codificação e eliminando a necessidade de usar SET EXACT ON e SET EXACT OFF.

Não há nada de errado com este método desde que não designamos o mesmo número para dois registros. Na maioria dos casos, isto implica escrever codificações em nosso programa que (1) checa para ver se um número já está em uso, e/ou (2) mantém o maior número usado como uma variável de sistema definida pelo usuário. Vamos examinar isso. Imaginem que o cliente, antes do mencionado Simões, tivesse sido o número 18.343. Isto seria armazenado na memória como uma variável numérica chamada "X: HI: NUM", gravado no disco como "SYSVAR MEM" e trazido durante o início do programa pelo comando:

```
. RESTORE FROM sysvar ADDITIVE
```

Assumindo que nunca usamos um número de clientes duas vezes, podemos incluir o registro Simões com as linhas de codificação:

```
APPEND BLANK
STORE x: hi: num + 1 TO x: hi: num
REPLACE link WITH x: hi: num
EDIT    (* Para preencher o resto dos campos primários *)

(*o restante para colocar os pedidos na secundária etc.*)
```

No momento de desativarmos nosso programa, teríamos de gravar este número X: HI: NUM no arquivo SYSVAR; MEM, junto com todas as outras variáveis de sistema que poderíamos ter definido (forneceríamos a eles a característica de prefixo "X:"):

```
SAVE TO sysvar ALL LIKE "X: *"
```

Vimos aqui uma maneira inteligente sobre este comando; usamos o número do registro e do dBase de cada registro primário como uma união. Então poderemos usar GOTO em vez de FIND, que eliminará um arquivo de índice, e a codificação para armazenar a união na secundária será simples:

```
STORE # TO Slink
SELECT SECONDARY
REPLACE link WITH Mlink
```

Isso poderia funcionar, depois de forçar o nosso arquivo na área primária a permanecer estático, mas não é recomendável. Lembre-se de que os números dos registros representam a maneira de o dBase manter a tabulação nos registros e isto não é permanente. Eles são mudados pelo

SORT, DELETE e PACK e pelo COPY TO ou APPEND FROM quando os índices estão em uso. Além disso, a natureza exata dessas mudanças é imprevisível.

Mesmo gravar o número do registro em um campo no registro não é seguro. Vamos imaginar que Simões foi incluído com o número do registro 00404. Substituímos o campo de união com 404 e prosseguimos com nossa operação.

Posteriormente após um PACK, o número do registro de Simões é reduzido para 398, mas a união permanece 404. Seis registros mais tarde, outro 00404 será incluído. É melhor abandonar os indicadores de números dos registros quando ativamos uniões.

A LEITURA DE UM TERCEIRO ARQUIVO DE DADOS

Embora o dBase limite o usuário ou programador a dois arquivos montados em uma área primária e secundária, um terceiro arquivo pode ser aberto e lido, se necessário. O comando envolvido é o APPEND FROM, e ele aceita condições.

Consideremos o seguinte banco de dados: os clientes estão na primária com NOME, UNIÃO e um booleano chamado HAS:PEDIDOS; os pedidos para o mês estão na área secundária pela União.

Um NO-FIND no arquivo de pedidos significa que um pedido está no arquivo do mês anterior.

Cliente	*União*	*tem:pedido*	*união*	*Pedido*	*preenchido*
BAR DO JO	1	.T.	3	Biscoitos	.T.
			2	Cervejas	.F.
			6	Amêndoas	.F.

```
* agora encontramos o pedido para o Bar do Jo
* coloque "1" na varmem
STORE união TO Munião
SELECT SECONDARY
* onde está o pedido do Jo?
FIND &Munião
* talvez não esteja nos pedidos deste mês ainda
IF # = 0
   * ainda estamos no arquivo de pedidos
   APPEND FROM mês passado FOR (união = &Sunião) .AND. .NOT. preenchido
ENDIF no-find if
```

Com certeza, mesmo se gostarmos desta codificação, devemos perguntar que tipo de programa de contabilidade nos permitiria entrar em um novo mês sem automaticamente reposicionar todos os pedidos do mês anterior que não foram preenchidos.

CAPÍTULO 9

Considerações Sobre o Desenho de Banco de Dados

O desenho dos bancos de dados representa um assunto complexo e fascinante e um capítulo não é o suficiente para discuti-lo. Nenhum livro sobre uma linguagem poderia colocar todas as ramificações de como estruturar um banco de dados. Felizmente, não precisamos fazer isto. A maioria dos procedimentos complexos desenvolvidos na ciência de gerenciamento de banco de dados surgiu dos computadores de grande porte e encontram-se fora do mundo do dBase.

O dBase é designado a usuários e tarefas únicas (embora tenha sido adaptado por alguns programadores para sistemas operacionais de multi-usuários tal como o MP/M Digital Research). De fato, são as limitações do dBase que mais afetam o desenho de um sistema. O fato de o dBase limitar-se a dois arquivos na memória ao mesmo tempo e consumir tempo de operação ao atualizar arquivos e índices, indica que os sistemas tendem a ser desenhados com cuidado.

O que discutiremos, entretanto, são as considerações sobre o desenho que podem mostrar a diferença entre um bom sistema e um quase impraticável de operação lenta.

OTIMIZAÇÃO NA ESTRUTURA DO REGISTRO

Nenhuma característica de desenho simples representa diferença na qualidade de um programa do que possuir um conjunto racional e bem desenhado de estruturas de registro em um banco de dados.

Esta é a única etapa que nunca deveria ser feita sem reflexão – sentar e digitar CREATE sem planejar, representa um passo para possíveis problemas, mesmo em um programa simples. Nossos objetivos são desenhar registros nos vários arquivos no banco de dados para que:

1. A programação seja simples e direta.
2. A velocidade do programa seja aumentada, eliminando ao mínimo os movimentos do banco de dados às entradas e saídas na memória.
3. A utilização do espaço do disco seja eficiente, assim os registros não terão espaço vazio e não conterão nada além do necessário.

Para alcançar uma estrutura de registro ideal, vários instrumentos de planejamento foram desenvolvidos durante anos. Entre eles, os mais aplicados em dBase são: "o dicionário de dados", "os diagramas circulares" e o conceito de "arquivo nivelado".

Os Dicionários de Dados

Os grandes bancos de dados possuem **dicionários de dados** no computador no qual cada item de dado no banco de dados é digitado com significado preciso: quem o utiliza, quem é responsável pela entrada, e quem pode ou não vê-lo. Estes dicionários, entre outras coisas, previnem a entrada do mesmo dado com dois nomes diferentes (sinônimos) ou do mesmo nome usado para dois dados diferentes (homônimos).

Isto pode parecer um desastre para um programa simples em dBase, mas o fato de criar um dicionário de dados implica um tempo bem empregado, especialmente para consultores que desenvolvem programações de clientes. Ocasionalmente, todos os consultores passam pela experiência de destruir um programa que decepciona o cliente. Na maioria dos casos, o problema poderia ser prevenido se o consultor e os usuários finais tivessem elaborado um dicionário de dados compreensível no qual o termo levasse ao dado, quem o utilizou e como.

No caso do dBase, a etapa definitiva na determinação de uma ótima estrutura de registro implica a determinação dos nomes de campo, tipos e larguras e da formalização das associações de campos. Se fôssemos listar o histórico dos dados obteríamos: qual é o documento-fonte, quem o inseriu e quando, quem o editou e quando; e a um nível inferior, quais os relatórios em que ele aparece (porque sempre assumiremos que se podemos obter o dado correto *dentro* do banco de dados, como resultado obteremos os relatórios corretos).

Quando desenvolvemos o dicionário de dados, ele deve circular entre todos os usuários finais e supervisores que o utilizarão no banco de dados, para que cada um possa verificar as idéias e comentários dos outros, e até discordar ou discutir. É mais do que uma política. Quase sempre as pessoas não têm idéia de quais dados seus colegas estão se baseando no trabalho a ser desenvolvido.

Os cobradores em uma doca de embarque podem estar gravando as datas das entregas achando que é necessário para o estoque. Em contrapartida, o estoque pode estar lançado pelo número da fatura. E o planejamento de materiais pode controlar as novas entregas pelos números originais de requisição.

Deve haver boas razões históricas para tudo isso, mas o programador não pode escrever um programa que encontrará a aprovação dos três departamentos sem obter a concordância sobre quais itens de dados seriam os campos-chaves, quais seriam os nomes etc.

Um dicionário de dados pode ser criado através do dBase. Uma possível estrutura seria:

NOME DE CAMPO	
DEFINIÇÃO	(o que representa exatamente este dado)
TIPO DE DADO	(C, N ou L)
LARGURA DO CAMPO	(número mais alto, número mais longo etc.)
DOCUMENTO-FONTE	(pedido, ação de troca pessoal etc.)
QUEM ENTRA	(empregados do almoxarifado, cobradores etc.)
QUANDO ENTROU	(lote, distribuição, periodicidade etc.)
QUEM EDITA	(quem troca este valor)
QUANDO FOI EDITADO	(o mesmo que "quando entrou", mas para trocas)

Eventualmente quando desenvolvemos o dicionário, queremos incluir campos para REGISTRO, ARQUIVO DE DADOS e DEPARTAMENTO DE CONTROLE. Estes dicionários são instrumentos muito úteis para ajudar o programador e o cliente a definir crítica e analiticamente a natureza do problema na programação.

Os Diagramas Circulares

Os **diagramas circulares** représentam uma forma de descrever visualmente a relação de dados dentro de um banco de dados, de forma que as estruturas de registro possam ser extraídas do todo. Criar um diagrama circular é relativamente simples. Todos os tipos distintos de itens de dados – campos em dBase – são representados por círculos ou ovais rotulados. As relações entre os campos são desenhadas com linhas entre os círculos, como mostramos na Figura 9-1. As setas são usadas simbolicamente para mostrar a natureza dessas relações. Se a relação for de um-para-um, desenharemos uma ponta na seta; no caso de ser um-para-muitos, usaremos duas pontas nas setas.

Quando um diagrama circular estiver completo – e pode ter vários desenhos até que obtemos todos os desenhos sem criar desordem – quase sempre as estruturas do registro tornam-se mais fáceis de definir. Os círculos que são as origens de várias setas de uma única ponta são bons candidatos para serem campos indexados nos registros que compõem os conteúdos da seta. Os círculos tocados pelas setas de dupla ponta são possíveis campos de união associando dois arquivos de dados no desenho de um-para-muitos.

Os diagramas circulares são instrumentos limitados e eu não os convencerei a usá-los como outros autores o fazem. Se como programadores possuímos experiência e boa intuição, então nossos diagramas circulares provavelmente refletirão isto, assim como as nossas estruturas de registros. No caso de ainda estarmos adquirindo experiência ou estarmos incertos em relação a um projeto, provavelmente os diagramas circulares serão confusos e não muito úteis e eu sinto algo problemático em recomendar um guarda-chuva que só possa ser usado em dias de sol. De qualquer forma, se os diagramas nos ajudam a cristalizar o raciocínio, devemos utilizá-los.

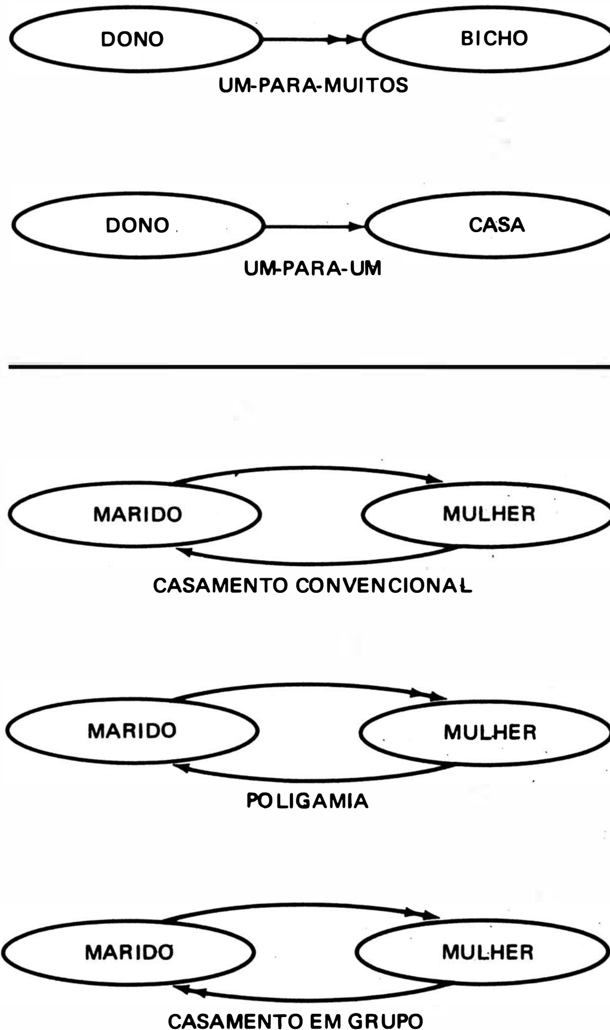


Figura 9-1 Os diagramas circulares usam pontas de setas simples e duplas para indicar as relações de um-para-um e um-para-muitos entre os itens de dados.

Eventos e Assuntos — Uma Alternativa para os Diagramas Circulares

Há uma analogia gramatical que pode ser útil ao definir a estrutura do registro. Um bom registro deve ser uma coleção de campos que representa um assunto. Os conteúdos dos campos devem definir o assunto. Se o registro for de um empregado, os campos devem defini-lo. Não queremos dizer que os registros sempre são sobre descrições de substantivos concretos. Muitos registros relacionam-se a eventos, transações financeiras, movimentos de materiais em um inventário, contratação de empregados, todos são fortes candidatos a uma estrutura simples de registro.

Mas é muito raro encontrarmos bons registros que contenham mais de um evento ou assunto. Podemos fazer um teste para sabermos se um campo pertence ou não a um determinado registro em um banco de dados. Em primeiro lugar, sobre qual evento ou assunto o registro se refere? Em segundo, o item naquele campo nos diz algo sobre o evento ou assunto para mostrar por que ele pertence àquele registro? Se a resposta é sim, o caminho está correto.

O Arquivo Nivelado

O conceito de **arquivo nivelado** representa provavelmente uma das idéias mais úteis para o desenho da estrutura de registro. Vamos ilustrar com um exemplo. Desenhamos um programa dBase para anotarmos os pedidos de uma loja de peças para automóveis. Eles nos forneceram uma cópia do pedido que preenchem (veja Figura 9-2):

NOME:
ENDEREÇO:
CIDADE: EST: CEP: TEL:

# PEÇA	QTD.	DESCRIÇÃO	PREÇO @	TOTAL
1				
2				
3				
4				
5				

Figura 9-2 Formulário de pedido de uma loja de peças de automóveis – a base para uma estrutura de registro de um banco de dados.

Ótimo! A estrutura do registro já está definida. É tão fácil programar! Mas façamos algumas perguntas assim mesmo, só para nos mostrarmos cuidadosos.

– Há lugar para cinco peças, dissemos. Este é o pedido médio?

– Não, geralmente um, dois, às vezes três.

– Hum! Sempre há pedidos para cinco peças?

– Sim, às vezes até 20 de um distribuidor. Mas a maioria é de apenas dois ou três; 20 é um grande pedido.

– Vinte! Como fazem um pedido de 20? Só há lugar para cinco peças.

– Bem, quando um pedido ultrapassa os cinco, continuamos em uma outra folha e as grampeamos juntas. Assim não precisamos preencher a parte de cima novamente.

O que temos, neste caso é um "arquivo recheado". Verificando o pedido de perfil, a estrutura do registro mostraria um volume grande de pedidos (veja a Fig. 9-3, vista de perfil).

Neste arquivo recheado, cada pedido pode ser visualizado como:

```
PEDIDO #
PEÇA #
QTD.
DESCRIÇÃO
PREÇO @
TOTAL
```

Os arquivos recheados podem ser transformados em arquivos nivelados quebrando a estrutura do registro em dois registros e unindo os dois arquivos em uma relação de um-para-muitos:

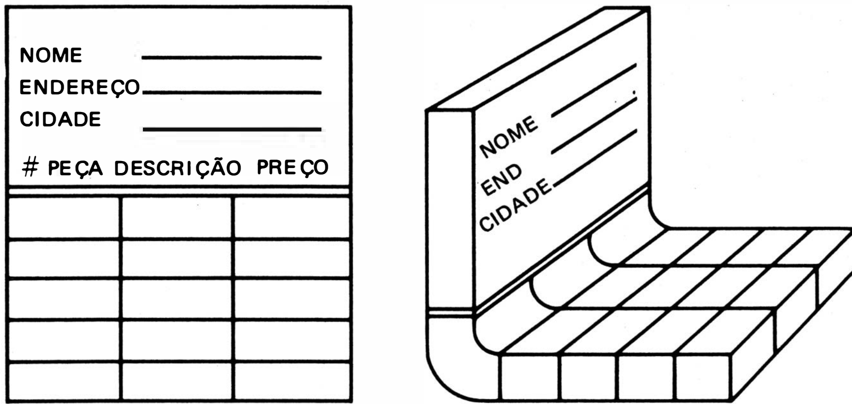
Primária	**Secundária**
NOME	UNIÃO
ENDEREÇO	# PEDIDO
CIDADE	# PEÇA
EST	QTD
CEP	DESCRIÇÃO
TEL	PREÇO
UNIÃO	TOTAL

Nem todos os casos de arquivos recheados são tão óbvios, mas eles geralmente podem ser divididos. Apresentaremos duas sugestões para reconhecermos quando nossos arquivos não são nivelados.

1. Suspeite da necessidade de divisão se os registros em um arquivo de dados possuírem vários campos em branco em uma alta percentagem dos registros.
2. Procure maneiras de elaborar arquivos separados se reutilizar nomes de campo dentro de um registro, como em SHP: DATA1, SHP: QTD1, SHP: DATA2, SHP: QTD2.

Não permita, porém, que o capricho na construção dos arquivos acabe por esconder a praticidade. Imagine um arquivo de dados de alunos no qual deixamos espaço para as atividades – futebol, banda, teatro, clube de xadrez, num total de dez. Montamos o registro com os dados do aluno e dez campos booleanos, um para cada atividade:

```
ALUNO #
{DADOS DO ALUNO}
FUTEBOL
BANDA
.
.         (*10 atividades*)
```



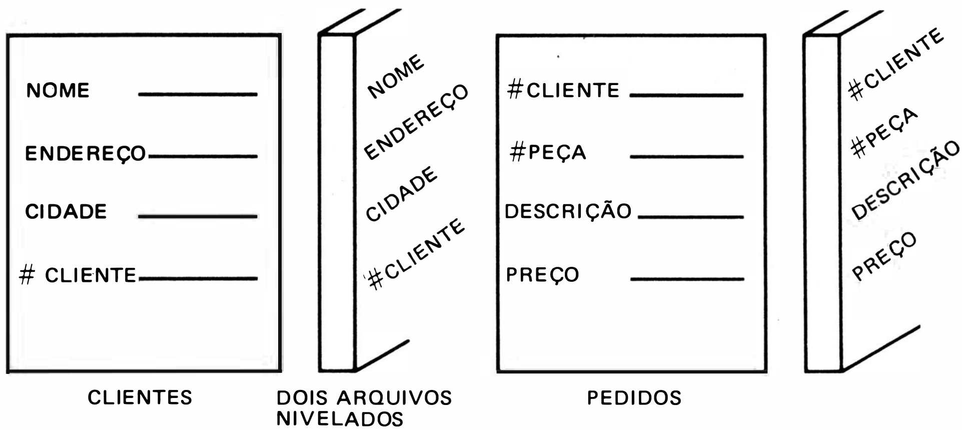


Figura 9-3 Arquivos nivelados e recheados.

Veja! Um arquivo recheado. Vamos quebrá-lo em um arquivo de alunos unido com um de atividades. Observemos primeiro a estrutura do registro no arquivo de atividades. Ele terá o número do aluno como união, talvez oito caracteres e então o nome do campo de atividade, talvez cinco, bem como um byte (caractere) de espaço reservado para a marca de apagamento. Serão necessários 14 caracteres para cada registro comparados com os dez caracteres do arquivo de alunos – e sabemos que quase todos os alunos terão pelo menos uma atividade.

Neste caso, quebrando o arquivo em dois arquivos nivelados, poderíamos complicar a nossa tarefa de programação, diminuir a velocidade do programa (todas aquelas seleções primárias e secundárias) e perder espaço no processo. De fato, a simplicidade do programa e as considerações sobre a velocidade fazem com que fique claro dizer que é melhor ter um arquivo recheado de vez em quando, contanto que as estruturas dos registros funcionem e não se desperdice espaço.

RESPONDENDO A PERGUNTAS FREQÜENTES

Um princípio do desenho de bancos de dados diz que os itens não devem ser repetidos. Pode haver muitos caminhos para um determinado item de dado – quais produtos temos em excesso no estoque, o que compramos numa loja e outra, quais itens de estoque foram os últimos a serem recebidos – mas todos esses caminhos deveriam levar ao mesmo campo no mesmo registro.

Isto representa muito mais do que uma questão de espaço. Imaginemos que duplicamos nosso arquivo de cliente em dois lugares. Uma secretária, trabalhando fora do expediente sem ninguém para consultar, datilografou um formulário para um cliente endereçado para "Godoi Carlos" e a etiqueta de correspondência para "Carlos Godoi". Quem saberá qual é o certo? Quando duplicamos os dados indiscriminadamente dentro de um banco de dados, os erros às vezes tomam-se incorrigíveis.

Estas colocações foram utilizadas para apresentarmos a próxima idéia. Às vezes é vantagem duplicar dados dentro de um banco de dados para respondermos perguntas freqüentes mais rapidamente. Quando agirmos desta maneira, deveremos fazer o programa copiar o dado duplicado automaticamente e *nunca* usar a imagem duplicada para uma saída. "Perguntas freqüentes" representam uma consideração importante no desenho de banco de dados. Quando desenhamos o nosso banco de dados ou o do cliente, devemos fazer os futuros usuários anotarem diariamente as perguntas que serão respondidas pelo computador quando o banco de dados estiver rodando. O programa será bom se não puder responder estas perguntas?

Essa lista nos mostrará quais campos devemos manter nas indexações secundárias, já que em muitas situações, para obtermos as respostas, apenas ativamos o índice ou usamos um FIND. Porém, algumas vezes, uma pergunta envolve dois arquivos de dados. Pior ainda, eles podem ser arquivos de dados que normalmente não são carregados juntos.

Um exemplo seria a empresa de pedidos por correspondência, mencionada anteriormente. Os telefonistas sempre atendem a clientes que desejam saber onde estão seus pedidos. É muito bom termos um arquivo de pedidos indexados pela união, mas poderia aumentar a velocidade das respostas se o sobrenome do cliente fosse transcrito em cada pedido. Apesar disso, não devemos imprimir as etiquetas de correspondência a partir do arquivo de pedidos – alguém pode ter corrigido os nomes dos clientes no arquivo de clientes. E sempre devemos compor uma lista de perguntas freqüentemente feitas como parte da pesquisa para o desenho de nossos bancos de dados.

OS BOOLEANOS NOS REGISTROS

A utilização mais óbvia dos valores booleanos é expressar claramente a condição falsa/verdadeira de um item de dado dentro de um registro. Eles possuem outras duas utilizações que ocorrem freqüentemente.

Os booleanos podem ser incluídos numa estrutura de registro para o preparo inicial programático. Por exemplo, se um programa for desenhado para que atenda a um objetivo de auditoria, qualquer campo poderá ser gerado simplesmente incluindo um booleano chamado ANT:REG em cada registro. Se ANT:REG for ignorado quando novos registros forem acrescen-

tados (de fato, ANT: REG deveria ser mantido fora do alcance do usuário) então ele será deixado em branco nos novos registros e será falso. Mais tarde um arquivo de comando chamado AUDTRAIL.CMD poderia ser chamado, com estas linhas:

```
SET PRINT ON
DO WHILE .NOT. EOF
   IF .NOT. old.reg
      DISPLAY
   ENDIF
   SKIP
ENDDO
SET PRINT OFF
REPLACE ALL old:rec WITH t
```

O segundo uso para os booleanos de preparo inicial serve para assinalar a existência de um registro em outro arquivo de dado não usado. Consideremos um programa de inventário de estoque. Os produtos entram e saem com um registro de operação para cada movimentação, contendo o número do produto, quantidade, data, projeto ou contagem e assim por diante. A maioria das transações é rotineira e seria um desperdício de espaço possuir campos de OBSERVAÇÕES, embora algumas a necessitem. Entretanto desenhamos um arquivo OBSERV.DBF que será carregado na área secundária quando necessário, assim podemos digitar uma observação para alguma transação que ficaria confusa sem ela.

Deveríamos fazer o programa pesquisar o arquivo de observações para cada transação ao elaborar o relatório? Isto diminuiria a velocidade. Em vez disto, colocaremos um campo booleano no registro chamado OBSERV. Um valor verdadeiro neste campo considerará a existência de uma observação para este registro, desta maneira nosso programa saberá quando deverá procurá-la.

AUMENTANDO OS CAMPOS

Quando discutimos sobre arquivos de dados unidos, falamos da limitação de 32 campos por registro. Muitas vezes esta limitação acontece somente pelo fato de que um desenho de registro foi insuficiente para utilizar ao máximo as informações armazenadas nos campos já definidos.

Tomando como exemplo o registro do aluno citado anteriormente, que possuía dez booleanos para as atividades dos alunos, provavelmente funcionaria melhor com um campo de codificação de atividade de dois para quatro caracteres. Mesmo um simples campo de dez caracteres salvaria campos preciosos para outras utilizações dentro do registro. Para obtermos o efeito desta operação, o programador ativaria uma codificação de uma única letra para cada atividade. Prevenindo os usuários de enganos, incluiríamos estas codificações no rodapé da tela do arquivo de formatação usado para os APPENDs e EDITs dos alunos. O arquivo de formatação possuiria também uma cláusula PICTURE no GET apropriado para colocar em maiúsculas todas as entradas.

Assim podemos imaginar um campo de atividades para um aluno, com futebol, banda e teatro tendo como conteúdos "FBT". Quando o aluno entrasse para o clube de xadrez, usaríamos a seguinte codificação:

```
* um GET ou um WAIT TO também funciona
ACCEPT "Digite uma nova codificação de atividade" TO nova:atividade
* concatenação sem a intervenção de espaços
REPLACE aluno: ativ WITH aluno: ativ— ! (nova*ativ)
```

A codificação que checaria se o aluno participou da atividade determinada usaria a função booleana da substring:

```
* qual atividade estamos pesquisando?
@ 11, 11 SAY "Insira a atividade para confirmar" GET ask :ativ PICTURE "!"
READ
IF "&ask: ativ" $ (aluno: ativ)
   @ 11, 12 SAY "O aluno participa da atividade"
ENDIF
```

A INDEXAÇÃO CRIATIVA E AS CHAVES DE CLASSIFICAÇÃO

Em uma classificação, o dBase permite apenas um campo-chave de cada vez, e é lógico, apenas uma indexação pode colocar ordem ao arquivo de dados, embora até sete indexações possam estar em uso (enviada automaticamente). Concluiríamos que se desejarmos classificações duplas, deveremos classificar duas vezes, ou classificar e depois indexar. A mesma lógica nos levaria a acreditar que a ordem de classificação dupla não pode ser imposta a uma indexação.

Para deixar claro sobre o que queremos dizer com "dupla classificação", consideremos um arquivo de dados de pedidos de venda que possui um campo cidade e um valor. Poderíamos organizar nossos registros "por cidade e por valor", assim a ordem aparente seria:

Cidade	**Valor ($)**	
Brasília	1	
Brasília	20	
Brasília	300	(*Cidades alfabeticamente—*)
Cuiabá	1	(*o valor subindo*)
Cuiabá	20	
Cuiabá	300	
Fortaleza	1	
Fortaleza	20	
(etc.)		

Vamos observar este arquivo que contém estes oito registros classificados, depois de algumas entradas aleatórias ao seu final:

```
. list
00001  Brasilia      1
00002  Brasilia     20
00003  Brasilia    300
00004  Cuiaba        1
00005  Cuiaba       20
00006  Cuiaba      300
00007  Fortaleza     1
00008  Fortaleza    20
00009  Cuiaba        5
00010  Aracaju      22
00011  Brasilia     23
00012  Sao Paulo    80
00013  Aracaju       1
```

Para obtermos a inserção correta dos registros de 9 a 13 ao índice "por cidade e valor", vamos utilizar um comando de indexação concatenada:

```
. INDEX ON (cidade + (STR(valor,5.0))) TO demo1
00013 RECORDS INDEXED
```

Note que o valor numérico VALOR é transformado para uma string antes de ser concatenado (incluso como uma string) à cidade. Isto previne um choque na formação do índice.

Apresentaremos o arquivo de dados listado após colocarmos a indexação em uso:

```
. LIST
00013  Aracaju       1
00010  Aracaju      22
00001  Brasilia      1
00002  Brasilia     20
00011  Brasilia     23
00003  Brasilia    300
00004  Cuiaba        1
00009  Cuiaba        5
00005  Cuiaba       20
00006  Cuiaba      300
00007  Fortaleza     1
00008  Fortaleza    20
00012  Sao Paulo    80
```

Uma vez que possuímos esta indexação, podemos incluir registros ao conteúdo e cada registro novo será colocado automaticamente na ordem apropriada no arquivo.

As expressões utilizadas são versáteis. Poderíamos, por exemplo, visualizar nosso arquivo de dados por cidade e valor com os valores na ordem decrescente. Para isto, usaríamos o mesmo princípio de um índice concatenado, mas antes de converter VALOR em uma string, faríamos uma operação aritmética:

```
. INDEX ON (cidade + (STR(1000-valor,5,0))) to demo2
00013 RECORDS INDEXED
```

Aqui está a indexação listada:

```
. LIST
00010 Aracaju       22
00013 Aracaju        1
00003 Brasilia     300
00011 Brasilia      23
00002 Brasilia      20
00001 Brasilia       1
00006 Cuiaba       300
00005 Cuiaba        20
00009 Cuiaba         5
00004 Cuiaba         1
00008 Fortaleza     20
00007 Fortaleza      1
00012 Sao Paulo     80
```

Vamos colocar alguns conceitos sobre criação de índices múltiplos:

1. Organize a expressão de indexação da esquerda para a direita com a classificação primária (ou "última classificação" ou a "classificação mais importante") à esquerda da expressão.
2. Subtraia os componentes aritméticos de uma constante mais alta se desejá-los em ordem decrescente.
3. Converta todos os números em strings antes de concatenar (lembre-se de que datas já são strings).
4. Não use TRIM( ) nos campos ou "–" nas operações de concatenação.
5. Se quisermos uma classificação com qualquer motivo – uma que use expressões ou campos múltiplos – podemos indexar com uma expressão e então usar COPY TO <novo nome de arquivo> com aquela indexação em uso. O novo arquivo estará na ordem de indexação.

ATIVANDO AS INDEXAÇÕES

Aparecem ocasiões em que queremos encontrar um registro rapidamente em um campo que não é a chave da indexação primária. Nestas situações, há uma técnica especial para ativar as indexações.

Vamos supor que mantemos o arquivo de dados SÓCIOS, com a indexação de sobrenome como um índice primário e a indexação da cidade como um secundário. Em algum lugar dentro do programa o usuário quer pesquisar pela cidade:

```
*obtenha a cidade a ser pesquisada
ACCEPT "Qual cidade procurar" TO find: cidade
*ative as indexações
SET INDEX TO socid, socsobre
*vá para o registro
FIND &find: cidade
IF # = 0
     *rotina no-find
ENDIF
*
*AQUI ESTÁ O TRUQUE
*
*salve o indicador
STORE #TO indicador
*coloque as indexações de volta
SET INDEX TO socsobre, socidade
*volte ao registro encontrado
GOTO indicador
```

O fato de ativarmos ou usarmos uma nova indexação, faz o indicador do registro trazer o primeiro naquela ordem. Por isso, o número do registro do find deve ser gravado antes de reativarmos a indexação. Além disso, não esqueça que se as rotinas NO-FIND usam LOOP, devemos voltar às indexações na ordem de entrada antes do circuito (veja o Capítulo 10 sobre depuração).

REVISÃO DA PROGRAMAÇÃO ESTRUTURADA

A **programação estruturada** é um título que inclui muitas técnicas e procedimentos diferentes, incluindo o refinamento por passos, a programação modular ou estrutural e o diagrama de fluxo. A filosofia citada acima é sempre a mesma. Os programas são desenhados de forma estruturada de acordo com um planejamento para que possuam mais chances de funcionar bem do que aqueles construídos sem estruturação, como algo pouco duradouro. Mas os programadores não gostam de planejar, pois não é tão divertido quanto escrever codificações.

Apresentaremos duas técnicas que funcionam bem com o dBase e não são tão formais: refinamento por passos e diagrama de fluxos de dados.

O Diagrama de Fluxos de Dados

Há alguns anos, os diagramas de fluxo eram odiados nos livros de texto de programação, e nenhum aluno escapava mesmo no curso mais básico de ter que desenhar incompreensíveis linhas confusas, caixas e um círculo rotulado "início". Se na verdade aprendemos algo sobre programação a partir destes diagramas de fluxo, isto é discutível. Pelo fato de estes diagramas serem difíceis de desenhar, eles perderam um pouco de sua utilização, principalmente em terminais de computador.

O problema verdadeiro com esses diagramas antigos é que eles lidavam com o fluxo de controle dentro do programa em vez do fluxo de dados dentro do trabalho. Se estivermos codificando em uma linguagem como BASIC, então pode ser uma ajuda ver um losango rotulado "IF X>Y THEN GOTO 1040", mas não nos informa se nossos depósitos bancários serão incluídos ao nosso balanço ou se serão subtraídos deles.

O diagrama de fluxo em dBase poderia ser chamado método de "entrada para saída". Embora pareça muito formal, o procedimento pode ser muito simples. Antes de iniciarmos a codificação do programa, colecionamos todos os documentos de entrada (de preferência os formulários que estiverem em uso), e de saída (de preferência os relatórios antigos) para os bancos de dados e empilhamos um ao lado esquerdo e outro ao lado direito de nossa mesa. Agora em uma folha, entre duas pilhas de entrada e saída, iniciamos delineando estruturas de registro e escrevemos pequenas descrições dos trabalhos que precisam ser realizados para movimentar os dados através das estruturas de registro da esquerda para a direita. Se nos sentirmos impedidos, devemos refletir, "Como um empregado faria isto?" ou melhor, "Como os empregados fazem isto agora?"

Conforme prosseguimos com o exercício, as descrições do trabalho poderiam ser transformadas em nomes de arquivo de comandos, tais como ADDSTOCK.CMD ou ABCDEF.CMD. Muitas das descrições desses trabalhos serão os pseudocódigos e os nomes dos arquivos de comandos podem ser agrupados em uma folha que aproximaremos da estrutura do menu do programa.

Neste processo devemos tentar manter os trabalhos, os menus e o fluxo de dados o mais perto possível da maneira lógica com que as pessoas lidam com os dados. Ocasionalmente os programadores são tentados a eliminar alguma parte do processo empregando pequenos cortes mas de boa aparência ou diminuindo um cálculo utilizando alguns truques de números. Eu recomendo que resistam às tentações. Quando os programas necessitam de modificações durante a operação, os truques e os pequenos cortes possuem a propriedade de fazer surgir o back up, geralmente bem no meio do módulo que necessita a mudança. Eles não são apenas difíceis de imaginar (por que o programador foi tão negligente para comentar sua codificação?) mas as etapas de truques tendem a ser mais "rígidas" e difíceis para adaptarmos a novos requisitos.

O Refinamento por Passos

O refinamento por passos poderia também ser chamado método de "dividir e conquistar". A idéia é: iniciamos nossa programação no topo e decidimos as inter-relações de tudo antes de nos determos em qualquer detalhe no nível inferior. Quando estamos operando designamos nomes a alguns módulos e procedimentos que não possuem codificação, mas não nos envolvemos muito com isto.

O refinamento por passos leva-nos à autoconfiança. Quando passamos por cada módulo, temos a certeza de que executaremos como um estalo quando voltarmos a ele – e se isto não acontecer, simplesmente submetemos o módulo a um refinamento por passos. Em dBase, estes refinamentos iniciam durante o diagrama de fluxos de dados quando as tarefas são segregadas aos menus e nomes para procedimentos selecionados. Se decidirmos que em um desenho de um programa de inventário STOCK.CMD será um lote de comandos para incluir um novo estoque e que será chamado ADDMENU.CMD, a codificação dos dois arquivos de comandos pode então ser colocada de lado até que EDITMENU.CMD, RPTMENU.CMD e SCRENRPT.CMD tenham sido delineados. É claro que enquanto o MAINMENU.CMD não tenha sido desenhado, não podemos delinear os arquivos de comandos.

As programações estruturadas e de refinamento por passos recompensam o programador que as utiliza, além do que poderia ser esperado. Por exemplo, quando observamos os nomes dos módulos acima, percebemos que o ADDMENU e o EDITMENU conterá procedimentos que poderiam partilhar os mesmos formatos de arquivos.

Podemos observar também que o EDITMENU e o SCRENRPT necessitarão de codificação que possa colocar o indicador de registro em um registro determinado – algum tipo de FINDREC.CMD. Assim, se não soubermos como FINDREC funcionará ficaremos alertas ao fato de querermos que ele seja uma codificação generalizada. Quando nos lançarmos ao EDITMENU, provavelmente teremos de reinventar um módulo find e refazer um formato de tela. Um ditado em xadrez que, como a programação, é um jogo de raciocínio, diz que mesmo um planejamento ruim é melhor do que não planejar. Os diagramas de fluxo, os refinamentos por passos e os diagramas circulares, e todos os outros truques de programação estruturada são apenas técnicas para nos auxiliar a planejar.

Os programadores bem-sucedidos são aqueles que aprenderam que a parte mais importante e sempre a mais difícil em escrever um bom programa aparece antes de ligarmos o computador. Para escrevermos um bom programa em dBase, devemos entender o trabalho a ser desenvolvido e ter uma idéia clara de como o dBase pode ser operado para realizar aquela tarefa utilizando os arquivos de comandos, linha por linha.

O PROGRAMA PARADIGMÁTICO

É uma banalidade conhecida dizer que a estrutura da linguagem que usamos é a mais importante influência no tipo de programa que escrevemos. Se o único instrumento de carpintaria que usamos é um martelo, então a maioria de nossas carpintarias envolverá marteladas.

O dBase II foi desenvolvido especialmente para manipular indexações, arquivos de dados com extensão fixa e isto é o que ele faz melhor. A "linguagem" do dBase é mais forte em

armazenamento de dados, recuperação e intercâmbio. Dentro das limitações das máquinas e sistemas operacionais que são montadas, ele é quase imbatível nestas áreas.

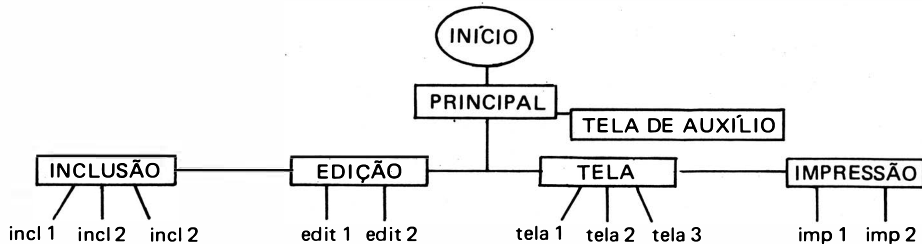
O dBase é mais fraco nas áreas de funções aritméticas transcendentais e em estatística (quase nada). Sua limitação nos tamanhos das estruturas dos dados são verdadeiras e o limite de dois arquivos (nas áreas primárias e secundárias somente) dificultam a programação.

Como conseqüência das fraquezas e potencialidades do dBase, a grande maioria dos programas é semelhante. Nesta seção, apresentaremos a estrutura de um programa paradigmático. O que ele faz não é importante. Sua estrutura pode ser copiada e adaptada para qualquer aplicação do dBase que o principiante poderia desejar. Iniciaremos observando o programa no ponto de entrada e o seguiremos seqüencialmente, de cima para baixo, observando a maneira que deveríamos desenhá-lo.

```
START.CMD
Chamado do
MODO INTERATIVO A > DBASE START
Faça!
mostre a mensagem de finalidade
obtenha as datas se necessário
ative o que for desejado para a intensidade, alarme etc.
Pare!
trabalhe dentro de um circuito
use qualquer arquivo de dados ou indexações
PRINMENU.CMD
Chamado do
start.cmd (start.cmd está completo e o arquivo está fechado)
Faça!
estabeleça um circuito infindável
mostre os menus dos submenus e saia
pode ter tela de auxílio disponível
escolha e chame o submenu
Pare!
use qualquer arquivo de dados ou indexações
ADDMENU.CMD
Chamado do
mainmenu.cmd
Faça!
inicie o circuito
mostre o menu de opções para incluir, tela de auxílio e saia
abra arquivos de dados apropriados e indexações com:
   opção de inclusão em lotes como para cada arquivo
   inclusões individuais para cada arquivo de dados
   inclusões individuais ou em lotes para arquivos unidos
feche todos os arquivos e indexações e:
```

    volte ao menu principal ou saia para o sistema operacional
Pare!
altere ou visualize dados
EDITMENU.CMD
Chamado do
prinmenu.cmd
Faça!
inicie o circuito
mostre o menu com as seguintes opções
use arquivos necessários e indexações, unidos ou individuais, para:
    encontrar arquivos específicos ou pares de registros
    editar os conteúdos ou apagar registros
    pesquise e limpe as duplicatas
pode ter tela de auxílio
feche todos os arquivos e indexações e volte ao menu principal ou
    ao sistema operacional
pode usar pack ou REINDEX
Pare!
inclua novos registros
SCRENRPT.CMD
Chamado do
prinmenu.cmd
Faça!
inicie o circuito
mostre o menu de opções de relatório de tela
use os arquivos e as indexações para:
    encontrar registros ou pares de registros
    ativar as indexações para encontros rápidos nas chaves alternadas
feche todos os arquivos de dados e indexações e volte ao menu
    principal ou ao sistema operacional
Pare!
inclua os registros
atualize os registros
imprima
PRINTRPT.CMD
Chamado do
prinmenu.cmd
Faça!
inicie o circuito
mostre o menu com opções de impressão
abra os arquivos de dados e indexações, ou crie indexações especiais para:
    obter condições especiais para imprimir e;
    trabalhe com relatórios prédefinidos
    e programados conjuntamente
    permita a saída de teste para a tela ou impressão e;
    imprima os relatórios

pode marcar campos booleanos específicos para justificar que
determinadas impressões ocorreram
pode sair automaticamente ao sistema operacional
pode fechar todos os arquivos de dados e voltar ao menu principal
Pare!
altere os arquivos mas não aqueles que marcam os Booleanos



Note que, neste desenho, os arquivos de dados são abertos quando uma tarefa específica se inicia e fechados quando está completa. Esta é uma prática significativa na programação dBase, mesmo que pareça ineficiente.

A razão principal para isto é que nunca há nenhuma garantia de onde o indicador do arquivo estará, na área primária ou secundária, após terminar uma determinada tarefa, mas o sucesso da próxima tarefa dependerá disto. Se usarmos constantes GOTO TOPs e parecidos, não obteremos muitas novidades. Se, por outro lado, soubermos quando escrevemos um módulo que ele estará perdido em um arquivo de dados, então saberemos o que precisamos fazer para efetivá-lo em todos os casos.

De fato, alguns arquivos de comando representam breves procedimentos que precisam saber um pouco das condições quando são chamados. Consideremos um arquivo de comando que diz:

```
@ 1, 55 SAY dele: rec
@ ... (*O resto do formato da tela*)
```

Em qualquer outro lugar no disco podemos ter o arquivo DELCHEK.CMD que contém:

```
IF *
  STORE "*** DELETED ***" TO DELE:REG
ELSE
  STORE " " TO DELE:REG
ENDIF
```

Por que nos incomodamos em escrever cinco linhas de codificação em um arquivo de comando sobre si próprio? A vantagem de um arquivo como DELCHEK é que ele pode ser chamado de qualquer rotina em qualquer lugar no programa.

Mas a maioria dos arquivos de comandos que não é menu possui alguns efeitos em arquivos de dados. Entradas de indexações mudarão, registros serão apagados e alterações importantes de dados serão feitas. É melhor não confundir a seqüência passando os arquivos de dados abertos. Se um arquivo de comandos abre um arquivo de dados, então ele seria o arquivo de comandos ideal para fechá-lo na maioria dos casos, embora ele possa chamar outros arquivos de comando para desenvolver operações nos dados.

OS MENUS E A PROGRAMAÇÃO ESTRUTURADA

Os menus do dBase foram *feitos* para a programação estruturada. Vamos trabalhar um pouco mais com os nossos velhos amigos SOCIOS e seu arquivo de dados relacionado, BICHOS. Mesmo sem usar qualquer pseudocódigo para procurarmos onde estamos, iniciaremos um programa de controle:

```
****************************************
* START.CMD  - para SOCIOS
****************************************
*
* ative a configuracao
SET TALK OFF
SET DEFAULT TO B
* sem rodeios
DO PRINMENU.CMD
* eof  start.cmd
```

A coisa mais importante para notarmos sobre START.CMD é que ele é linear – nosso programa entra no topo e sai na base, desta maneira START não conta contra nosso limite de 16 arquivos abertos. Continuando:

```
****************************************
*  PRINMENU.CMD  - para SOCIOS E BICHOS
****************************************
* loop sem fim
DO WHILE t
  * desenhe o menu principal
  ERASE
  ?
  ? [                         MENU PRINCIPAL]
  ? [                         --------------
  ?
  ? [         A = INCLUI REGISTROS]
  ?
```

```
  ? [         E = EDITA REGISTROS
  ?
  ?           Q = SAI P/SISTEMA OPERACIONAL
  ?
  * pegue a opcao
  WAIT TO opcao
  * faca alguma coisa com ela - um CASE,por exemplo
  DO CASE
     CASE !(opcao) = "A"
          * damos o nome agora e fazemos depois
          DO ADDMENU
     CASE !(opcao) = "E"
          * mesma coisa
          DO EDITMENU
     CASE !(opcao) = "Q"
          * isto podemos fazer aqui
          ERASE
          @ 11,8 SAY "CIAU BAMBINO"
          SET CONSOLE OFF
          QUIT
     * para os casos de opcao inconsistente
     OTHERWISE
          LOOP
  ENDCASE
     * porque esta fora do circuito, ele nunca vai tocar
     ? CHR(7)
ENDDO
*
*  eof prinmenu.cmd
```

Que ótimo — terminamos a parte principal de controle de nossa programação e ainda não nos detemos a nenhum dos detalhes de EDITMENU ou ADDMENU. Não abrimos sequer um arquivo de dados!

Já que está tudo caminhando tão bem, vamos fazer um ADDMENU:

```
*********************************************
* ADDMENU.CMD - mais socios
*********************************************
*
* chegamos aqui do meu principal
* vamos fazer outro circuito sem fim
DO WHILE t
   * e um menu interno
```

```
   ERASE
   ?
   ? [                   MENU DE INCLUSAO]
   ? [                   ----------------]
   ?
   ? [        opcoes de entradas  ]
   ?
   ? [        B = ENTRADAS EM BLOCO]
   ?
   ? [        S = ENTRADAS INDIVIDUAIS]
   ?
   ? [        R = RETORNA AO MENU PRINCIPAL]
   ?
   * obtenha a operacao desejada
   WAIT TO op
   DO CASE
      CASE !(op) = "B"
           DO BATCHADD
      CASE !(op) = "S"
           DO SINGADD
      CASE !(op) = "R"
           * este e facil
           RETURN
      * nao precisa otherwise
   ENDCASE
ENDDO
* eof addmenu
```

Agora arregaçamos nossas mangas a fim de escrever a codificação para SINGADD.CMD – mas esperem, já não possuímos uma biblioteca de programas para SOCIOS.DBF? É claro, escrevemos um no Capítulo 8 com o nome de ADDDONO.CMD. Então em vez de trocar o nome do arquivo de comando do Capítulo 8 (embora funcionasse) voltaremos à estrutura do ADDMENU e faremos uma troca:

```
CASE ! (op) = 'S'
     *singadd = adddono
     DO adddono
```

Terminado, ADDDONO precisa de uma edição – ele limpa tudo antes de voltar, mas também ativa talk on novamente, que não é apropriado para um módulo chamado dentro de um programa.

O restante do programa SOCIOS seria desenvolvido da mesma maneira. Com um diagrama de fluxos de dados, as estruturas de registros bem desenhadas, e uma programação estruturada usando as estruturas dos menus e pseudocódigos para os módulos que trabalham de verdade, o programador em dBase pode aproximar qualquer designação de programação de bancos de dados com confiança.

CAPÍTULO 10

A Depuração

Os programas não funcionam na primeira vez que os colocamos na máquina porque é de sua natureza possuir falhas, na lógica ou nas aplicações. Raramente a falha encontra-se na linguagem que estamos utilizando para escrever nossos programas.

Para fazer com que nossos programas funcionem temos de depurá-los. Esta operação é gratificante e necessita de uma personalidade especial para fazê-lo. Como outros aspectos da programação, a aplicação e prática em depurar torna-se mais fácil e até criativa. Devemos nos disciplinar para alcançarmos a habilidade.

O dBase possui comandos de depuração muito práticos e úteis que serão apresentados neste capítulo. Mostraremos também uma longa operação de depuração que demonstra o uso dos comandos. No final do capítulo, forneceremos algumas sugestões de como utilizar os comandos de depuração do dBase eficientemente e também uma listagem dos erros mais comuns e como aparecem.

SET ECHO ON

Ao ativarmos ECHO, ele repetirá o comando executado na tela. Dos comandos de depuração, SET ECHO ON pode ser o mais útil para consertos rápidos – diagnosticando o que está errado com pequenos arquivos de comandos sem muito trabalho. Observando os comandos do arquivo através da tela (eles podem ser interrompidos com um ^S e abortados com um <esc>) é sempre interessante acompanharmos a operação porque obteremos a depuração com sucesso.

Vamos demonstrar ECHO com SOCIOS e um pequeno arquivo de comandos com erros chamado DEMO.CMD:

```
DO WHILE .NOT. EOF
   DISPLAY nome,2*contrib
   SKIP
   DISPLAY cep,VAL(CEP)/2
   SKIP
ENDDO
```

Como podemos observar, DEMO é um tanto excêntrico. Ele mostra um sobrenome e duas vezes a contribuição de um registro e o CEP e a metade do CEP do próximo. Aqui está o arquivo que ele trabalhará:

```
. LIST OFF  (*sem número de registro, por favor *)
Mila     Moreira   R.Elma, 111          Suite 1    Sao Paulo  SP 92109  5.00 .T.
Monica   Medeiros  R. Diva, 2219                   Curitiba   PR 44110  5.99 .T.
Pedro    Penedo    R.Sao Luiz, 22       Apt. 232   Sao Paulo  SP 92109  0.00 .F.
Carlos   Covas     R. "A", 88                      Sao Paulo  SP 92101  0.25 .T.
Artur    Dunas     Rua das Flores, 123  Sala 17    Aracaju    SE 36222  1.00 .T.
Julio    Belo      Praia da Lua                    Recife     PE 92022  0.00 .F.
```

Para as duas operações de DEMO, o TALK será desativado para suprimir as tagarelices da tela. Aqui está uma operação de DEMO sem ECHO:

```
. DO DEMO
00001 Mila       10.00
00002 44110  22055
00003 Pedro       0.00
00004 92101  46050
00005 Artur       2.00
00006 92022  46011
```

Claramente, DEMO é um arquivo de comando que funciona. Apresentaremos uma operação com ECHO – note que o último comando é um "DO WHILE .NOT. EOF". O programa termina com ele porque quando o teste foi realizado pela última vez, EOF tornou-se verdadeiro:

```
. SET ECHO ON


. DO DEMO
DO DEMO                (* echo "DEMO")
DO WHILE .NOT. EOF
```

```
   DISPLAY NOME,2*CONTRIB
00001  Mila          10.00
   SKIP
   DISPLAY CEP,VAL(CEP)/2
00002  44110   22055
   SKIP
ENDDO
DO WHILE .NOT. EOF
   DISPLAY NOME,2*CONTRIB
00003  Pedro          0.00
   SKIP
   DISPLAY CEP,VAL(CEP)/2
00004  92101   46050
   SKIP
ENDDO
DO WHILE .NOT. EOF
   DISPLAY NOME,2*CONTRIB
00005  Artur          2.00
   SKIP
   DISPLAY CEP,VAL(CEP)/2
00006  92022   46011
   SKIP
ENDDO
DO WHILE .NOT. EOF      (*aqui EOF é verdadeiro e termina o DEMO *)
```

SET STEP ON

A característica STEP permite a movimentação em passos através de um arquivo de comandos, quando ativado. Quando executamos cada linha do arquivo de comandos o dBase volta o controle para o usuário no terminal com a linha:

```
SINGLE STEP Y:= STEP, N:= KEYBOARD CMD,ESC:= CANCEL
```

O usuário pode digitar um "Y", "N" ou "<esc>", com os seguintes efeitos: o "Y" liberará o dBase para executar o próximo passo. O "N" fornecerá o prompt para que um comando simples possa ser inserido no programa antes de voltar à execução do passo. E a tecla <esc>, como ela faz em outras situações, cancelará a execução e voltará ao modo interativo (embora STEP ainda esteja ativado).

Embora o comando STEP seja freqüentemente usado com ECHO ativado, ele pode ser usado sozinho. Quando estamos operando em um programa o ideal é ativar ECHO e TALK. Quando manipulamos um erro, por que esconder a informação?

Provavelmente a característica mais útil de STEP está em podermos inserir comandos. Por exemplo, podemos usar "N" para cancelar comandos do arquivo de comando, como SET TALK OFF, que poderia esconder o erro. Alternativamente, podemos inserir comandos que nos indicará o que acontece por trás das operações, como DISPLAY MEMORY se suspeitarmos alguma desorganização na memória, ou "? #" se acharmos que o indicador está no registro errado. Se acreditarmos que as indexações podem estar erradas ou que poderiamos estar na área primária quando desejamos estar na secundária, utilizamos a opção "N" para fazer um DISPLAY STATUS.

Para continuar a operação de depuração usamos STEP com ECHO.

SET DEBUG ON

O comando ECHO movimenta-se muito rápido para permitir a visão clara de quando STEP está desativado e ele atravanca a tela, como veremos na operação de depuração. Por essas razões o dBase nos proporciona uma última opção de depuração, SETDEBUG. Quando ativamos DEBUG, os resultados de ECHOs e STEPs serão enviados à impressora em vez da tela. Isto é tudo que o DEBUG faz. Ele ainda é útil, especialmente com o ECHO sozinho.

UMA OPERAÇÃO DE DEPURAÇÃO

Para demonstrarmos a depuração, necessitamos de algo para depurar. O programa TENTA.CMD servirá para o nosso teste. TENTA esperar encontrar SOCIOS na área primária com ou sem indexação, e os BICHOS indexados no sobrenome dos sócios que são donos através de DONOS.NDX. A proposta de TENTA é localizar os bichos corretos nos sócios que são donos e substituir o nome do bicho para o nome do sócio. O arquivo foi escrito com muitas falhas e erros propositadamente, mas somente cinco depurações foram necessárias para obter todas, e poucos teriam realizado esta operação de verdade.

Sabemos que essas operações às vezes são hipnotizantes, com os comandos e saídas que formam modelos geométricos em uma página que levam os leitores a dormir, eu peço desculpas. Porém, esta operação foi bem traçada a fim de contradizer a colocação. Seria mais interessante àqueles leitores que estão acompanhando com o computador ligado digitar a versão original de TENTA.CMD e seguir a depuração na tela.

Para ser mais breve, eu abreviei muitos dos comandos necessários para voltar ao ponto inicial da depuração. Se acompanharem no computador, lembre-se de que entre as operações devemos voltar à área primária e GOTO TOP.

TENTA – Versão 1

```
SET TALK OFF
DO WHILE .NOT. EOF
   DISPLAY sobrenome
   SELECT SECONDARY
   FIND p.sobrenome
   REPLACE p.nome WITH nome
ENDDO
```

Ativamos a configuração com:

```
. USE SOCIOS INDEX SOBR
. SELE SECO
. USE BICHOS INDEX DONOS
. SELE PRIM
```

Sem STEP ou ECHO, nos lançaremos nele:

```
. DO TENTA
00006 Belo  (*Aqui a tela cessa — nada mais acontece*)
```

Quando a tela trava, como aconteceu, devemos primeiro tentar a tecla <esc>. Neste caso funcionou — obtivemos o prompt — e então observaremos a configuração para verificarmos até onde caminhamos:

```
. DISP STAT

DATABASE SELECTED - B:BICHOS  .DBF
SECONDARY USE DATABASE

INDEXES:          KEY EXPRESSION:
 B:DONOS   .NDX   SOBRENOME


UNSELECTED DATABASE - B:SOCIOS  .DBF
PRIMARY USE DATABASE

INDEXES:          KEY EXPRESSION:
 B:SOBR    .NDX   SOBRENOME
```

Já que estamos na secundária, vamos parar. Vamos ver o que acontece em uma operação de depuração.

Após prepararmos novamente (SELECT PRIMARY, GOTO TOP), ativaremos STEP e ECHO. O restante da operação será feita por etapas com nossa resposta para "yes/no/escape" que aparece no final da linha de pergunta (como ele funciona na tela):

```
. DO TENTA
DO TENTA
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD.  ESC:=CANCELY
```

```
SET TALK OFF
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELN
. SET TALK ON
SET TALK ON
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
DO WHILE .NOT. EOF
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
   DISPLAY SOBRENOME
00006  Belo
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
   SELE SECO
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
   FIND P.SOBRENOME
NO FIND
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCEL <esc>
```

Ah!, um NO-FIND! Uma olhada na Versão 1 de TENTA mostra que dissemos "FIND p.sobrenome". Talvez precisemos armazenar o sobrenome do dono na memória. Chamamos rapidamente TENTA em MODIFY COMMAND (não há necessidade de ativar o processador de texto para isto), e prosseguimos:

TENTA – Versão 2

```
SET TALK OFF
DO WHILE .NOT. EOF
   DISPLAY sobrenome
   STORE sobrenome TO Msobrenome
   SELECT SECONDARY
   FIND Msobrenome              (*Aqui a modificação *)
   REPLACE p.nome WITH nome
ENDDO
```

Voltaremos para a outra operação convencidos de que agora funcionará:

```
. DO TENTA
DO TENTA
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
SET TALK OFF
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELN
. SET TALK ON
SET TALK ON
```

```
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
DO WHILE .NOT. EOF
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
   DISPLAY sobrenome
00006  Belo
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
   STORE sobrenome TO Msobrenome
Belo
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
   SELE SECONDARY
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
   FIND Msobrenome
NO FIND
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCEL
```

Nosso tempo não foi desperdiçado porque aprendemos algo. Find espera uma string e não um nome de uma variável. Vamos tentar uma macro:

TENTA – Versão 3

```
SET TALK OFF
DO WHILE .NOT. EOF
   DISPLAY sobrenome
   STORE sobrenome TO Msobrenome
   SELECT SECONDARY
   FIND &Msobrenome              (*Uma macro para a atualização de string *)
    REPLACE p.nome WITH nome
ENDDO
```

E agora a próxima operação:

```
. DO TENTA
DO TENTA
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
SET TALK OFF
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELN
. SET TALK ON
SET TALK ON
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
DO WHILE .NOT. EOF
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
```

```
   DISPLAY sobrenome
00006  Belo
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
   STORE sobrenome TO Msobrenome
Belo
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
   SELE SECONDARY
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
   FIND Belo.
NO FIND
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELN
. LIST
LIST
00005  BELO       foca     Ester
00006  BELO       galinha  Poka
00007  BELO       peru     Leo
00004  MEDEIROS   cachorro Bola
00001  PENEDO     cachorro Fido
00003  SIMOES     pombo    Apollo
00002  ZORZI      gato     Arthur
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCEL (esc)
```

Quando obtivemos um NO FIND, nossa dúvida foi tão grande que interrompemos a etapa para inserir um comando LIST. Como sempre, o dBase estava certo, BICHOS está indexado no campo SOBRENOME, mas os conteúdos daqueles campos estavam em maiúsculas.

A nossa próxima versão de TENTA deve refletir isto colocando em maiúscula o sobrenome do arquivo SOCIOS quando armazenado na memória. Inserimos a função de maiúsculas em FIND:

TENTA – Versão 4

```
SET TALK OFF
DO WHILE .NOT. EOF
   DISPLAY sobrenome
   STORE !(sobrenome) TO Msobrenome          (*Aqui a modificação *)
   SELECT SECONDARY
   FIND &Msobrenome
   REPLACE p.nome WITH nome
ENDDO
```

Direto para a nova operação:

```
. DO TENTA
DO TENTA
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
SET TALK OFF
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELN
. SET TALK ON
SET TALK ON
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
DO WHILE .NOT. EOF
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
   DISPLAY sobrenome
00006  Belo
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
   STORE !(sobrenome) TO Msobrenome
BELO
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
   SELE SECONDARY
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
   FIND BELO
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELN
. ? #
? #
     5
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELN
. DISP
DISP
00005  BELO             foca     Ester
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
   REPLACE p.nome WITH nome
00001 REPLACEMENT(S)
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELN
. ? p.nome
? p.nome
Ester
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
ENDDO
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
DO WHILE .NOT. EOF
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
   DISPLAY sobrenome
00005  BELO
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
```

```
   STORE !(sobrenome) TO Msobrenome
BELO
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELN
. SET ECHO OFF
SET ECHO OFF
SINGLE STEP  Y:=STEP, 'N:=KEYBOARD CMD,  ESC:=CANCELN
. SET STEP OFF
00001 REPLACEMENT(S)
00005  BELO
BELO
00001 REPLACEMENT(S)     (*Com STEP e ECHO desativados, a operação continua e TALK mostra que estamos
00005  BELO              parados em BELO — precisamos de um SKIP*)
BELO
00001 REPLACEMENT(S)
00005  BELO
BELO
. LIST STATUS
DATABASE SELECTED - B:BICHOS  .DBF
SECONDARY USE DATABASE

INDEXES:          KEY EXPRESSION:
 B:DONOS   .NDX   sobrenome

UNSELECTED DATABASE - B:SOCIOS  .DBF
PRIMARY USE DATABASE

INDEXES:          KEY EXPRESSION:
 B:SOBR    .NDX   SOBRENOME
```

A próxima operação é mais conhecida do que as outras, porque prosseguiremos e testaremos vários aspectos diferentes de TENTA.CMD. Observamos claramente que nosso arquivo de comando necessita que usemos um SKIP depois de sua ação, porque a operação permanece no mesmo registro. Observamos também que esquecemos de voltar à área primária.

Mas um problema permaneceu. A etapa com REPLACE, que o dBase nos disse que havia continuado sem problemas, servia para escrever os nomes dos bichos, "Ester" dentro do campo que contém o nome do dono, "Júlio".

Durante a etapa de depuração, obtivemos "00001 REPLACEMENT(S)" e quem somos nós para discordar? Com a checagem rápida durante a etapa "? p. nome" obtivemos a mensagem "Ester". O dBase está nos comunicando que as substituições foram feitas.

Lembre-se de que as regras do uso das áreas primárias e secundárias dizem que podemos escrever *somente* na área selecionada. Como podemos ter um comando correto escrito em uma área que não foi selecionada? Para checarmos, voltaremos à primária e listaremos os registros em SOCIOS:

```
. SELECT PRIMARY
. LIST
00006  Julio    Belo      Praia da Lua                       Recife     PE 92022  0.00 .F.
00004  Carlos   Covas     R. "A", 88                         Sao Paulo  SP 92101  0.25 .T.
00005  Artur    Dunas     Rua das Flores, 123  Sala 17       Aracaju    SE 36222  1.00 .T.
00002  Monica   Medeiros  R. Diva, 2219                      Curitiba   PR 44110  5.99 .T.
00001  Mila     Moreira   R.Elma, 111          Suite 1       Sao Paulo  SP 92109  5.00 .T.
00003  Pedro    Penedo    R.Sao Luiz, 22       Apt. 232      Sao Paulo  SP 92109  0.00 .F.
```

O que suspeitávamos foi confirmado, estamos prontos para escrever outra versão de TENTA com a movimentação apropriada entre as áreas primárias e secundárias e com um SKIP:

TENTA – Versão 5

```
SET TALK OFF
DO WHILE .NOT. EOF
   DISPLAY sobrenome
   STORE !(sobrenome) TO Msobrenome
   SELECT SECONDARY
   FIND &Msobrenome
   SELECT PRIMARY
   REPLACE nome WITH s.nome
   SKIP
ENDDO
```

```
. DO TENTA
DO TENTA
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
SET TALK OFF
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELN
. SET TALK ON
SET TALK ON
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
DO WHILE .NOT. EOF
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
   DISPLAY sobrenome
00006  Belo
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
   STORE !(sobrenome) TO Msobrenome
BELO
```

```
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
   SELE SECONDARY
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
   FIND BELO
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELN
. DISP
DISP
00005  BELO           foca     Ester
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
   SELECT PRIMARY
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
   REPLACE nome with s.nome
00001 REPLACEMENT(S)
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELN
. DISP
DISP
00006  Ester     Belo           Praia da Lua          Recife     PE 92022   0.00 .F.
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
   SKIP
RECORD: 00004
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELN
. DISP
DISP
00004  Carlos    Covas          R. "A", 88            Sao Paulo  SP 92101   0.25 .T.
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
ENDDO
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
DO WHILE .NOT. EOF
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
   DISPLAY sobrenome
00004  Covas
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
   STORE !(sobrenome) TO Msobrenome
COVAS
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
   SELE SECONDARY
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
   FIND COVAS
NO FIND
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
   SELECT PRIMARY
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
   REPLACE nome with s.nome
00001 REPLACEMENT(S)
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELN
```

```
. DISP
DISP
00004  Ester      Covas           R. "A", 88            Sao Paulo  SP 92101   0.25 .T.
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELN
. SELECT SECONDARY
SELECT SECONDARY
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELN


. LIST
LIST
00005  BELO             foca     Ester
00006  BELO             galinha  Poka
00007  BELO             peru     Leo
00004  MEDEIROS         cachorro Bola
00001  PENEDO           cachorro Fido
00003  SIMOES           pombo    Apollo
00002  ZORZI            gato     Arthur
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCEL <esc>
.
```

Uma listagem rápida do arquivo SOCIOS revelará a última falha em nossa lógica. Já que não fizemos previsões em TENTA para os registros em SOCIOS que não estão unidos com nenhum dos registros em BICHOS, alguns registros que não estão unidos terão uma substituição errada com um valor falso:

```
. LIST
00006  Ester   Belo      Praia da Lua                      Recife     PE 92022   0.00 .F.
00004  Ester   Covas     R. "A", 88                        Sao Paulo  SP 92101   0.25 .T.
00005  Artur   Dunas     Rua das Flores, 123  Sala 17      Aracaju    SE 36222   1.00 .T.
00002  Monica  Medeiros  R. Diva, 2219                     Curitiba   PR 44110   5.99 .T.
00001  Mila    Moreira   R.Elma, 111          Suite 1      Sao Paulo  SP 92109   5.00 .T.
00003  Pedro   Penedo    R.Sao Luiz, 22       Apt. 232     Sao Paulo  SP 92109   0.00 .F.
```

Para mantermos a integridade de nosso teste, editamos os registros de Belo e Covas de volta a seus estados originais. Agora a depuração tornou-se tediosa; porém, necessitamos de mais uma modificação em TENTA:

TENTA – Versão Final

```
SET TALK OFF
DO WHILE .NOT. EOF
   DISPLAY sobrenome
   STORE !(sobrenome) TO Msobrenome
   SELECT SECONDARY
   FIND &Msobrenome
   IF # <> 0
       SELECT PRIMARY
       REPLACE nome WITH s.nome
   ELSE
       SELECT PRIMARY
   ENDIF
   SKIP
ENDDO
```

Só falta testar o arquivo de comando depurado. Deixaremos ele operar somente através dos casos Belo e Covas – não precisamos de mais exemplos:

```
. DO TENTA
DO TENTA
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
SET TALK OFF
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELN
. SET TALK ON
SET TALK ON
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
DO WHILE .NOT. EOF
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
   DISPLAY sobrenome
00006  Belo
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
   STORE !(sobrenome) TO Msobrenome
BELO
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
   SELE SECONDARY
```

```
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
   FIND BELO
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
   IF # <> 0
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
      SELECT PRIMARY
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
      REPLACE nome WITH s.nome
00001 REPLACEMENT(S)
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELN
. DISP
DISP
00006  Ester      Belo             Praia da Lua             Recife     PE 92022   0.00 .F.
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
   ELSE
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
   SKIP
RECORD: 00004
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
ENDDO
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
DO WHILE .NOT. EOF
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
   DISPLAY sobrenome
00004  Covas
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
   STORE !(sobrenome) TO Msobrenome
COVAS
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
   SELE SECONDARY
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
   FIND COVAS
NO FIND
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
   IF # <> 0
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
      SELECT PRIMARY
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
   ENDIF
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
```

```
   SKIP
RECORD: 00005
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
ENDDO
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCELY
DO WHILE .NOT. EOF
SINGLE STEP  Y:=STEP,  N:=KEYBOARD CMD,  ESC:=CANCEL
```

(*e aqui nós a abandonamos *)

Note que quando o dBase controla uma estrutura como um IF-ELSE-ENDIF durante o processo por etapas, ele acessa o valor verdadeiro do booleano e apenas "passa a etapa" da seção de codificação que não foi executada. Ele também faz o mesmo com as linhas em branco e as linhas de comentário, que às vezes dá a impressão de esquecimento do depurador. Não perca a fé – quando continuar em outra etapa, pressione Y e nada acontecerá, isto geralmente mostra que o dBase possui uma linha para interpretar o que o interpretador foi programado para não considerar.

Vamos ver os resultados de TENTA.CMD funcionando:

```
. LIST
00006 Ester    Belo      Praia da Lua                       Recife    PE 92022  0.00 .F.
00004 Covas    Carlos    R. "A", 88                         Sao Paulo SP 92101  0.25 .T.
00005 Artur    Dunas     Rua das Flores, 123  Sala 17       Aracaju   SE 36222  1.00 .T.
00002 Bola     Medeiros  R. Diva, 2219                      Curitiba  PR 44110  5.99 .T.
00001 Mila     Moreira   R.Elma, 111          Suite 1       Sao Paulo SP 92109  5.00 .T.
00003 Fido     Penedo    R.Sao Luiz, 22       Apt. 232      Sao Paulo SP 92109  0.00 .F.
.
. SELE SECO
. LIST
00005 BELO          foca     Ester
00006 BELO          galinha  Poka
00007 BELO          peru     Leo
00004 MEDEIROS      cachorro Bola
00001 PENEDO        cachorro Fido
00003 SIMOES        pombo    Apollo
00002 ZORZI         gato     Arthur
```

Viram, funcionou! De fato ele não serve para nada, mas o processo de depuração foi demonstrado amplamente.

OS PROCEDIMENTOS CORRETOS PARA A DEPURAÇÃO

Quando depuramos os programas, é impraticável (e extremamente maçante) passar por centenas e até milhares de passos em arquivos de comando encaixados. Além disso, alguns erros são fáceis de detectar e consertar.

A depuração é parecida com qualquer outro processo, não faz sentido nos utilizarmos de um instrumento potente para simplesmente perseguir os pequenos problemas. O que necessitamos são procedimentos corretos para a depuração a fim de atingirmos respostas compatíveis aos programas que não funcionam. No caso de não conseguirmos eliminar o erro, o programador pode encontrar um outro instrumento apropriado para a depuração.

Apresentaremos aqui alguns procedimentos ordenados para seguirmos ao depurar um programa dBase:

1. Leia o programa. Os programas são, acima de tudo, arquivos de texto. Liste os arquivos de comando na impressora e *leia-os* – devagar e várias vezes, se necessário. Reflita. Pergunte-se se cada etapa funcionará.
2. Opere os módulos sozinhos, usando ou não STEP. Para esta etapa, ative a configuração e a memória manualmente e chame o arquivo de comandos pelo prompt. Quando o módulo terminar, visualize o status; cheque a localização do indicador do registro nas áreas primária e secundária; cheque a memória; liste os registros. Obteve os resultados desejados?
3. "Comente" as linhas de codificação suspeitas; isto é, marque as linhas que deseja ativar nas laterais com asteriscos, transformando-as em linhas de comentário. Teste os programas novamente e se for possível, use os procedimentos acima.
4. "Use escape" nos WAITs e examine a configuração. Isto é, quando obtivermos a mensagem WAITING, pressione <esc>, que cancelará o arquivo de comando. Visualize o status, a memória e a localização dos indicadores de registro.
5. Opere o programa apenas ativando ECHO – não use STEP ainda. Se não puder ver o erro enquanto os comandos continuam na tela, ative DEBUG e ECHO de forma que os comandos repetidos sejam enviados à impressora. Teste isto com o comando SET TALK OFF comentado.
6. Agora vamos falar sério. Ainda não funciona. Vamos agora acompanhar os passos, porém isso não significa que teremos de acompanhar o programa todo. Usaremos o editor para colocar as linhas "SET ECHO ON" e "SET STEP ON" no ponto em que ocorreu a quebra, assim poderemos acompanhar o processo deste ponto em diante. Geralmente o problema está no topo de um arquivo de comandos determinado. Quando o programa estiver funcionando, tire as linhas STEP e ECHO.

SEIS "FUROS" COMUNS (E COMO CONSERTÁ-LOS)

Apresentaremos seis tipos de erros e "furos" comuns, difíceis para o diagnóstico dos iniciantes, que merecem uma atenção especial:

1. NO FINDs estúpidos ou obviamente falsos.

   – Se obtivermos um NO FIND e *sabemos* que há um registro:
   – Cheque para verificar se está na área errada (primária ou secundária) para o FIND.
   – Cheque para verificar se está com a indexação errada para o FIND. Está usando SOCIDADE.NDX, enquanto o FIND está procurando SOBRENOME?
   – Certifique-se de que a string FIND foi corretamente convertida para comparar a expressão de indexação. Se estiver tentando encontrar "Sonia" e a indexação estiver em !(sobrenome), o FIND nunca funcionará.

   Lembre-se de que, NO FIND significa que o dBase não pode encontrar o que foi solicitado. Se souber que o registro desejado está no arquivo de dados, enviamos ao dBase um FIND descrito incorretamente.

2. Um erro "RECORD OUT OF RANGE" em um FIND, LIST ou qualquer outro comando de movimentação.

   Isto quase *sempre* quer dizer que a indexação em uso não corresponde ao arquivo de dados. Use REINDEX (na Versão 2.4) ou INDEX novamente. Se isto acontecer no programa, a codificação está usando um APPEND ou trocando parte da chave de indexação quando este não estiver aberto.

3. Um erro "FILE IS CURRENTLY OPEN" em um RENAME, DELETE FILE ou MODIFY COMMAND.

   Geralmente quando obtemos esta mensagem, é óbvio o que está acontecendo. O arquivo pode ser fechado, bem como o RENAME e DELETE e tudo mais que foi desenvolvido. Porém, em dois casos, esta mensagem pode parecer muito confusa. Primeiro, podemos ter esquecido quando um formato de arquivo ".FMT" foi aberto, especialmente quando a operação que causou o erro não parece estar relacionada – um CLEAR ou um RESET pode ser utilizado. Se nos confundirmos com um erro "CURRENTLY OPEN", devemos usar DISPLAY STATUS para verificar se o formato de arquivo foi ativado, neste caso, limpe-o com "SET FORMAT TO". O segundo caso envolve um comando de fechamento ou DELETE FILE em um arquivo de dados. Quando CURRENTLY OPEN for anunciado, digitamos USE para fechá-lo e tentamos novamente – a fim de obtermos a mesma mensagem de erro. Invariavelmente isto significa que o arquivo aberto está em outra área e que a mensagem de erro pode ser precipitada por algo tão simples como tentar abrir SOCIOS na secundária quando já estiver em uso na primária.

4. Um programa funciona no começo e depois pára. Geralmente com um ou dois FINDs obtemos sucesso e então começa um NO FIND. Ou depois de uma operação cíclica que funcionou corretamente, ao retornarmos, os SYNTAX ERROs surgem na tela ou os formatos de tela são absurdos.

   Quando isto acontece, quase sempre significa que LOOP ou RETURN, os comandos GOTO secretos do dBase, nos apunhalou pelas costas. Consideremos esta estrutura:

```
*circuito infindável para fora
DO WHILE t
     *algum processo complicado
     a
     .
     .
     SELECT SECONDARY
     .
     b
     c
     FIND &alguma coisa
     IF # = 0
        LOOP
     ELSE
        *o procedimento importante de verdade
        DO rlymp
     ENDIF
     .
     y
     z
ENDDO
```

Este arquivo de comandos sempre funcionará, se não tivermos um NO FIND. Porém, a primeira vez que a estrutura IF detectar um NO FIND, ela devolverá o controle ao DO WHILE e ninguém nos tirará da área secundária. RETURNs podem fazer a mesma coisa.

Observe seus LOOPs e RETURNs! Se os usarmos deveremos colocá-los no alto do controle da estrutura ou do arquivo de comandos, para não parecer que trocamos as coisas. Antes de usarmos estes comandos, deveremos ativar tudo – indexações, área selecionada, memória – como quando o controle da estrutura foi chamado. Do contrário, ficaremos loucos!

5. Uma linha perfeitamente correta de codificação fornece um SYNTAX ERROR várias vezes. Geralmente isto significa que o nosso processador de texto ou o editor de nosso programa deixou um caractere de controle escondido no arquivo de comando. A linha de codificação alterou um caractere de maneira que não mostra na tela (foi ativado o bit mais alto em um byte). A primeira coisa a fazer é lembrarmos de usar um modo "não documento" ou de "programação" do processador de texto todas as vezes que alterarmos um arquivo de comando. Isto prevenirá o problema. Mas se estamos com o problema, ou:

– Eliminamos a linha com um apagamento completo ou digitamos em cima desta linha no modo de sobredigitação. Depois digitamos novamente a linha no modo não documento.

– Usamos um programa de processamento de texto apropriado para zerar os bits mais altos no arquivo. Em CP/M isto é feito com PIP.COM, usando a opção "Z":

A > PIP <nome nome> = <nome antigo> [Z]

6. O dBase some, a tela fica branca e ninguém está ouvindo quando digitamos. Aqui estão as três causas mais importantes deste erro frustrante, na ordem em que elas aparecem:
   – Um NO FIND com um TALK desativado e nenhum teste para ele. O dBase não tem para onde ir e não pode informá-lo sobre isto. Saia com um <esc> e teste o NO FIND com "? #".
   – Um erro de sintaxe depois de um SET CONSOLE OFF. Novamente saia com <esc> e (porque a tela ainda estará em branco) digite SET CONSOLE ON. Lembre-se de usar "escape" primeiro, porque quando tentarmos digitar o comando CONSOLE ON, atrás da tela escurecida, o dBase estará tentando comunicar o diálogo de correção de comando conosco. Um <esc> cessa todo este processo.
   – Um DO WHILE, DO CASE ou IF que não está fechado. Isto pode nos confundir durante dias, mas é um conserto rápido se soubermos como fazê-lo. Vá ao arquivo de comandos com o processador de texto e coloque uma pesquisa em "END DO" ou "END CASE". Elas devem ser escritas como uma palavra a fim de fecharmos o controle das estruturas. Em qualquer caso em que a tela ficar branca e que não apareça nenhum prompt, tente primeiro um <esc> e se não funcionar, tente SET CONSOLE ON ou ENDCASE ou ENDDO, dependendo da codificação no arquivo de comando.

E tire proveito de sua depuração. Afinal, esta é uma parte da programação e deve ser feita.

CAPÍTULO

11

Comandos e Técnicas Avançadas

Nos dez últimos capítulos apresentamos os comandos dBase que seriam suficientes para escrever qualquer programa e manipular os bancos de dados no modo interativo. Entretanto, o dBase possui outros comandos úteis que aumentam sua potência e flexibilidade.

Neste capítulo, examinaremos alguns desses comandos e técnicas. Em especial os seguintes:

SET ESCAPE OFF	INSERT
INPUT	JOIN
CHANGE	CREATE FROM
BROWSE	&&

SET ESCAPE OFF

Até agora, utilizamos a tecla escape ("< esc >", como a chamamos) como uma maneira de cancelar os comandos dBase de operação longa – LISTA, arquivos de comandos em uma mensagem WAITING, tudo que pode precisar ser interrompido.

Sem < esc >, os circuitos infindáveis ocasionais que aparecem em nossos programas travariam o computador de forma que nada além de um reset recuperaria a operação. Porém, quando terminamos e depuramos os programas e os entregamos ao cliente, a função da tecla < esc > torna-se uma ameaça à integridade do programa.

Por exemplo, um empregado, contratado na época de Natal com muito movimento, ao operar o computador, traz o prompt apenas pelo fato de ter tocado acidentalmente na tecla escape.

Como já imaginávamos pelo título desta seção, a característica <esc> pode ser desligada (e ligada) usando um SET ESCAPE OFF (ON). Este é um comando absoluto – estar ligado ou não – e geralmente é o último processo antes da distribuição do programa. Ele pertence ao módulo de entrada do programa ou a iniciação do arquivo de comandos.

A razão para o incluirmos por último é que a depuração está próxima do impossível quando ESCAPE está desativado. De fato, pode ser embaraçoso realizar a depuração adequada uma vez que SET ESCAPE OFF foi incluso ao programa. Se estivermos trabalhando no programa com a assistência do cliente, comente a linha SET ESCAPE ou passe pela etapa daquela parte do programa (está no alto, certo?) e ative escape novamente antes de liberar a operação do programa.

INPUT

O comando INPUT é um paralelo exato ao comando ACCEPT em seu uso e estrutura, exceto que INPUT aceitará o dado pelo tipo de dado somente. Isto é, os valores numéricos e lógicos são digitados como são, e o dado de caractere deve ser delimitado, como em [Esta é uma string delimitada por colchete].

ACCEPT, como lembraremos, armazena tudo que entrou como dado de caractere.

```
. INPUT "Informe um numero " TO Mnum
. Informe um numero: errado  (*string sem delimitador *)
 SYNTAX ERROR - RE-ENTER  (*dbase recusa *)

 :3.1417  (*segunda tentativa *)
 3.1417
? Mnum   (*Vamos ver o que está guardado *)
  3.1417
```

Há razões muito pequenas para usarmos INPUT, já que os valores numéricos podem entrar usando ACCEPT se forem recuperadas usando macros. Os valores lógicos que digitaremos serão provavelmente melhores com um WAIT TO, embora relacionem-se com o estilo de programação. INPUT está lá se desejar usá-lo.

CHANGE

O comando CHANGE é mais útil no modo interativo. Ele fornece uma função de edição de registro por registro, onde um arquivo de dados ou uma parte dele pode ser folheado enquanto o operador realiza as alterações.

A edição é feita através da substituição de string em vez de usarmos a edição de tela-cheia, de maneira a assemelhar-se com o diálogo de comando de correção. A característica mais

importante do comando é que com CHANGE um ou mais campos específicos devem ser especificados e eles estão isolados do restante dos campos em uma estrutura de arquivo. Sua sintaxe é:

```
CHANGE [ <escopo>] FIELD <list> [FOR <expressão>]
```

e vamos tentar uma operação curta com SOCIOS:

```
CHANGE NEXT 3 FIELD nome,sobrenome

RECORD: 00001

NOME: Mila
CHANGE?                         (*Digite <cr> para indicar "Não troque" *)

SOBRENOME: Moreira
CHANGE? M
TO       X

SOBRENOME: Xoreira
CHANGE?

RECORD: 00002

NOME: Bola
CHANGE?

SOBRENOME: Medeiros
CHANGE? e                       (*Ele selecionará o primeiro "e" *)
TO       o

SOBRENOME: Modeiros
CHANGE? ei
TO       a

SOBRENOME: Modaros
CHANGE?

RECORD: 00003

NOME: Fido
CHANGE? F
TO       M
```

```
NOME: Mido
CHANGE?

SOBRENOME: Penedo
CHANGE? P
TO        J

SOBRENOME: Jenedo
CHANGE? p
TO        j
NO FIND                     (* Lógico, o dBase não é tolo! *)

SOBRENOME: Jenedo
CHANGE?                     (*Um <cr> sai do CHANGE*)
```

BROWSE

Dos comandos de edição direta para o uso de modo interativo, BROWSE talvez seja o mais poderoso. Sua ação é acumulada ao descarregamento do disco e à edição – visualizamos o arquivo de dados inteiro (ou campos específicos) como uma forma de grade e o cursor pode ser movimentado entre os registros trocando todas as coisas.

BROWSE é muito rápido. É como classificar pérolas com um martelo – se trabalharmos muito rapidamente, podemos misturar tudo sem perceber. Eu sugiro primeiro testar BROWSE em um TEMP.DBF. Sua sintaxe é:

```
BROWSE [FIELDS <listagem dos campos>]
```

A tela de BROWSE para SOCIOS se parece com:

```
. BROWSE   RECORD # :00001
           NOME------ SOBRENOME------ ENDERECO------------ COMPL----- CIDADE---- ES CEP--
           Mila       Moreira         R.Elma, 111          Suite 1    Sao Paulo  SP 92109
           Monica     Medeiros        R. Diva, 2219                   Curitiba   PR 44110
           Pedro      Penedo          R.Sao Luiz, 22       Apt. 232   Sao Paulo  SP 92109
           Carlos     Covas           R. "A", 88                      Sao Paulo  SP 92101
           Davi       Dunas           Rua das Flores, 123  Sala 17    Aracaju    SE 36222
           Julio      Belo            Praia da Lua                    Recife     PE 92022
```

BROWSE é raramente usado dentro de um programa, mas tem valor no modo interativo.

INSERT

Às vezes o programador quer selecionar um arquivo para uma determinada ordem dos registros, embora o dBase possua um comando SORT e as funções de indexação que podem impor uma ordem aparente ao arquivo de dados. Para isto existe o comando INSERT que permite acrescentar um registro entre outros no arquivo de dados. Se, por exemplo, em um arquivo grande onde a ordem dos registros é muito importante, fizer uma nova classificação usando SORT depois de incluir um ou dois registros seria uma perda de tempo. A sintaxe de INSERT é:

```
INSERT [BLANK] [BEFORE]
```

Por default, o INSERT coloca o registro *após* o registro indicado. O argumento opcional BEFORE faz o contrário. Quando fornecemos o comando, como INSERT BLANK, ele age da mesma maneira que o APPEND BLANK, assim como INSERT sem argumento fornece uma tela igual ao APPEND. INSERT também se adaptará ao formato usado. Vamos realizar uma operação curta demonstrando INSERT:

```
. LIST
00001  Mila    Moreira   R.Elma, 111           Suite 1    Sao Paulo  SP 92109   5.00 .T.
00002  Monica  Medeiros  R. Diva, 2219                    Curitiba   PR 44110   5.99 .T.
00003  Pedro   Penedo    R.Sao Luiz, 22        Apt. 232   Sao Paulo  SP 92109   0.00 .F.
00004  Carlos  Covas     R. "A", 88                       Sao Paulo  SP 92101   0.25 .T.
00005  Davi    Dunas     Rua das Flores, 123   Sala 17    Aracaju    SE 36222   1.00 .T.
00006  Julio   Belo      Praia da Lua                     Recife     PE 92022   0.00 .F.

. GOTO 3
. DISPLAY
00003  Pedro   Penedo    R.Sao Luiz, 22        Apt. 232   Sao Paulo  SP 92109   0.00 .F.
```

INSERT (*Inserindo um novo registro 4*)

```
RECORD # 00004

NOME       :Edi      :
SOBRENOME  :Evirto       :
ENDERECO   :R. Lima, 22          :
COMPL      :              :
CIDADE     :Maceio     :
ESTADO     :AL:
CEP        :     :
CONTRIB    : 21.00:
PAGAMENTO  :y:
```

```
. LIST
00001  Mila     Moreira   R.Elma, 111           Suite 1    Sao Paulo  SP 92109   5.00 .T.
00002  Monica   Medeiros  R. Diva, 2219                    Curitiba   PR 44110   5.99 .T.
00003  Pedro    Penedo    R.Sao Luiz, 22        Apt. 232   Sao Paulo  SP 92109   0.00 .F.
00004  Edi      Evirto    R.Lima, 22                       Maceio     AL        21.00 .T.
00005  Carlos   Covas     R. "A", 88                       Sao Paulo  SP 92101   0.25 .T.
00006  Davi     Dunas     Rua das Flores, 123  Sala 17    Aracaju    SE 36222   1.00 .T.
00007  Julio    Belo      Praia da Lua                     Recife     PE 92022   0.00 .F.
```

Neste caso, o registro "E" foi inserido na cópia classificada de SOCIOS em seu lugar apropriado. Agora vamos demonstrar INSERT BLANK:

```
. GOTO 3
. DISPLAY
00003  Pedro    Penedo  R.Sao Luiz, 22      Apt. 232  Sao Paulo  SP 92109   0.00 .F.

. INSERT BLANK      (*Um novo registro 4 que também está em branco*)
. LIST
00001  Mila     Moreira   R.Elma, 111           Suite 1    Sao Paulo  SP 92109   5.00 .T.
00002  Monica   Medeiros  R. Diva, 2219                    Curitiba   PR 44110   5.99 .T.
00003  Pedro    Penedo    R.Sao Luiz, 22        Apt. 232   Sao Paulo  SP 92109   0.00 .F.
00004                                                                            0.00 .F.
00005  Edi      Evirto    R.Lima, 22                       Maceio     AL        21.00 .T.
00006  Carlos   Covas     R. "A", 88                       Sao Paulo  SP 92101   0.25 .T.
00007  Davi     Dunas     Rua das Flores, 123  Sala 17    Aracaju    SE 36222   1.00 .T.
00008  Julio    Belo      Praia da Lua                     Recife     PE 92022   0.00 .F.

. GOTO 7
. INSERT BLANK BEFORE
. LIST
00001  Mila     Moreira   R.Elma, 111           Suite 1    Sao Paulo  SP 92109   5.00 .T.
00002  Monica   Medeiros  R. Diva, 2219                    Curitiba   PR 44110   5.99 .T.
00003  Pedro    Penedo    R.Sao Luiz, 22        Apt. 232   Sao Paulo  SP 92109   0.00 .F.
00004                                                                            0.00 .F.
00005  Edi      Evirto    R.Lima, 22                       Maceio     AL        21.00 .T.
00006  Carlos   Covas     R. "A", 88                       Sao Paulo  SP 92101   0.25 .T.
00007                                                                            0.00 .F.
00008  Davi     Dunas     Rua das Flores, 123  Sala 17    Aracaju    SE 36222   1.00 .T.
00009  Julio    Belo      Praia da Lua                     Recife     PE 92022   0.00 .F.
```

Quando estamos usando INSERT para mantermos o arquivo na ordem de classificação, o programador é responsável em escrever a codificação que colocará o indicador no registro apropriado antes do INSERT. O comando LOCATE é próprio para isto em arquivos pequenos e médios.

```
LOCATE FOR (<nova chave>) < (<chave de registros existentes>)
INSERT BEFORE
```

Em arquivos de dados grandes, uma indexação tem de ser mantida no campo-chave, mesmo que o arquivo seja classificado, a fim de obtermos o lugar de início apropriado rapidamente:

```
STORE $(<nova chave>1,1,) TO finder
FIND&finder
DO WHILE (<nova chave>) (<chave de registros existentes>)
    SKIP
ENDDO
INSERT BEFORE
```

JOIN

Quando dois arquivos de dados em um banco de dados estão em uma relação de um-para-muitos, sempre é desejável gerar uma saída deles como se fossem somente um arquivo de dados.

Isto pode ser feito através de um programa ou usando JOIN e REPORT FORM. O efeito de JOIN é precisamente o mesmo do que faríamos para escrever o programa de relatório de saída. Primeiro, sua sintaxe:

```
JOIN TO <nome do arquivo de saída> FOR <exp> [FIELDS <lista de campo>]
```

Para usarmos JOIN, o programador deve utilizar o arquivo de único registro na primária e de registros múltiplos na secundária antes de chamar o comando JOIN:

Primária	**Secundária**
Registro "A"	"A" registro 1
	"A" registro 2
	"A" registro 3
Registro "B"	"B" registro 1
	"B" registro 2
	"B" registro 3

Com isto feito, JOIN escreverá um novo banco de dados com um registro para cada "comparação" combinada que se encontra com a condição especificada na cláusula FOR. A listagem dos campos permite que o programador controle o tamanho do arquivo de saída e seus conteúdos.

O JOIN pode ficar em dificuldades com as limitações do dBase em duas áreas. A primeira, se os registros combinados possuírem mais de 32 campos, o dBase usará todos os campos no arquivo primário e quantos campos couberem no arquivo secundário.

A segunda, se a condição na cláusula FOR não for convincente o bastante, o dBase pode terminar escrevendo ou tentando escrever mais de 65.535, o máximo de registros, a menos que o espaço do disco estoure antes.

Vamos unir SOCIOS com BICHOS.

```
. LIST STATUS
DATABASE SELECTED - B:SOCIOS.DBF
PRIMARY USE DATABASE
INDEXES:            KEY EXPRESSION:
  B:SOCSOBRE.NDX    SOBRENOME
UNSELECT DATABASE - B:BICHOS.DBF
  B:DONOS.NDX       SOBRENOME
```

Agora observaremos o material sem processamento desta união — primeiro o SOCIOS:

```
. LIST OFF
Julio     Belo      Praia da Lua                        Recife     PE 92022  0.00 .F.
Carlos    Covas     R. "A", 88                          Sao Paulo  SP 92101  0.25 .T.
Davi      Dunas     Rua das Flores, 123    Sala 17      Aracaju    SE 36222  1.00 .T.
Monica    Medeiros  R. Diva, 2219                       Curitiba   PR 44110  5.99 .T.
Mila      Moreira   R.Elma, 111            Suite 1      Sao Paulo  SP 92109  5.00 .T.
Pedro     Penedo    R.Sao Luiz, 22         Apt. 232     Sao Paulo  SP 92109  0.00 .F.
```

E os BICHOS:

```
. SELE SECO
. LIST OFF
BELO        foca     Ester
BELO        galinha  Poka
BELO        peru     Leo
MEDEIROS    cachorro Bola
PENEDO      cachorro Fido
SIMOES      pombo    Apollo
ZORZI       gato     Arthur
```

Agora o comando JOIN:

```
. SELE PRIM
. JOINT TO tempo FOR !(p.sobrenome) = s:sobrenome
. USE TEMPO       (*Vamos ver ! *)
. LIST OFF
Julio      Belo           Praia da Lua                      Recife     PE 92022
0.00 .F. BELO            foca     Ester
Julio      Belo           Praia da Lua                      Recife     PE 92022
0.00 .F. BELO            galinha  Poka
Julio      Belo           Praia da Lua                      Recife     PE 92022
0.00 .F. BELO            peru     Leo
Monica     Medeiros       R. Diva, 2219                     Curitiba   PR 44110
5.99 .T. MEDEIROS        cachorro Bola
Pedro      Penedo         R.Sao Luiz, 22        Apt. 232    Sao Paulo  SP 92109
0.00 .F. PENEDO          cachorro Fido
```

Há um problema em usar este comando dentro de um programa. A menos que uma listagem de campo seja especificada, aqueles campos nos dois arquivos que partilham nomes de campos comuns serão repetidos no registro de saída com os nomes redundantes. Possuir um arquivo com dois campos nomeados "SOBRENOME" talvez não *o* atrapalhe, mas eu acho que isto representa um tormento.

Recomendamos especificar uma lista de campo para o arquivo de saída todas as vezes que usarmos JOIN. Com esta prevenção, nada mais podemos dizer a não ser que JOIN pode ser um comando valioso.

CREATE FROM

Imagine um livro razão no qual cada conta do cliente é por si só um arquivo de dados. Sempre que uma nova conta fosse aberta o programa teria de criar um novo arquivo com a estrutura apropriada, talvez com uma codificação tal como:

```
USE custo antigo    (*CUSTOANT.DBF é um cliente antigo*)
COPY STRUCTURE TO custonovo     (*CUSTONOV; DBF é o nosso novo arquivo*)
USE custonov    (*Para abrir o novo arquivo*)
(*E depois o resto do programa*)
```

Isto não é tão ruim, mas o dBase nos fornece uma seqüência de comandos mais fácil para realizar esta operação. Primeiro, se dissermos:

```
COPY STRUCTURE EXTENDED TO <nome de arquivo>
```

o dBase formará um novo arquivo que *gravará* a estrutura do arquivo em uso, sob a forma de registros. Ficará mais claro se demonstrarmos – vamos usá-lo com BICHOS:

```
. USE BICHOS
. DISPLAY STRUCTURE                    (*Para ver que estamos iniciando*)
STRUCTURE FOR FILE:  B:BICHOS  .DBF
NUMBER OF RECORDS:   00007
DATE OF LAST UPDATE: 01/01/80
PRIMARY USE DATABASE
FLD        NAME       TYPE WIDTH   DEC
001      SOBRENOME     C    015
002      TIPO          C    008
003      NOME          C    010
** TOTAL **               00034


. COPY STRUCTURE EXTENDED TO BICFORM
00003 RECORDS COPIED                   (*Um para cada campo em BICHOS*)

. USE BICFORM
. LIST
00001  SOBRENOME  C   15    0  )
00002  TIPO       C    8    0  }   (*Estes três registros gravam a estrutura de BICFORM.DBF*)
00003  NOME       C   10    0  )
```

Devemos ler este parágrafo várias vezes. Os campos em BICHOS.DBF foram colocados como registros de BICFORM.DBF. Continuando o raciocínio, a primeira coisa que devemos pensar é que os arquivos resultantes de *todos* os comandos COPY STRUCTURE EXTENDED possuiriam a mesma estrutura – de fato, isto aconteceu. Observe a estrutura de BIC.FORM.DBF:

```
. LIST STRUCTURE
STRUCTURE FOR FILE:  B:BICFORM .DBF
NUMBER OF RECORDS:   00003
DATE OF LAST UPDATE: 01/01/80
PRIMARY USE DATABASE
FLD        NAME       TYPE WIDTH   DEC
001      FIELD:NAME    C    010
002      FIELD:TYPE    C    001
003      FIELD:LEN     N    003
004      FIELD:DEC     N    003
** TOTAL **               00018
```

A segunda implicação desta operação é que poderíamos incluir e extrair campos ou trocá-los através da edição, acréscimo ou apagamento dos registros da estrutura do arquivo – e novamente podemos realizar esta operação:

```
. APPEND

RECORD # 00004
FIELD:NAME:EXTRA:FLD :
FIELD:TYPE:C:
FIELD:LEN : 13:
FIELD:DEC :  0:

. LIST
00001  SOBRENOME  C   15    0
00002  TIPO       C    8    0
00003  NOME       C   10    0
00004  EXTRA:FLD  C   13    0
```

Agora, todo este processo seria mais acadêmico se não houvesse uma maneira para fazer um novo arquivo fora daquele que gravou a estrutura. De fato, há um comando – por que o dBase aumentaria a estrutura? O comando é:

```
CREATE <novo nome de arquivo> FROM <nome de arquivo de estr aum>
```

Como demonstração, vamos usar um CREATE FROM para criar um novo arquivo chamado "BICDEMO.DBF", com as mesmas estruturas de BICHOS.DBF:

```
. CREATE BICDEMO FROM BICFORM
. DISPLAY STRUCTURE                   (*O novo arquivo está em uso*)
STRUCTURE FOR FILE:  B:BICDEMO .DBF
NUMBER OF RECORDS:   00000
DATE OF LAST UPDATE: 01/01/80
PRIMARY USE DATABASE
FLD       NAME      TYPE WIDTH   DEC
001     SOBRENOME    C    015
002     TIPO         C    008
003     NOME         C    010
004     EXTRA:FLD    C    013          (*Este é o nosso novo campo*)
** TOTAL **             00047
```

(&&) DUPLOS

Até aqui devemos estar bem familiarizados com macros. Elas representam comandos poderosos. Vamos desenhar suas figuras lógicas:

& <nome de varmem> ⟶ <conteudos de varmem>

O que aconteceria se os conteúdos de uma variável de memória fossem os nomes de outras variáveis? Experimente:

& <1> ⟶ <2> AND & <2> ⟶ <3> ===> && <1> ⟶ <3>

Vamos (mais uma vez) direto ao exemplo. Primeiro necessitamos de algumas variáveis de memória armazenadas pelo nome:

```
. STORE "Carlos" to C
Carlos
. Store "Mila" to M
Mila
. STORE "Mila" TO m
Mila
. STORE "Janete" TO J
Janete
```

Agora armazenaremos aqueles nomes de variáveis de memória para outras variáveis:

```
. STORE "C" TO A1
C
. STORE "M" TO A2
M
. STORE "J" TO A3
J
```

Vamos observar o que temos:

```
. LIST MEMORY
C          (C)  Carlos
M          (C)  Mila
J          (C)  Janete
A1         (C)  C
A2         (C)  M
A3         (C)  J
** TOTAL **     06 VARIABLES USED  00025 BYTES USED
```

Agora demonstraremos a equação lógica de cima:

```
. ? "&C"              (*"O que temos em C?" *)
Carlos

. ? "&A1"             (*"O que temos em A1?" *)
C

. ? "&&A1"            (*O que temos em (o que temos em A1)?" *)
Carlos
```

Esta peculiaridade interessante de macros pode ser usada, entre outras coisas, para construirmos matrizes do dBase na memória. Uma matriz, acima de tudo, é nada mais do que uma estrutura de dados que pode ser atravessada e manipulada por indexação. Observe a ação de um pequeno arquivo de comandos chamado AMPDEM.CMD ("uma demonstração de '&' "):

```
AMPDEM.CMD

ERASE
STORE 1 TO etapa
DO WHILE etapa < = 3
   STORE "A" + STR (etapa, 1) TO var  (*Esta é a razão de ter escolhido os nomes
   ?                                     das variáveis terminando
   ? "&&& var"                           em inteiros *)
   STORE etapa + 1 TO etapa
ENDDO
```

Vamos ver como isto funciona! Primeiro sem o TALK:

```
. SET TALK OFF
. DO AMPDEM

Carlos

Mila

Janete
```

Não é tão espetacular assim, vamos tentar novamente com o TALK ativado, e desta vez note que cada nível da expansão macro está sendo chamada de volta:

```
. SET TALK ON
. DO AMPDEM
  1
A1

Carlos
  2
A2

Mila
  3
A3

Janete
  4
```

Na verdade, aqui acompanhamos todas as etapas da listagem das variáveis pela indexação – criamos uma matriz, a estrutura de dados esquecida pelo dBase. Note também que os valores resultantes na matriz – "Carlos", "Janete" e nas outras – poderiam ser de tipos de dados diferentes. Vamos ver as matrizes realizarem *esta* operação em Pascal! Mas isto seria injusto ao Pascal, ao dBase e a nós, como programadores, abandonarmos a discussão assim. Não estou sugerindo o uso de macros encaixadas para criar estruturas de matriz parecidas, porque a técnica não é bem empregada para isto.

Além disso, por que usamos matrizes em outras linguagens? Primeiramente, nós a utilizamos para escrever nossas próprias classificações – e o dBase já possui uma classificação. O segundo uso das matrizes serve para chamar novamente os itens de dados de um tipo pelo deslocamento indexado. Por este motivo, deveríamos saber que as macros encaixadas existem, mas antes de fazermos uma matriz fora delas, devemos procurar uma maneira mais simples de realizar o trabalho. Sempre o que é feito com as matrizes em outras linguagens pode ser realizado de outras formas em dBase.

OTIMIZANDO A CODIFICAÇÃO

Como os programas dBase podem ser programados para rodar mais rapidamente? Há maneiras de escrevê-los que aumentam a velocidade de execução?

A resposta para a segunda pergunta é "sim". O restante desta seção é a resposta da primeira. Existem medidas que darão velocidade aos programas dBase – e outras que não.

Vamos lembrar como executamos um arquivo de comandos dBase. Assumindo que um arquivo de comandos entrou através de uma linha de comando "DO <xyz>", o dBase sairá do disco default, encontrará (se puder) o arquivo "XYZ.CMD" e o abrirá, e depois iniciará a ***interpretação*** das linhas de codificação, uma por uma. Prosseguindo este processo, o dBase irá

ao encontro dos comandos "USE <arquivo1> INDEX <arquivo2>, <arquivo3>" ou "APPEND FROM <arquivo4>". Em cada um destes comandos, ele deve ir ao disco e abrir um ou mais arquivos. Cada uma destas operações, chamada um **acesso ao disco**, é muito lenta para os computadores-padrão. Entretanto, seria impossível escrever programas sem esses tipos de comandos.

Dentro dos arquivos de comandos, o dBase também atravessará as linhas de codificação que chamarão outros arquivos de comandos: "DO <abc>" chamado de dentro de XYZ.CMD. Os acessos ao disco podem ou não ser necessários – acima de tudo, poderíamos simplesmente usar nosso processador de texto para "ler" o arquivo ABC.CMD, tornando parte de XYZ.CMD:

Antes

```
XYZ.CMD                  ABC.CMD
linha1
linha2
linha3                   linhaA
DO ABC  } ------------>  linhaB
linha4                   linhaC
linha5
```

Depois

```
          linha1
          linha2
          linha3
          *DO ABC

                         linhaA
                         linhaB
                         linhaC
                         linha4
                         linha5
```

Agora podemos propor dois mecanismos pelos quais a velocidade de um programa dBase poderia ser teoricamente otimizada:

1. Poderíamos eliminar tudo de *dentro* de um arquivo de comandos que fosse desnecessário, a fim de diminuir o tempo que o interpretador deve usar decodificando nosso programa. Faríamos isto, eliminando as linhas de comentário, os espaços extras de linhas e diminuindo todas as palavras dBase para quatro caracteres, como em "DISP STRU" em vez de "DISPLAY STRUCTURE".
2. Poderíamos eliminar *quanto possível* todos os acessos ao disco fora dos arquivos de comandos que estão sendo operados pela combinação de nossos módulos de arquivos de comandos, separados em um grande arquivo de comandos.

Os dois métodos propostos foram testados em seu escritório e calculados em prova-padrão.

Chegamos às seguintes conclusões:

1ª A diminuição dos comandos dBase e a eliminação dos comentários não traz efeito na velocidade do programa. Porém, ele fará com que o programa alcance a ilegibilidade. Não faça isto.

2ª A eliminação do acesso ao disco aumentará a velocidade um pouco, e se o fizermos com discrição, é altamente aconselhável.

Alguns exemplos podem tornar mais claras estas decisões.

A Combinação de Substituições – Sempre uma Boa Idéia

Um comando REPLACE deve levar um acesso ao disco a fim de escrever novas informações dentro do arquivo de dados. Já que a sintaxe de REPLACE nos permite colocar vários na mesma linha (para o limite de 254 caracteres para uma linha de comando), podemos salvar um tempo considerável:

Codificação Lenta

```
REPLACE <campo 1> WITH <var 1>
REPLACE <campo 2> WITH <var 2>   (*Três acessos ao disco*)
REPLACE <campo 3> WITH <var 3>
```

Codificação mais Rápida

```
REPLACE <c1> WITH <v1>,<c2> WITH <v2>,<c3> WITH <v3>
```

A Combinação de Arquivos de Comandos – Sempre uma Boa Idéia

Quando escrevemos programas novos escrevemos arquivos de comandos pequenos para trabalhos simples, afinal esta característica faz parte da natureza da programação dBase.

Esta é uma boa prática, desde que mantemos a codificação estruturada e ativamos a lógica clara. Porém, quando a velocidade de um programa é otimizada, com cada DO encaixado surge a pergunta: deveríamos ler dentro do arquivo que está sendo chamado? Geralmente ele salva um acesso ao disco, mas às vezes este tempo recuperado é deslocado pelo tempo perdido ao interpretador.

Consideremos um grande menu principal usando uma estrutura CASE. Ele poderia ter oito seleções, da qual cada uma chama um submenu. Cada submenu, como resposta, quatro ou cinco chamadas de arquivos de comandos. Agora imaginem um processador de texto ou algum programa utilitário lendo estes arquivos separados – talvez 300 ou 1000 linhas de codificação – em um arquivo de texto longo. Este é o nosso novo programa, e esperamos que seja rápido. Agora

surge o menu principal deste novo programa rápido e fazemos nossa seleção. Optamos pela primeira escolha na estrutura CASE e a primeira escolha do submenu – e, na verdade, esta foi a mais rápida pelos dois acessos ao disco.

Ao último teste. Optamos pela *última* escolha no menu principal pretendendo optar pela última no submenu – simplesmente não estamos alcançando o submenu! Antes de chegarmos lá, o interpretador tem de ler e descartar cada linha de codificação em cada escolha e *em cada submenu e arquivo de comandos.*

Em outras palavras, em linguagens compiladas, o controle pula para a codificação de linguagem de máquina na memória do computador. Mas em uma linguagem interpretada como dBase, os únicos "pulos" que podemos fazer, devem ser mandados pelo programador, usando os comandos "DO" e "RETURN". E embora DO e RETURN tragam o acesso ao disco, às vezes um acesso ao disco de dois a quatro segundos é mais rápido na prática do que um arquivo de comando longo.

APÊNDICE A

Um Resumo dos Comandos dBase

? – exibe uma expressão, variável ou campo.

?? – exibe uma lista de expressões sem alimentação de linha.

@ – exibe os dados do usuário formatado na tela ou impressora.

ACCEPT – permite a entrada de strings de caractere nas variáveis de memória.

APPEND – inclui informações de outros bancos de dados do dBase II, arquivos delimitados, ou no formato SDF.

BROWSE – permite a visualização em forma de janelas e a edição de bancos de dados.

CANCEL – cancela a execução de um arquivo de comando.

CHANGE – permite a edição fora do modo de tela-cheia dos campos de um banco de dados.

CLEAR – fecha os bancos de dados em uso e libera todas as variáveis de memória atuais.

CONTINUE – continua a ação de busca de um comando LOCATE.

COPY – cria uma duplicata de um banco de dados já existente.

COUNT – conta o número de registro em um banco de dados, seguindo um critério específico.

CREATE – cria a estrutura de um novo banco de dados.

DELETE – apaga um arquivo ou marca os registros para apagamento.

DISPLAY – mostra os arquivos, os registros ou estruturas dos bancos de dados, as variáveis de memória ou o status.

DO – executa arquivos de comando ou circuitos estruturados em arquivos de comando.

EDIT – permite a alteração de conteúdos de registros do banco de dados.

EJECT – provoca o salto de uma página na impressora.

ELSE – alterna o percurso da execução do comando dentro de IF.

ENDCASE – finaliza um comando CASE.

ENDDO – finaliza um comando DO WHILE.

ENDIF – finaliza um comando IF.

ENDTEXT – finaliza um comando TEXT.

ERASE – limpa a tela.

FIND – coloca o indicador em um registro correspondendo à chave nos arquivos indexados.

GO ou GOTO – coloca o indicador em um registro específico em um banco de dados.

HELP – acessa arquivo de auxílio.

IF – permite a execução condicional de comandos.

INDEX – cria um arquivo de índice.

INPUT – permite a entrada de expressões nas variáveis de memória.

INSERT – insere um novo registro dentro de um banco de dados.

JOIN – une a saída de dois bancos de dados.

LIST – lista arquivos, registros ou estruturas de bancos de dados, variáveis de memória e status.

LOCATE – encontra um registro que corresponde a uma condição especificada.

LOOP – interrompe a seqüência de um comando DO WHILE.

MODIFY – usado para a criação e edição de arquivos de comando e modificação da estrutura de um banco de dados já existente.

NOTE ou * – permite a inserção de comentários em um arquivo de comandos.

PACK – apaga registros marcados para o apagamento.

QUIT – encerra o dBase e volta ao sistema operacional.

READ – exibe os dados e indica as informações no modo de tela-cheia.

RECALL – apaga a marca para apagamento.

REINDEX – atualiza um arquivo de índice existente.

RELEASE – elimina as variáveis de memória indesejáveis e libera espaço na memória.

REMARK – permite a exibição de qualquer caractere.

RENAME – troca o nome de um arquivo.

REPLACE – troca as informações em um registro, campo a campo.

REPORT – formata e exibe informações.

RESET – informa ao sistema operacional que o disco foi trocado.

RESTORE – recupera as variáveis de memória armazenadas em arquivos.

RETURN – finaliza um arquivo de comando.

SAVE – copia as variáveis de memória atuais no arquivo de disco.

SELECT – seleciona entre os arquivos USE as áreas primária e secundária.

SET – ativa os parâmetros de controle do dBase.

SKIP – adianta ou retrocede o indicador em um banco de dados.

SORT – escreve uma cópia de um banco de dados classificado em um dos campos de dados.

STORE – cria variáveis de memória.

SUM – computa e exibe a soma de um campo.

TEXT – permite a saída de um bloco de texto de um arquivo de comandos.

TOTAL – cria uma cópia resumida de um banco de dados combinando informações dos campos específicos.

UPDATE – permite a atualização em lotes de um banco de dados.

USE – especifica o banco de dados para USE até que o próximo comando seja ativado.

WAIT – interrompe o processamento do arquivo de comandos até que receba uma entrada do usuário.

APÊNDICE

B

Funções do Sistema

@ **(<stringl >, <string2 >)** – função "arroba" produz um inteiro cujo valor é o número em < string2 > que inicia uma substring idêntica a < stringl >.

***** – função de registros marcados para o apagamento; é considerada como uma função lógica verdadeira se o registro atual estiver marcado para o apagamento.

– função de número de registro; indica o inteiro que corresponde ao número do registro atual.

! (< string >) – função de maiúscula; produz < string > em caracteres maiúsculos.

$ (< string >, < start >, < lenght >) – função substring; forma uma string de caractere da parte especificada de outra string.

CHR (< integer >) – converte o caractere de ASCII equivalente da < expressão numérica >. Por exemplo, ?CHR(7) toca o alarme.

DATE( ) – transforma a string de caractere que contém o sistema de data no formato mm/dd/aa.

EOF – função de fim-de-arquivo; considera verdadeiro se for feita uma tentativa para passar o último registro em um banco de dados.

FILE(< file >) – função de existência; considera como uma lógica verdadeira se < file > existir no drive default e uma lógica falsa se não existir.

INT(< numeric expression >) – função de inteiro; separa a parte inteira de um número.

LEN(< string >) – função de comprimento; retorna o número de caracteres de < string >.

RANK(< string >) – transforma o valor (numérico ASCII) do primeiro caractere em < string >.

STR(< Numeric expression >, < width > [,< decimals>]) – converte uma expressão numérica em uma string de caracteres.

VAL(< char string >) – função de valor; converte uma string de caracteres de numerais em uma expressão numérica.

TRIM(< string >) – função trim; elimina espaços em branco de uma < string >.

TYPE(< exp >) – produz uma *série* de um caractere que é um 'C', 'N', 'L' ou 'U' para mostrar se a < exp > é de caractere, numérica, lógica ou um tipo de dado indefinido.

TEST(< exp >) – função para determinar se < exp > é válido e aceitável. < exp > pode ser uma expressão numérica, outra função, um nome de campo ou qualquer combinação (mas não uma palavra de comando dBase). Uma < exp > válida transforma-se em 1.

APÊNDICE

C

Códigos de Movimentação do Cursor

Quando em Tela-Cheia

TODOS OS COMANDOS

Ctrl-X move o cursor para BAIXO – para o próximo campo.
(Ctrl-F tem a mesma função).

Ctrl-E move o cursor para CIMA – para o campo anterior.
(Ctrl-A tem a mesma função).

Ctrl-D move o cursor um caractere à DIRETA.

Ctrl-S move o cursor um caractere à ESQUERDA.

Ctrl-G apaga o caractere abaixo do cursor. <Rubout> ou <DEL> apaga o caractere à esquerda do cursor.

Ctrl-Y limpa o campo atual à direita do cursor.

Ctrl-V liga e desliga o modo de INSERÇÃO.

Ctrl-W grava qualquer alteração efetuada e volta ao dBase.

EM MODO DE EDIÇÃO

Ctrl-U marca e desmarca o registro para APAGAR.

Ctrl-C grava o registro atual no disco e *avança* para o próximo registro.

Ctrl-R grava o registro atual no disco e *volta* ao registro anterior.

Ctrl-Q ignora as alterações do registro atual e volta ao dBase.

Ctrl-W grava todas as alterações no disco e volta ao dBase.

NO MODO BROWSE

Ctrl-B move a janela um campo à DIREITA.

Ctrl-Z move a janela um campo à ESQUERDA.

EM MODO MODIFY

Ctrl-T APAGA a linha atual e reposiciona as linhas que estiverem abaixo.

Ctrl-N INSERE uma nova linha onde o cursor estiver.

Ctrl-C avança metade da página.

Ctrl-W grava todas as alterações no disco e volta ao dBase.

Ctrl-Q ignora todas as alterações e volta ao dBase.

EM MODO APPEND

<enter> finalizará o comando APPEND quando o cursor estiver na primeira posição do primeiro campo.

Ctrl-W grava o registro no disco e avança para o próximo registro.

Ctrl-Q ignora o registro atual e volta ao dBase.

CHAVES DE CONTROLE QUANDO FORA DE TELA-CHEIA

Ctrl-P LIGA e DESLIGA a impressora.

Ctrl-R repete o último comando executado.

Ctrl-X apaga a linha de comando sem executar um comando.

Ctrl-H executa um backspace.

Ctrl-M executa um return.

APÊNDICE

D

Mensagens de Erro

BAD DECIMAL WIDTH FIELD – Digitar novamente o decimal que faz parte da definição de campo.

BAD FILE NAME – Erro de sintaxe no nome do arquivo.

BAD NAME FIELD – Definir novamente o nome do campo enquanto em CREATE.

BAD TYPE FIELD – Deve ser C (caractere), N (numérico) ou L (lógico).

BAD WIDTH FIELD – Definir novamente o tamanho do campo do dado entre 1 e 255.

*****BEYOND STRING** – Escrever novamente a substring ($) com o parâmetro correto.

CANNOT INSERT – THERE ARE NO RECORDS IN DATA BASE FILE – Para incluir um registro, usar o comando APPEND.

CANNOT OPEN FILE – Checar a existência ou integridade do arquivo MEM ou HEX.

COMMAND FILE CANNOT BE FOUND – Checar a escrita e o drive default.

DATA ITEM NOT FOUND – Escrever novamente o comando REPLACE ou checar a estrutura do arquivo para o nome do campo correto.

DATA BASE IN USE IS NOT INDEXED – **FIND** é permitido somente em bancos de dados indexados.

DIRECTORY IS FULL – O diretório do sistema operacional não pode reter mais arquivos.

DISK IS FULL – Não há espaço no disco. Usar DELETE FILE para apagar alguns arquivos desnecessários.

END OF FILE FOUND UNEXPECTEDLY – O banco de dados em USE não está no formato correto. O dBase não tem certeza se é um arquivo DBF.

"FIELD" PHRASE NOT FOUND – Escrever novamente a linha de comando CHANGE.

FILE ALREADY EXISTS – Apagar o arquivo não desejado antes de RENAME.

FILE DOES NOT EXIST – Usar DISPLAY FILE LIKE * .* para ter certeza que o arquivo existe.

FILE IS CURRENTLY OPEN – Digitar um comando USE ou CLEAR para fechar o arquivo.

FORMAT FILE CANNOT BE OPENED – Checar a integridade de um arquivo .FMT.

FORMAT FILE HAS NOT BEEN SET – Ativar o arquivo .FMT apropriado.

ILLEGAL DATA TYPE – SORT não pode classificar em um campo lógico.

ILLEGAL GOTO VALUE – O registro endereçado deve ser >0 e <65.535.

ILLEGAL VARIABLE NAME – São permitidos somente alfanuméricos e dois pontos em variáveis e nomes de campo. Redefinir as variáveis ou nome de campo.

INDEX DOES NOT MATCH DATA BASE – O dBase não pode comparar a chave de índice com o banco de dados. Tentar um outro arquivo de índice.

INDEX FILE CANNOT BE OPENED – Checar a escrita ou a indexação do banco de dados.

JOIN ATTEMPTED TO GENERATE MORE THAN 65.534 RECORDS – A cláusula FOR gerou muitos registros de saída na união.

KEYS ARE NOT THE SAME LENGTH – O comando UPDATE requer chaves idênticas.

MACRO IS NOT A CHARACTER STRING – Para que a variável seja ampliada por uma macro (&), ela deve ser uma string de caractere.

MORE THAN 5 FIELDS TO SUM – SUM está limitado a cinco campos de cada vez.

MORE THAN 7 INDEX FILES SELECTED – O número máximo de arquivos de índice que podem ser abertos é sete. (Menos proporcionará desenvolvimento mais rápido.)

NESTING LIMIT VIOLATION EXCEEDED – Não pode ter mais de 16 arquivos de comandos abertos de cada vez.

NO EXPRESSION TO SUM – O comando SUM precisa de uma expressão numérica para somar.

NO "FOR" PHRASE – Escrever novamente o comando JOIN com a sintaxe correta.

NO "FROM" PHRASE – Escrever novamente o comando UPDATE com a sintaxe correta.

NO FIND – Mais um diagnóstico do que uma mensagem de erro. O dBase não pode encontrar o campo-chave. O # do registro volta com 0.

NON-NUMERIC EXPRESSION – O comando SUM precisa de uma expressão numérica para somar.

NOT A dBASE II DATA BASE – Arquivo DBF aberto não foi criado pelo dBase.

"ON" PHRASE NOT FOUND – Escrever novamente o comando UPDATE ou INDEX com a sintaxe correta.

OUT OF MEMORY FOR MEMORY VARIABLES – Reduzir o número ou o tamanho das variáveis de memória.

RECORD LENGTH EXCEEDS MAXIMUM SIZE (OF 1000) – Reduzir o tamanho de alguns campos ou criar um segundo banco de dados com uma chave comum.

RECORD NOT IN INDEX – Arquivo de índice não foi atualizado após a inclusão de um registro: Indexar o arquivo novamente.

RECORD OUT OF RANGE – Foi chamado um número de registro que é maior do que o número de registros no banco de dados. O arquivo de índice não é o atual; indexar o arquivo novamente.

SORTER INTERNAL ERROR, NOTIFY SCDP – Erro interno; contactar o representante do software.

SOURCE AND DESTINATION DATA TYPES ARE DIFFERENT – Checar se os tipos de dados são numéricos, caracteres ou lógicos.

*****SYNTAX ERROR***** – O dBase não entendeu o comando.

SYNTAX ERROR IN FORMAT SPECIFICATION – O comando @ SAY GET PICTURE foi elaborado errado.

SYNTAX ERROR, RE-ENTER – Os comandos INPUT, ACCEPT e REPORT requisitam uma entrada sintaticamente correta. Pode ser esperado um tipo de dado diferente.

"TO" PHRASE NOT FOUND – Escrever novamente o comando com a sintaxe correta.

TOO MANY CHARACTERS – Somente fora de tela-cheia. Os dados entrados distribuídos excederam o comprimento do campo.

TOO MANY FILES ARE OPEN – Podem ser abertos apenas 16 arquivos de todos os tipos (Command, .FMT, .NDX) de cada vez.

TOO MANY MEMORY VARIABLES – Foi excedido o número máximo de 64 variáveis de memória.

TOO MANY RETURNS ENCOUNTERED – Provavelmente um erro na estrutura de um arquivo de comando. Checar o número e a localização dos RETURNs.

"WITH" PHRASE NOT FOUND – Escrever novamente o comando REPLACE com a sintaxe correta.

UNASSIGNED FILE NUMBER – Erro interno; entrar em contato com o representante do software.

*****UNKNOWN COMMAND** – Checar a escrita. O dBase não entendeu o comando.

VARIABLE CANNOT BE FOUND – Necessidade de criar uma variável ou checar a escrita do nome do campo na estrutura do banco de dados.

*****ZERO DIVIDE** – Foi feita uma tentativa para dividir uma expressão numérica por zero.

APÊNDICE E

Tabela de Conversão de Decimal para ASCII

0—NUL
1—^A
2—^B
3—^C
4—^D
5—^E
6—^F
7—^G
8—^H
9—^I
10—^J
11—^K
12—^L
13—^M
14—^N
15—^O
16—^P
17—^Q
18—^R
19—^S
20—^T
21—^U
22—^V
23—^W
24—^X
25—^Y
26—^Z
27—Escape
28—FS
29—GS
30—RS
31—US

32—Space
33—!
34—"
35—#
36—$
37—%
38—&
39—'
40—(
41—)
42—*
43—+
44—,
45—-
46—.
47—/
48—0
49—1
50—2
51—3
52—4
53—5
54—6
55—7
56—8
57—9
58—:
59—;
60—<
61—=
62—>
63—?

64—@
65—A
66—B
67—C
68—D
69—E
70—F
71—G
72—H
73—I
74—J
75—K
76—L
77—M
78—N
79—O
80—P
81—Q
82—R
83—S
84—T
85—U
86—V
87—W
88—X
89—Y
90—Z
91—[
92—\
93—]
94—^
95—_

96—'
97—a
98—b
99—c
100—d
101—e
102—f
103—g
104—h
105—i
106—j
107—k
108—l
109—m
110—n
111—o
112—p
113—q
114—r
115—s
116—t
117—u
118—v
119—w
120—x
121—y
122—z
123—{
124—|
125—}
126—~
127—Delete

APÊNDICE F

Limites e Limitações do dBase

Número de campos por registro	32	máx.
Número de caracteres por registro	1000	máx.
Número de registros por banco de dados	65535	máx.
Número de caracteres por string de caracteres	254	máx.
Precisão de campos numéricos	10	dígitos
Maior número	1.8 x 10** 63	aprox.
Menor número	1.0 x 10** – 63	aprox.
Número de variáveis de memória	64	máx.
Número de caracteres por linha de comando	254	máx.
Número de expressões no comando SUM	5	máx.
Número de caracteres no cabeçalho de REPORT	254	máx.
Número de campos em REPORT	24	máx.
Número de caracteres na chave de índices	99	máx.
Número de GETs pendentes	64	máx.
Número de arquivos abertos ao mesmo tempo	16	máx.
Comprimento do arquivo de comando (programa)		ilimitado

ÍNDICE ANALÍTICO

!( ), 119
“. 12
#, 49, 121
$ as in <string1>$(<string2>, 126
$ ( ), 125
&, 99
&&, 292
', 12
*, 120, 28
+ as a concatenator, 135
– as a concatenator, 135
<, 108
<=, 137
<>, 138
= as a logical operator, 145
>, 108
>=, 138
?, 89, 104, 184
??, 104, 184
@ x,y, 119
@( ), 119
[ ], 12
^P, 42, 103
^S, 42
ACCEPT, 187
Acesso ao Disco, 296
ADDITIVE, 94, 239
Agrupamento, 2
Alavancas, 42
Alimentação de Formulário, 102
Antecedente, 117
Apagamento
    controles, 40
    marcas, 12, 120
APPEND, 39
APPEND BLANK, 71
APPEND FROM, 71, 240
Área Primária, 143
Área Secundária, 144, 206
Argumentos, 29
Arquivo de Cartões, 10
Arquivo Recheado, 246
Arquivo Nivelado, 245
Arquivo Temporário, 54
Arquivos, 10
Arquivos de Comandos, 145, 295
movimentação, 158
encaixe, 155
Arquivos de Comandos Encaixados, 157, 278
Arquivos de Extensão Fixa, 75
Arroba, 118
ASCII, 12
Asterisco, 28, 120
Ativando as Indexações, 253
Bancos de Dados
    desenho, 241
    tamanho, 28
Bancos de Dados, 1
Bancos de Dados Hierárquicos, 7
Bancos de Dados Relacionais, 9, 210
BELL, 82
Boole-Sr. George, 16
Booleanos, 16
Booleanos em Registros, 248
BOTTOM, 49
BROWSE, 285
Cabeçalho, 26

Campos, 11
CANCEL, 158, 207
CARRY, 83
CASE, 152, 165, 183, 281
com WAIT, 154
CHANGE, 283
Chaves de Classificação, 250
Checagem de Duplicatas, 170
CHR ( ), 126
Cifrão, 125
Classificação Primária, 252
Classificações e Indexações, 5
CLEAR, 93, 206
Comando de Impressão, 89
Comandos Avançados, 282
Comentários, 161
    nas linhas ENDIF e ENDDO, 163
Comparadores Booleanos, 137
Compiladores, 142
Concatenação, 100
Conceitos de Gerenciamento de Bancos de Dados, 10
Condições, 46
CONFIRM, 83
CONSOLE, 83
Console E/S, 182
Construção do Menu, 155
CONTINUE, 52
Controle de Endereçamento do Cursor, 118
Controle de Teclas, 40
Controles de Edição de Tela Cheia, 40
Controles do Cursor, 40
COPY STRUCTURE, 70
COPY STRUCTURE EXTENDED, 290
COPY TO, 69
Coringa, 93
COUNT, 96
CREATE, 36
CREATE FROM, 290
Dado Tipo Caractere, 12
Dados
    definição, 1
    dicionários, 242
Dados
    diagrama de fluxo, 254
    histórico do item, 242
Dados
    agrupamento, 2
    tipos, 12, 116, 133
Dados
    regras de seleção, 17
    visualização, 3
Dados Lógicos, 16
Dados Numéricos, 13
Data, 26
Data do Sistema, 27, 126
DATE ( ), 26, 126
    como strings, 119
dBase
    organização, 143
    limitações, 241
Declarações Booleanas, 138
Declarações da Largura dos Campos Numéricos, 37
DEFAULT, 84
DELETE, 43, 83
DELETE FILE, 86
Delimitadores, 12
DELIMITED, 72
Depuração, 262
Desenho do Registro, 18
Diagramas Circulares, 243
Diálogo de Correção de Comando, 35, 80
DISPLAY, 43, 233
DISPLAY FILES, 85
DISPLAY MEMORY, 265
DISPLAY STATUS, 81
DISPLAY STRUCTURE, 38
DO, 145, 157
DO WHILE, 150
EDIT, 66
EJECT, 104
ELSE, 150
Encaixe, 155
ENDCASE, 152
ENDDO, 150
Endereçamento do Cursor, 192
ENDIF, 151
ENDTEXT, 184
Entrada, 162
Entrada de String, 187
EOF, 127
ESCAPE, 83
Escolhendo Uniões, 236
Escopo, 51
Espaço do disco, 242
Estilo de Programa, 199
Estrutura, 26
Estrutura de um para muitos, 223, 243
Estrutura muitos para um, 211
Etiquetas de Correspondência, 176
EXACT, 83
Expressão, 116
Expressões Aritméticas, 117
FILE IS CURRENTLY OPEN erro, 279
FILE( ), 129
FIND, 58, 233
FOR, 136
FOR circuito, 154
Formatos de Arquivo, 198
Função Booleana de Fim de Arquivo, 127

Função Booleana Substring, 126
Função da Posição de Substring, 119
Função de Comprimento, 130
Função de Maiúsculas, 123
Função Substring, 125
Função Truncada, 129
Funções, 118
GET, 193
GOTO, 48, 233
GOTOs escondidos, 159
Green-Adam, 163
HELP, 33
IF, 150,
IFs encaixados, 156
Impressora - uso, 101
Imprimindo, 102
INDEX, 55
Indexação Linear, 23
Indexações - criação, 250
Indexações Múltiplas, 58
Indicador, 43
Indicador de Registro, 49, 121
Índices Binários e de Árvore-B, 22
Índices de Árvore Binária, 23
Índices de Árvore-B#, 24
Iniciando o dBase, 30
Início, 30
INPUT, 283
INSERT, 286
Instalação, 30
INT( ), 133
INTENSITY, 83
Intercâmbio de Tipos de Dados, 5
Interpretadores, 142
Interrompendo a Impressão, 42
ISAM – Módulo de Banco de Dados, 6
JOIN, 288
LEN( ), 130
Limitações de Campos, 249
LINKAGE, 233
LIST, 41, 233
LIST (ou DISPLAY) MEMORY, 90
LIST OFF, 84
LOCATE, 52, 233
Lógica versus Posição Real, 50
LOOP, 158, 279
Macros, 98, 189, 292
Macros Encaixadas, 293
Maiúsculas, 161
Manipulação de String, 134
Memória de Acesso Aleatório (RAM), 47
Menus e a Programação Estruturada, 259
Modelo de Grade, 19
MODIFY COMMAND, 146
MODIFY STRUCTURE, 78
MODIFY STRUCTURE – regras, 79
Movimentação em Arquivo de Comandos, 158
NEXT, 51
NO FINDs, 279
Nomes de Campo, 37
Nomes de Variáveis de Memória, 37
Número para String, 134
Números de Ponto Flutuante, 15
Números de Registros como Uniões, 239
Opção de União, 232
Operações Aritméticas, 13
Operadores de Concatenação, 134
Operadores Lógicos, 138
OTHERWISE, 152
Otimizando a Estrutura do Registro, 241
Otimizando Codificações, 295
PACK, 45
Parênteses, 118
Pascal, 75
Perguntas Freqüentes, 248
PICTURE, 194
    controle de caracteres, 194
Ponto de Exclamação, 123
Ponto Flutuante, 117
Pontos de Bifurcação, 151
Procedimentos, 157
Procedimentos para Back up, 4
Procedimentos para Depuração, 278
Programação Estruturada, 163, 253
Pseudocódigos, 164, 190
Quebras Comuns, 278
QUIT TO, 87
RANK ( ), 130
RAW, 83
READ, 193
RECALL, 45
RECORD OUT OF RANGE, 60, 279
Refinamento por passos, 255
Registros, 60
Registros de Final Duplo, 226
Registros Delimitados, 20
Registros Fixos e de Comprimento Variável, 20-21
Regras Primárias e Secundárias, 207, 272
RELEASE, 92
RENAME, 86
REPLACE, 68, 297
REPORT FROM, 106, 195, 229, 233
RESET, 86
RESTORE, 93
RETURN, 158, 207, 279
Rotina do Computador, 4
Saída de Arquivos de Dados Unidos, 227
Saída de ASCII, 126
Saída de Impressora Usando @, 195
SAVE, 93, 239

SAY, 193
SDF, 72, 73
SELECT, 206
Seqüência de Intercalação, 13, 130
Seqüência Escape, 102
SET, 82
  ALTERNATE TO/ON/OFF, 105
  CONSOLE, 184
  CONSOLE OFF, 281
  DATE TO, 112
  DEBUG ON, 265
  ECHO ON, 262
  EJECT OFF, 114
  ESCAPE OFF, 282
  EXACT, 238
  format to, 199
  FORMAT TO PRINT, 194
  FORMAT TO SCREEN, 194
  HEADING TO, 113
  INDEX, 55
  MARGIN, 114
  PRINT ON/OFF, 105
  STEP ON, 264
  TALK OFF, 278
  TALK ON/OFF, 105
Sinal Numérico (#), 121
Sintaxe, 31
  função de checagem, 132
Sintaxe de Comando, 31
SKIP, 51, 233
SORT, 53
STORE, 90
STR ( ), 131
String para Numéricos, 134
SUM, 98
Superposição, 30
Superprogramação, 199
SYNTAX ERROR, 280
Tabela de Consulta, 211
Tabelas de Salto, 152
TALK, 84
Tecla Escape, 42
Telas de Auxílio, 184
Telas de Auxílio sem TEXT, 186
Telas de E/S, 192
Terminologia, 10
TEST( ), 132
TEXT, 184
Tipo de Arquivo, 36
TOO MANY MEMORY VARIABLES mensagem, 92
TOP, 49
TRIM ( ), 132, 176
TYPE ( ), 133
Um para Muitos, 211, 243
Um para Um, 243
  registros, 232
  estrutura, 210
União de Dois Arquivos, 210
Uniões, 212
Uniões Calculadas, 237
USE, 38
USING, 194
  controle de caracteres, 194
Utilitários, 203
VAL( ), 134
Valor dos Dados, 4
Variáveis de Memória, 87
Velocidade da Recuperação, 2
Velocidade do Programa, 242
Versão, 30
WAIT TO, 183
WHILE, 281
WordStar - conexão, 75

Composição e Arte:
JAG Composições e Artes Gráficas Ltda.

Fotolito do Miolo
Binhos Fotolito S/C Ltda.

DAG GRÁFICA E EDITORIAL LTDA.
Imprimiu
Av. Nossa Senhora do Ó, 1.782
Tel.: 857-6044

OUTROS LIVROS NA ÁREA

SOFTWARE

Boccia	— dBASE II — Guia do Operador
Byers	— dBASE II — Aplicações Comerciais
Castlewitz	— Visicalc — Guia do Usuário
Ettlin	— Wordstar — Guia do Operador Versão 8 Bits CP/M
Freedman	— dBASE II Para Principiantes
Hogan	— CP/M — Guia do Usuário
Ingraham	— CP/M — Guia do Operador
Mottola	— Linguagem de Programação Assembly para Apple II — 6502
Townsend	— dBASE II — Guia do Usuário — Edição Revisada
Wilson	— Visicalc — Guia do Operador
Zuccolo	— Wordstar — Guia do Operador — Versão 8 Bits CP/M

HARDWARE

Gifford	— Apple II — Guia do Operador
Poole	— Apple II — Guia do Usuário 2/e — II Plus e IIe
Sanders	— Manual do Apple Macintosh

PROGRAMAS

Poole	— Programas Práticos para o IBM PC
Poole	— Programas Práticos em BASIC
Poole	— Programas Usuais em BASIC - Apple II
Poole	— Programas Usuais em BASIC-TRS 80
Poole	— Programas Usuais em BASIC

JOGOS

Wilcox	— Apple II — Jogos

0-07-450033-3

Mc Graw Hill