ARTE DE VIVER

DAVID LAWRENCE

PROGRAMAÇÃO AVANÇADA PARA O AMSTRAD

Para o CPC 464 e CPC 664

PUBLICAÇÕES EUROPA-AMÉRICA

PROGRAMAÇÃO AVANÇADA PARA O AMSTRAD

Obras publicadas nesta colecção:

1 — *111 Receitas com Ovos*, Etelvina Lopes de Almeida
2 — *O Livro do Casal*, Pierre-Marie Brémond
3 — *Aprenda a Fotografar*, Antoine Desilets
4 — *Guia da Interpretação dos Sonhos*, Louis Stanké
5 — *A Arte de bem Receber*, Marguerite du Coffre
6 — *Guia do Comportamento Sexual*, Dubois-Caballero
7 — *Como Reparar Avarias na Estrada — Manual de Todo o Automobilista*, Miguel de Castro Vicente
8 — *Guia Prático e Completo da Costura*, Lise Chartier
9 — *Guia Íntimo das Relações Sexuais*, Pierre Valinieff
10 — *Guia dos Jovens — A Vida e o Amor*, Dr. Benjamin Spock
11 — *111 Receitas de Tapas e Entradas*, Etelvina Lopes de Almeida
12 — *Guia da Futura Mãe durante a Gravidez*, Dr. José M.ª Carrera
13 — *Como Suprimir as Suas Dores com a Simples Pressão de Um Dedo*, Dr. Roger Dalet
14 — *O Livro das Boas Maneiras*, Marcelle Fortin-Jacques
15 — *111 Receitas de Frango*, Jacky Davin
16 — *Doenças Transmitidas pelas Relações Sexuais*, Dr. Lionel Gendron
17 — *Hatha-Yoga*, Suzanne Piuze
18 — *Os Segredos do Amor Táctil*, A. Vignati e O. Caballero
19 — *Como Socorrer o Seu Filho*, Marie Hermand
20 — *Métodos Anticonceptivos e Planeamento Familiar*, Santiago Dexeus e Margarita Riviere
21 — *A Técnica da Fotografia*, Antoine Desilets
22 — *Amor, Sexo e Astrologia*, Teri King
23 — *Como Vencer a Timidez*, François Suzzarini
24 — *111 Receitas de Coelho*, Anne Vernon
25 — *Os Remédios da Avozinha*, Barbara Kamir
26 — *A Mulher depois dos 40 Anos*, Santiago Dexeus e Teresa Pámies
27 — *Viver bem depois dos 50 Anos*, Dr. Hugues Destrem
28 — *Conservas, Compotas e Xaropes*, Maria Emília Abreu Semedo
29 — *Como Proteger a Saúde e a Beleza com a Simples Pressão de Um Dedo*, Dr. Roger Dalet
30 — *111 Receitas para Emagrecer*, Dr. Jean-Paul Ostigny
31 — *Eu... Tu... e os Outros*, Anna Boyer e Isabelle Nicolas
32 — *O Rosto, Espelho do Carácter*, Louis Stanké
33 — *O Seu Aquário de Peixes Tropicais*, Brian Ward
34 — *Pílula — A Solução Mortal*, Dr. Dominique Chatain
35 — *O Seu Futuro nas Cartas*, Louis Stanké
36 — *111 Refeições Naturistas*, Maria Cândida de Albuquerque Cardoso
37 — *A Bíblia do Bridge*, Claude Derwy
38 — *A Congelação dos Alimentos*, Pamela Dotter
39 — *A Celulite*, Gerald J. Leonard
40 — *Guia Sexual da Moça Moderna*, Wardel B. Pomeroy
41 — *101 Conselhos aos Diabéticos*, Prof. Georges Tchobroutsky
42 — *A Beleza pela Saúde*, Dr. Pierre Fournier
43 — *Plantas de Interior*, Brian Ward e Tom Wellsted
44 — *111 Receitas de Cozinha Africana*, Maria de Lourdes Chantre
45 — *Saber Maquilhar-Se*, Josette Ghedin
46 — *111 Receitas de Massas*, Anne Vernon
47 — *Alimentação Natural*, José Lyon de Castro
48 — *A Mulher e o Sexo*, Dr. Lionel Gendron
49 — *A Menopausa*, Dr. Lionel Gendron
50 — *111 Receitas de Arroz*, Déda Frachon
51 — *Trate o Seu Cão, o Seu Gato, os Seus Pássaros com a Simples Pressão de Um Dedo*, Roger Dalet
52 — *A Chave da Longevidade*, Dr. Hugues Destrem
53 — *Tudo sobre Acupunctura*, Dr. Jean Vibes
54 — *101 Respostas sobre a Depressão*, Dr.ª Marie Claude Navikoff e Dr. Jean Pierre Olié
55 — *111 Receitas para Panela de Pressão*, Janet Warren
56 — *Como Vencer as Enxaquecas*, Dr. Claude Loisy e Dr. Sidney Pélage
57 — *Como Viajar de Avião sem Ter Medo*, Afra Botteri/Cécile Gateff
58 — *Tempo Que Mata, Tempo Que Cura*, Dr. Fernand Attali
59 — *Como Manter a Virilidade*, Paul Stanley
60 — *A Alimentação da Criança*, Louise Lambert-Lagacé
61 — *O Sexo e o Amor no Casamento*, Bernard Delon e Germaine Lanoë
62 — *Conheça-Se a Si Próprio — I*
63 — *Conheça-Se a Si Próprio — II*
64 — *Ténis Prático — Técnica — Conselhos — Campos*, Christian Collin
65 — *101 Segredos da Medicina Natural*, Dr. Péron-Antret

66 — *Manual de Protecção contra o Crime*, Ira A. Lipman
67 — *111 Receitas de Caça*, Ana Isabel de Castro
68 — *Manual Médico da Família*, Dr. David Kellett Carding
69 — *A Cozinha Astrológica*, Marie Gebert e Monique Maine
70 — *Doenças de Cães e Gatos Transmissíveis a Crianças*, Silva Leitão
71 — *Manual de Sobrevivência na Situação de Guerra Nuclear — Como Viver durante e após um Ataque Nuclear*, Barry Popkess
72 — *Conheça os Computadores*, John Shelley
73 — *Como Tratar o Seu Filho com a Simples Pressão de Um Dedo*, Dr. Tan Poh Choon
74 — *Tudo sobre Astrologia*, H.-M. de Campigny
75 — *Horóscopos Árabes*, Paula Delsol
76 — *Horóscopos Chineses*, Paula Delsol
77 — *Aventuras com o Spectrum*, Tony Bridge e Roy Carnell
78 — *Enciclopédia dos Pontos Que Curam*, Dr. Roger Dalet
79 — *A Dianética*, L. Ron Hubbard
80 — *Auto-Análise*, L. Ron Hubbard
81 — *Como Vencer no Trabalho e na Vida*, L. Ron Hubbard
82 — *Como Planear e Construir a Sua Lareira*, Margaret e Wilbur F. Eastman Jr.
83 — *As Previsões Astrológicas para 1985*, Catherine Aubier
84 — *Faça Você Mesmo — I — Alvenaria, Telhados, Carpintaria*
85 — *Faça Você Mesmo — II — Electricidade, Canalização, Pintura, Vidraria*
86 — *Faça Você Mesmo — III — Revestimentos, Isolamentos, Refrigeração*
87 — *8 Exercícios para Um Corpo Perfeito*, Sheri Blair
88 — *Guia Prático da Sorte*, Cécile Donner e Jean-Luc Caradeau
89 — *Programação Prática para o Spectrum em Linguagem Máquina*, Steve Webb
90 — *Aplicações Domésticas no seu Microcomputador*, Mike Grace
91 — *Como Fazer Amor com a Simples Pressão de Um Dedo... e não só*, Hsuan Tsai Su-Nu
92 — *111 Receitas de Cozinha Indiana*
93 — *Como Fazer Amor com Um Homem*, Régine Dumay
94 — *Terapêutica Biológica*, Adriano de Oliveira
95 — *Inteligência Artificial no Spectrum*, Keith e Steven Brain
96 — *111 Receitas de Cozinha Chinesa*
97 — *O Spectrum Funcional*, David Lawrence
98 — *Domine o Seu ZX Microdrive*, Andrew Pennell
99 — *Desenvolvimento de Aplicações no Sinclair QL — Ideias práticas para utilizações domésticas e em negócios*, Mike Grace
100 — *Truques de Ilusionismo*, Araújo
101 — *Receitas de Refeições para Bebés, Crianças e Jovens — Crescer com Saúde*, Catherine Lewis
102 — *Comer Bem e Barato com Saúde — Dieta Para Evitar o Cancro*, Carmel Berman Reingold
103 — *A Inteligência Artificial no Sinclair QL — Faça o Seu Micro Pensar*, Keith e Steven Brain
104 — *Manual de Defesa Pessoal*, Prof. J. A. Fonseca Gaspar
105 — *Ervas — Aplicações Culinárias Decorativas e Cosméticas*, Jack Harvey
106 — *A Conservação de Alimentos*, Pamela Dotter
107 — *Jogos de Aventuras para o Sinclair QL — O manual do microaventureiro*
108 — *Manual de Sobrevivência*
109 — *Quill, Easel, Archive e Abacus no Sinclair QL — Como integrar os quatro «pacotes» de software da Psion*, Alison McCallum-Varey
110 — *Problemas de Xadrez — História-Antologia-Iniciação Técnica-Regras Básicas do Jogo*, Rui C. Nascimento
111 — *102 Programas de Jogos para o Amstrad*, Jacques Deconchat
112 — *Programação Avançada para o Amstrad*, David Lawrence

DAVID LAWRENCE

PROGRAMAÇÃO AVANÇADA PARA O AMSTRAD

Para o CPC 464 e CPC 664

ARTE DE VIVER®

PUBLICAÇÕES EUROPA-AMÉRICA

Título original: Amstrad Advanced Programming Techniques

Tradução de F. C. Araújo

Capa: estúdios P. E. A.

First published in English 1985 by:
Sunshine Books (an imprint of Scot Books Ltd)
12/13 Little Newport Street
LONDON WC2H 7PP

Editor: Francisco Lyon de Castro

PUBLICAÇÕES EUROPA-AMÉRICA, LDA.
Apartado 8
2726 MEM MARTINS CODEX
PORTUGAL

Edição n.º 133112/4200

Execução técnica:
Gráfica Europam, Lda.,
Mira-Sintra — Mem Martins

Depósito Legal n.º 13352/86

Índice

Introdução

Pretende-se que este livro seja diferente de qualquer outro da sua biblioteca. Não é uma colecção de programas, nem uma introdução aos comandos disponíveis em BASIC. Também não é uma colecção de rotinas triviais ao nível da conversão de Fahrenheit em centígrados.

Tal como *O Amstrad Funcional,* este livro é dedicado aos que desejam utilizar o micro em tarefas úteis, ou seja, naquilo que é costume designar por aplicações. A era em que toda a potência do micro era utilizada em jogos terminou para sempre. Agora as pessoas querem pôr os micros a trabalhar. O problema consiste em conceber programas para isso.

Existe muita especulação ligada à programação de aplicações. Uma aplicação é, muito simplesmente, algo que permite a introdução de informação, sua armazenagem e processamento para posterior entrega de forma útil. A informação pode ser constituída por nomes e endereços, artigos e preços, registos financeiros, enfim, uma lista interminável de assuntos. A maneira de processar a informação é também muito variada. Alguma é simplesmente armazenada, outra listada ordenadamente, e outras haverá que necessitem de sofrer complexas operações matemáticas. Não existe nenhum livro que possa dar orientações sobre todas as maneiras de processar a informação num micro. Isso dependerá do tipo de informação e do motivo que nos levou a guardá-la.

O que pode ser feito é dar orientações genéricas que permitam chegar ao âmago da questão. Como conceber um programa de maneira a que fique mais apto a funcionar, como expurgá-lo de erros e conseguir que aceite a informação e trate os erros. Como transferir a informação para a memória de forma rápida e económica. Como listar ordenadamente e retirar a informação de estruturas complexas. A parte mais importante da programação de qualquer apli-

cação é feita destas coisas, e este livro tenta explicar como conseguir isto tudo com economia e sucesso.

O nível de complexidade dos assuntos tratados varia imensamente. Nos capítulos seguintes encontrará muitas técnicas que necessitam apenas de uma linha de programa, e outras de longas e complexas rotinas para que os dados sejam inseridos em grandes matrizes a alta velocidade. Incluem-se porque são úteis, não se pretendeu fazer um livro teórico. O que se tentou foi sistematizar técnicas que eu utilizo e que são utilizadas por muitas outras pessoas e encontrar temas e métodos comuns. Não há neste livro nada que não tenha sido utilizado por mim, com vantagem, na solução de problemas específicos.

Desejo que vá buscar o livro, de tempos a tempos, para lançar luz nos inevitáveis problemas que a programação comporta. Acima de tudo, desejo que lhe dê ideias e lhe abra novas áreas de programação. É um livro com ferramentas para a programação e terá cumprido a sua missão não quando compreender essas ferramentas mas quando as puser a funcionar em novas utilizações.

CAPÍTULO I

A programação modular

Pode parecer estranho que num livro sobre técnicas de programação BASIC se comece por um capítulo que tem menos a ver com BASIC que com problemas de estilo e planificação. Há, contudo, uma boa razão para isto. Ao tentar dar resposta às perguntas e problemas dos possuidores de micros que fazem os seus programas, convenci-me de que a maior dificuldade não reside tanto na compreensão do funcionamento do BASIC como na existência de uma certa confusão no modo de o pôr a funcionar num programa.

A técnica de programação que sempre utilizei tanto em livros como nos programas feitos por mim foi a da «programação modular». De maneira simples diremos que isto significa escrever programas constituídos por secções auto-suficientes. A maior parte dos exemplos deste livro estão escritos deste modo. Podem retirar-se da página em que se encontram e serem facilmente inseridos em qualquer programa. A programação modular, no entanto, vai mais longe e constitui mesmo uma filosofia de estilo de programação. Se não quisermos parecer pretensiosos diremos apenas que aplicámos as regras do bom senso à escrita de programas para computadores.

No presente capítulo discutiremos alguns degraus necessários à execução de programas bem sucedidos, degraus que devem começar muito antes de tocar no teclado do micro.

DEFINIÇÃO DO TRABALHO

O pior programa foi o que mais nos entusiasmou antes de começarmos. Tínhamos uma ideia, e cheios de confiança e entusiasmo corremos para o teclado, tentando levar à prática o que

idealizáramos. Se sabíamos o que estávamos a fazer, tentámos rapidamente pôr a parte central do programa em ordem de funcionamento...

...verificámos depois que não havia maneira de introduzir os dados adequadamente e então ou conseguimos arranjar espaço, ou juntámos no fim uma rotina para resolver o assunto...

...depois juntámos algo mais para retirar os assuntos errados...

... depois inventámos uma maneira de tratar as entradas não válidas, que dariam cabo do programa (isto é o que acontece quando outra pessoa pega no programa)...

... depois tinham de ser criadas mais umas linhas para armazenar dados em fita ou disco...

... depois ninguém é capaz de trabalhar com o programa e temos de confeccionar um menu melhor, talvez com algumas instruções...

... depois...

No fim disto tudo o programa é uma bagunça de linhas desordenadas e de GOTO apontando para um lado e para outro. Quando os inevitáveis erros aparecerem, torna-se difícil saber o que os originou e ainda mais onde é que estão, e perdem-se dias ou semanas a corrigi-los.

Os melhores programas são muitas vezes os que somos obrigados a escrever sem sabermos como. A razão é simples. Quando não estamos confiantes que sabemos escrever o programa desejado, sentamo-nos a pensar no assunto. Deste modo ultrapassámos um dos mais difíceis obstáculos na concepção de um programa bem sucedido.

Um programa não é só o meio de executar a tarefa principal de, por exemplo, calcular o imposto ou armazenar ficheiros de nomes e moradas. É também um sistema complexo de funções que devem conduzir à adequada introdução da informação e sua mais conveniente apresentação na saída. Deve ainda ser capaz de indicar claramente o seu modo de utilização, de lidar com erros e de permitir correcções à informação anteriormente introduzida. Todas estas tarefas, e muitas mais, são tão importantes como a rotina principal. Mesmo considerando somente a tarefa principal e as linhas necessárias à sua execução, a primeira ideia que temos raramente é a melhor. Queremos calcular o imposto, mas que desejamos incluir? As despesas dedutíveis? Ter em conta o efeito da aplicação de diferentes taxas? Qual o período a cobrir? Se for superior a um ano, que acontecerá se a taxa variar de ano para ano (ou até no mesmo

ano)? Desejamos que utilize análises do tipo «Que acontecerá se...», perguntas como «Que acontecerá se o nosso rendimento aumentar 15 000$00 por mês?», sem alterarmos os dados já introduzidos? Teremos de armazenar a informação ou introduzimo-la de novo cada vez que fizermos correr o programa? Será de incluir o rendimento do cônjuge? Estas são questões que tenho agora na ponta da língua. Após um par de horas a trabalhar, você será capaz de encher uma folha com as coisas que necessitará de saber antes de começar a programar o que antes lhe tinha parecido óbvio. Se não pensar nalgumas implicações do programa, pode suceder que o que escreva funcione; porém, não se esqueça de que, por esse mundo fora, existem montes de cassetes cheias de programas que *funcionam,* mas nada têm de *útil.*

A primeira coisa a fazer para escrever um programa é pensar nele. Escrevemos em linguagem corrente tudo o que queremos que o programa faça. Feito isto deixa-se de parte por uns tempos e volta-se à carga mais tarde. Raras vezes parecerá tão perfeito da segunda como da primeira vez, por isso acrescentaremos algumas funções. Se o programa se destinar a ser utilizado por outra pessoa, teremos de acrescentar o que ela espera encontrar. Escusa de tentar convencer os seus filhos de que o novo programa que escreveu é muito interessante e divertido, se o programa não fizer o que eles querem que faça. Um programa ou faz o que as pessoas esperam que faça ou não. Se não faz, perdemos tempo.

A conclusão é que devemos sempre sobredimensionar as especificações do programa e planificá-lo para níveis superiores aos das necessidades inicialmente pensadas. Pode acontecer que, quando se passar ao planeamento pormenorizado do programa e respectiva escrita, algumas das especificações tenham de ser deitadas fora por falta de memória ou de conhecimentos. A maior parte das vezes, no entanto, descobrirá que o programa que planeou *pode* ser escrito. Os programas que tenham em conta um certo nível de exigências serão aqueles que irá buscar de vez em quando.

PLANEAR O PROGRAMA

Não tratámos já disto? Bem, ainda não, até agora não pensámos no programa para computador. Definimos apenas a tarefa. Se trabalhámos bem, ainda não demos atenção àquilo que sabemos

fazer, ou àquilo que pode ser feito no 464. O objectivo agora é partir as especificações idealizadas em pequenas unidades que o micro possa digerir, e nós programar. Possivelmente podemos fazer isto melhor em duas fases. A primeira fase será definir as grandes áreas do programa, tais como *input, output,* processamento de dados, armazenamento, etc. Concluída esta tarefa mais simples segue-se a mais difícil, que consiste em identificar as unidades a que chegámos, ou módulos, a partir dos quais o programa é construído.

Agora a regra é dividir as funções do programa em unidades tão pequenas quanto possível, mesmo se isso levar a dividir funções que sempre andaram juntas. Um programa rigorosamente dividido em unidades funcionais é sempre mais fácil de escrever e de emendar quando chegar a fase inevitável de corrigir os erros. Para mais, paradoxalmente, os programas bem divididos em unidades funcionais são, muitas vezes, mais pequenos que os que ficam com funções indiferenciadas no conjunto. A razão disto é que quando se procede à divisão criteriosa acabamos por descobrir que certas funções são utilizadas repetidas vezes por diferentes partes do programa, em vez de repetidas em diferentes lugares.

Pensemos no exemplo do programa do imposto que atrás mencionámos. Para a impressão de dados no visor, se a quantia a imprimir for negativa, será necessário fazer uma pequena rotina que modifique a cor INK para vermelho, ou, então, colocar um sinal menos, caso os dados sejam enviados para impressora. Para isto não serão precisas apenas duas linhas de programação e poderemos facilmente colocá-las na parte referente à saída de dados. Depois, quando for necessário apagar assuntos, veremos, contudo, que seria interessante imprimi-los no mesmo formato, para que o utilizador possa ver o que vai ser apagado. O mesmo acontece quando se introduz algum dado, já que gostaríamos também de ter a reprodução no visor, no formato adequado. Como colocámos a nossa pequena rotina noutro lado do programa, não há maneira de a utilizar agora e teremos de a escrever cada vez que dela precisarmos.

Para vermos que isto acontece basta olhar para os programas impressos na contracapa das revistas de informática, onde as repetições deste tipo são muito frequentes. Nos programas adequadamente escritos pode-se aceder a qualquer função a partir de qualquer ponto do programa. Na verdade, com muita frequência, nos programas adequadamente escritos, podem ser adicionadas maiores capacidades, simplesmente através da chamada de uma série de funções, ordenadas de forma diferente.

A longo prazo, o mais importante é que, quando tivermos três

ou quatro programas escritos, em que cada função esteja claramente individualizada, a tarefa de escrever os programas seguintes estará enormemente simplificada. Realmente, são poucas as técnicas importantes em programação, e uma vez que as tenha escrito num programa verá que as vai buscar com frequência, para montar novos programas, tal como acontece com os brinquedos de armar das crianças.

A técnica principal, para identificar cada uma das funções necessárias, é descrever cada aspecto do programa, para si, como no exemplo seguinte:

> «Nesta secção do programa quero dar instruções do modo de introduzir dados a partir do utilizador, procurar erros nos dados, perguntar ao utilizador se os dados são os pretendidos e, depois, colocá-los no local adequado da memória.»

Nesta frase relativamente simples, e pouco técnica, identificámos já pelo menos cinco funções do programa que devem ser escritas imediatamente. Percorra, deste modo, todas as grandes áreas do programa e acabará numa lista de funções que parecerão suficientes para fazer o programa.

Necessitamos agora de identificar as funções que podem ainda ser divididas. Assim, no exemplo anterior, dissemos que a entrada deverá ser impressa no visor, para que o utilizador veja se está correcta. Se deseja formatar esta passagem pelo visor terá de construir a rotina respectiva, como vimos mais atrás. Outro exemplo é o da função, mencionada em último lugar, de inserir os dados correctamente no lugar respectivo da memória. Se os assuntos ficam por ordem alfabética, ou de datas, cada inserção implicará, primeiro, uma busca da localização correcta, e depois arranjar espaço para o assunto. São dois processos diferentes, conforme verificaria se mais tarde quisesse dar ao utilizador a possibilidade de especificar um determinado assunto que quisesse chamar de novo e imprimi-lo no visor. Isto basta para ver que a rotina de busca está ligada às mesmas linhas que inserem o assunto.

Tal como se procedeu com as áreas de programação global, há que esmiuçar cada uma das funções do programa descrevendo-as. Se terminarmos com duas ou três acções distintas, para uma mesma função do programa, é possível que se esteja perante uma candidata a outra divisão.

No fim deste processo deverá ser encontrada uma lista de fun-

ções de que, presumivelmente, o 464 necessitará para executar o que se pretende. Teremos também uma ideia do programa no seu todo, tamanho, estrutura que poderá ter e das áreas que irão dar mais problemas. Como verificação final percorrem-se as operações principais que gostaríamos de executar com o programa, utilizando as notas que tirámos, em vez do 464:

> «Ligar, RUN (lá me esqueci do menu para indicar o que ele deve fazer), especificar que vou introduzir dados, confirmar a entrada, introduzir outro assunto...»

Se tudo estiver correcto, as notas que tirámos funcionarão quase tão bem como o 464.

ESCREVER OS MÓDULOS

Nesta altura, a tentação de agarrar no 464 e começar a matraquear o teclado é quase irresistível — mas ainda não é desta! O pior local para escrever um programa é o computador e o melhor será numa folha de papel. Não digo que todas as linhas tenham de ser escritas antes da introdução de qualquer coisa. É neste momento que se torna necessário conceber um modelo funcional de cada função do programa. Estes modelos vão guiar-nos quando estivermos a introduzir as funções, como módulos do programa, mas não serão certamente tão pormenorizadas como o próprio módulo.

O método mais conhecido de desenvolver um programa com algum pormenor é através de diagramas de fluxos (ou fluxogramas). O problema é que, apesar de os profissionais de informática recomendarem o seu uso, os donos dos micros não os utilizam. Mais realisticamente, penso que é de utilizar «uma linguagem de desenvolvimento de programas» — LDP (ou PDL na forma inglesa). Nas mãos de programadores de elevado nível, isto tornou-se assustador como instrumento, e é de facto uma ferramenta demasiado difícil para a maioria dos programadores de BASIC. Não me refiro a tão inacessível instrumento, mas sim a uma mistura elementar de BASIC e português que se assemelhará ao módulo final, mas que é muito mais rápida de escrever.

Para terminar vou exemplificar escrevendo o módulo de entrada como segue:

```
//    PRINT (TÍTULO)
      PRINT (COMANDOS DISPONÍVEIS) NOVO AS-
      SUNTO/SAIR
%%    INPUT «QUANTIA»; TA
      IF TA=-9999 TERMINA ESTA SECÇÃO
      GO SUB VERIFICAÇÃO DE ERROS
      IF ERRO REPITA DESDE %%
      INPUT «DESCRIÇÃO»; TEMP_D$
      INPUT «QUANTIA»; TEMP_QUANTIA
      CALL FUNÇÃO FORMATO
      CLEAR A PARTIR DO MENU
      PRINT DESCRIÇÃO (TD$)«;» QUANTIA (FOR-
      MATO$)
      INPUT «CORRECTO»; Q $: «N» REPITA DESDE %%
      CALL FUNÇÃO DE BUSCA DE LUGAR
      CALL FUNÇÃO INSERÇÃO DE DADOS
      REPITA DESDE //
```

VARIÁVEIS NECESSÁRIAS: NENHUMA
VARIÁVEIS CRIADAS:
TA = armazenamento temporário para a quantia
TD$ = armazenamento temporário para a descrição
Q$ = entrada temporária
FORMATO$ = quantia formatada pela rotina de formatação

Escrito desta maneira vê-se perfeitamente para que serve o módulo e tem-se uma ideia clara de como o programa pode ser feito. Repare que escrevi algumas coisas por extenso, porque não compensava abreviar. Para as funções que se referem à impressão dos assuntos no visor, é mais fácil apenas referi-las e deixar para a última entrada do módulo a decisão do número de linhas do visor a limpar, de escolha da cor para o título e menu se o assunto tiver de ser introduzido de novo. Algumas vezes incluo também uma linha para descrever um procedimento que no momento não vislumbro como descrever mas sei que é solúvel com o desenvolvimento do trabalho. Nestes casos pode colocar-se uma nota junto da linha e posteriormente escrever uma folha correspondendo à anotação.

Não numerei as linhas, em parte porque as não descrevi exaustivamente e também porque me atrasaria a escrita. Uma ou outra referência tal como // e %% é suficiente. Também não dei destino aos GOSUB, indicando apenas as rotinas a que dizem respeito e que podem, ou não, estar feitas. No momento de passar à execução

do programa decido o conjunto de linhas para cada módulo, com preferência por blocos de pelo menos 1000 cada módulo, e tomo nota da localização, no cimo de cada uma das folhas, do conteúdo das funções do programa.

No esboço anterior não se vê qualquer comando GOTO. Em parte porque a pormenorização ainda não atingiu esse estádio. Uma das vantagens de pensar em termos de funções em vez de linhas pormenorizadas é encorajar a identificar as partes do programa e incluí-las em sub-rotinas ou ciclos WHILE com condições apropriadas a definir quando, e se, forem utilizadas. O desenvolvimento do BASIC nos últimos anos tem sido no sentido da eliminação da instrução GOTO, com base na noção de que conduz a programas confusos. Programas entrelaçados de GOTO podem ser eficientes, se forem bem escritos, mas tornam-se de difícil ou mesmo impossível compreensão e desenvolvimento em fases posteriores. O estilo da programação actual leva a escrever os programas em unidades facilmente identificáveis, executáveis uma ou mais vezes, com base em condições claramente expressas.

No fim da listagem está a indicação das variáveis necessárias. Algumas servem para lembrar as variáveis que têm de ser definidas antes de utilizar o módulo. As variáveis temporárias, utilizadas somente dentro do módulo e cujo conteúdo será depois esquecido, ou transferido para outras variáveis de armazenamento definitivo, podem ser repetidas noutros módulos. Outras há cuja repetição inadvertida, em circunstâncias diferentes e noutros módulos, pode originar o caos.

É óbvio que uma aproximação tão afortunada não é possível em todos os módulos. Alguns têm uma ou duas linhas de cálculos matemáticos, e estes ficarão por extenso. Em geral um método como este, adaptado às necessidades de cada um de nós, permite dar uma imagem nítida do programa e é uma forma de introduzir mais rapidamente o módulo do que começar directamente no teclado. Não pretendo escravizar o leitor à utilização do exemplo anterior. Pretendo, sim, fazer notar que o programa pode ser escrito muito mais rapidamente, e de forma mais clara, utilizando este tipo de aproximação.

A INTRODUÇÃO DO PROGRAMA

Agora podemos pegar no 464 para introduzir o programa, com a certeza de termos alguma coisa digna desse nome. Todo o equipamento necessário é um molho de folhas de papel com notas referentes às várias funções do programa. O que ninguém deseja é introduzir o programa todo de uma só vez.

A programação modular permite a correcção dos programas à medida que são introduzidos, e é muito mais fácil corrigir módulo a módulo, à medida que vão sendo introduzidos, do que corrigir o programa todo de uma vez, quando já fizemos RUN. A correcção nesta fase é incompleta, dado que alguns erros só se tornarão aparentes com a interacção de todos os módulos, mas evitam-se mais tarde dores de cabeça. Para corrigir o programa à medida que é introduzido torna-se necessário identificar os módulos mais frequentemente utilizados, mesmo que com objectivos triviais, e introduzi-los em primeiro lugar.

No caso do módulo para introdução de dados que listámos anteriormente, podemos testá-lo sem a presença das rotinas PLACE SEARCH (BUSCA DE LUGAR) ou DATA INSERT (INSERÇÃO DE DADOS). Estas rotinas podem ser substituídas muito simplesmente por duas instruções de RETURN, no ponto certo de início das linhas dos blocos que eventualmente ocupariam. Não se pode no entanto testar o módulo de introdução de dados sem ter introduzido primeiro as rotinas de ERROR CHECK (VERIFICAÇÃO DE ERROS) e FORMAT (FORMATAÇÃO). Depois dar entrada a estas duas pode introduzir dados à vontade. O funcionamento do módulo nesta fase não corresponde a nada, mas é suficiente para vermos se a impressão no visor está correcta e se as entradas são aceites e testadas. Não é possível conseguir uma sequência de introdução de módulos totalmente correcta. Ver-se-á muitas vezes na contingência de afectar valores a uma ou outra variável em módulo directo (por exemplo, LET A$ = «TAX REBATE» (DESCONTO DO IMPOSTO), sem indicação de número de linha, e depois introduzir GOTO para iniciar a rotina (RUN apagaria o valor das variáveis que tivesse introduzido). Algumas vezes terá de introduzir alguns módulos em conjunto, como por exemplo, PLACE SEARCH e DATA INSERT (INSERÇÃO DE DADOS), dado que trabalham muitas vezes em série. Apesar destas excepções à regra, pode-se tentar sempre testar tudo, o mais cedo possível, após a introdução, com a certeza de que cada erro detectado é relativamente mais fácil

de encontrar e corrigir, por haver 95% de probabilidades de estar no último módulo introduzido.

O resultado ainda não é um programa sem erros, mas haverá muitos menos do que se tivéssemos batido no teclado logo do princípio até ao fim. Pelo menos deve acabar o programa sem um único «SINTAX ERROR» (erro de sintaxe), dado que todas as linhas do programa terão corrido antes de o programa estar todo introduzido.

SUGESTÕES E INDICAÇÕES

Damos de seguida algumas indicações a ter em conta na escrita de programas por módulos. A lista não é exaustiva, mas representa o tipo de assuntos que muitas vezes são negligenciados na programação de micros:

1) Os programas ficam mais claros se dermos a todos os módulos um título explicativo. Por mim utilizo sempre um formato do tipo:

```
1000 REM **********
1001 REM NOME DO MODULO
1002 REM **********
```

Esta forma referencia o módulo no programa e traz benefícios incalculáveis quando tivermos de o listar de novo passado algum tempo.

2) Em todos os GOTO e GOSUB devemos indicar o número da linha do título da rotina que mencionámos em 1). A razão é que no título não se mexe habitualmente, mas as linhas seguintes podem deixar de existir, ou ser alteradas, e então lá apanhamos com a mensagem de erro «LINE DOES NOT EXIST» todas as vezes que o módulo for chamado. A posição do título não muda, e por isso as alterações feitas ao conteúdo da rotina não trarão problemas.

3) Seja liberal na ornamentação do programa com comentários incluídos em instruções REM em todos os pontos que, no mínimo, possam vir a levantar dúvidas. Três meses mais tarde, quando

quiser alterar o programa, abençoará o dia em que teve esse trabalho. Sem os comentários levará algum tempo a saber o que se passa. As anotações podem ser feitas no fim de uma linha utilizando os sinais «». Pode tornar-se mais claro o significado do programa, espaçando as secções com linhas em branco contendo simplesmente dois pontos, por exemplo:

1760:

4) Todos os programas, mas principalmente os maiores, ficam mais fáceis de compreender se utilizarmos nomes descritivos para as variáveis, mesmo que fiquem extensos.

5) A maior parte das variáveis utilizadas num programa modular são, ou devem ser, temporárias e são utilizadas apenas no decurso de um módulo, podendo em seguida desaparecer. Não há nada contra cada uma ter uma designação diferente; mas, normalmente, um ou dois nomes como TEMP$, Q$, TEMP e Q cumprem a função sem confusão, já que são esquecidos no fim do módulo.

6) Algumas são menos temporárias que outras. No módulo de introdução de dados, que se apresentou anteriormente, os dados são armazenados em duas variáveis (TA e TD$), que vão ser passadas para outros dois módulos, antes de serem colocadas nas matrizes de dados principais do programa. Neste caso será útil principiar o nome da variável pela letra «T», para indicar que é temporária, mas juntar uma ou mais letras que recordem a sua função, tendo sempre presente o perigo da duplicação de nomes.

7) A disposição dos módulos no programa requer algum cuidado. As duas considerações principais a ter em conta são:

a) Os módulos usados com mais frequência são executados ligeiramente mais depressa se forem colocados no início do programa.

b) Por outro lado, não há nada mais difícil de compreender que um programa em que os módulos aparecem todos misturados, sem ordenação lógica.

Se não está a fazer nada excessivamente complicado, que faça perder muito tempo e envolva grandes quantidades de cálculos, é

melhor dispor o programa de forma a que os módulos fiquem em grupos lógicos, e no fim os pequenos módulos que são utilizados repetidamente em várias áreas do programa (como a rotina FORMAT, no exemplo dado atrás). É extremamente improvável que você se aperceba de qualquer diferença da velocidade.

8) A construção WHILE... WEND pode ser utilizada para clarificar a estrutura do programa, marcando claramente as suas secções e as condições em que decorre. Algumas vezes as variáveis especiais necessitam de ser declaradas para se conseguir o efeito desejado da instrução WHILE, tal como executar uma secção de código só uma vez, e não repetidamente, se for verificada uma certa condição. Normalmente tais linhas são precedidas por uma instrução GOTO, tal como:

```
1760 IF A<10 THEN GOTO 1810
```

que garante a execução da secção somente se a variável A for maior que 10.

Um método ligeiramente mais longo, mas que ajuda a tornar mais clara a estrutura do programa, é utilizar um ciclo:

```
1760 done = 0
1770 DO WHILE done = 0 AND A> = 10
—
—
—
1810 done = 1
1820 wend
```

9) A iniciação do programa, isto é, a declaração das diversas matrizes e variáveis, constitui uma função separada do programa e deve ter um módulo separado, com nome ou sem nome. Isto permite afinar o programa quando for usado pela primeira vez e ainda limpar qualquer dado chamado posteriormente. Também se podem fazer os programas de modo a serem «auto-iniciados», ou seja, afinar as matrizes quando for necessário, mas evitando limpar a memória se já existirem dados armazenados. Para isto, a secção do programa que dimensiona as matrizes e limpa a memória é colocada no início do programa e dentro de um ciclo que testa uma variável importante, para ver quando é nula. A variável escolhida terá a característica de nunca admitir o valor zero quando o programa

tem dados na memória. O ciclo faz que a rotina de iniciação seja ignorada se a variável escolhida não for igual a zero. Repare no exemplo seguinte:

```
1000 REM ************
1001 REM Inicializar
1002 REM ************
1010 DONE = 0 : WHILE DONE = 0 AND ITEMS = 0
1020 CLEAR
1030 DIM A$(10),A(500),B(100),C$(100)
1040 DONE = 1 WEND
```

Utiliza-se aqui ITEMS para registar o número de assuntos armazenado pelo programa. Se quiser introduzir alguns assuntos, pare o programa e recomece-o com GOTO 1000. Em vez de perder os dados, o ciclo salta por cima das linhas que limpam a memória e deixa as matrizes prontas para novos dados. Quando desejar limpar os dados existentes basta começar com RUN, já que assim limpa a memória e põe a zero todas as variáveis, incluindo ITEMS, e o ciclo será executado.

10) Todo o programa digno desse nome tem um menu que, pelo menos, esboça o que o programa faz e indica o que se pode fazer com as suas diferentes funções. Além deste menu principal, muitos dos módulos, se derem acesso a mais de uma função, podem ser melhorados através de um pequeno menu.

11) Dê um intervalo grande entre a numeração das diferentes linhas do seu módulo. Quando, posteriormente, desejar desenvolver o programa (e não duvide que o fará), não há nada mais aborrecido que ter de estragar uma estrutura arrumada, quer através de renumerações sucessivas, quer de linhas cuja numeração é espremida até ficar de um em um, quer ainda juntando no fim novos módulos que ficariam melhor se fizessem parte de um grupo no meio do programa. Um espaçamento de 2000 para cada módulo não será demasiado para a maioria dos programas.

12) Lembre-se de que, feito o programa na forma modular, é fácil mudá-lo. Esteja alerta à possibilidade de conseguir melhorar as funções que lhe interessem. Quando descobrir um melhoramento numa revista, retire o módulo original e substitua-o por outro que incorpore a nova técnica. Deixar de actualizar o seu programa sig-

nifica perder uma das maiores vantagens deste estilo de programação.

13) É por vezes vantajoso armazenar em separado os módulos que contêm técnicas importantes, isso além de os incluirmos num determinado programa. Deste modo pode-se facilmente adicioná-lo a programas novos através do comando CHAIN MERGE. Lembre-se de que pode precisar de mudar os números das linhas do programa antes de inserir o módulo.

14) Para finalizar, lembre-se de que quase tudo pode ser modularizado. Os programas de bons profissionais contêm um grande número de sub-rotinas de duas ou três linhas. É possível ir muito longe com este processo, mas não é fácil.

CONCLUSÃO

Há uma pequena dúvida em considerar este o capítulo mais fácil de deixar para trás. Não contém nem exemplos práticos nem de BASIC funcional. Somente pondo em prática os princípios deste capítulo tirará partido do material contido neste livro. É este o capítulo que tornará substancialmente diferentes as suas tentativas de escrever com sucesso programas úteis e compreensíveis.

CAPÍTULO 2

Eliminar erros

Uma revista de informática publicou recentemente uma série de artigos com as diferentes linguagens de computador presentemente existentes. Quando a série terminou, um leitor escreveu para indicar que se tinham esquecido de uma que é comum a todos os utilizadores, qualquer que seja a sua máquina — a das pragas.

Quando tiver introduzido o seu programa (testando-o à medida que progride), e o tiver corrido muitas vezes sem problemas aparentes, chegará a altura em que esbarrará num caos. Isto já é bastante mau num programa feito por si, cujo método de funcionamento compreende. É muito pior se for um programa de outro, talvez de um livro ou de uma revista. Mesmo com a melhor das boas vontades do mundo não terá desenvolvido o instinto que lhe permita ver o que se passa. Sei de alguns leitores meus que passaram semanas, ou meses, a tentar descobrir erros num programa — erros cometidos por eles ao introduzi-lo. Na altura que me contactaram estavam à beira do desespero, convencidos de que se o 464 deles não estivesse a funcionar mal estaria eu certamente. A verdade é que, na maioria dos casos, poderiam ter resolvido os problemas nalguns minutos se conhecessem e usassem algumas técnicas simples.

INFORMAÇÃO-CHAVE PARA ELIMINAR ERROS

Quando se encontra um erro num programa a primeira coisa a ter em conta é que toda a informação necessária para o localizar está na memória do 464 e que a última coisa que desejamos é apa-

gar esses valiosos elementos. Por isso, quando encontrar um erro, nunca o ignore, nem execute o programa na esperança de que ele não o volte a incomodar. Pode acertar, e o programa correr perfeitamente, mas perdeu a oportunidade de remover um erro que inevitavelmente voltará à carga no futuro, talvez quando tiver na memória informação valiosa. A primeira regra da caça ao erro é perseguir cada erro no momento em que é detectado.

Depois de tomar esta decisão, como irá optimizar a informação disponível? Para o decidir, tem de saber o significado dessa informação:

1) Pode acontecer que a paragem do programa seja provocada por um tipo de erro localizado numa determinada linha. Neste caso o 464 identifica o tipo de erro e localiza-o no programa. A mensagem de erro terá a forma:

?(MENSAGEM DE ERRO) IN (NÚMERO DA LINHA)

2) Alguns erros podem parar o programa mesmo que a linha em que pára pareça totalmente correcta. No mesmo grupo se incluem os erros que não param o programa mas conduzem a resultados absurdos. Este tipo é o mais difícil de descobrir, porque, na maior parte dos casos, não há indicação da sua localização.

MENSAGENS DE ERRO DE UMA LINHA

Na maior parte dos casos em que existe mensagem de erro, metade dos problemas ficam resolvidos, porque o 464 indica a linha do programa que necessita de ser revista. Pode acontecer que a correcção tenha de ser feita noutra linha ou linhas, mas a chave que conduz à natureza do erro está sempre na linha indicada na mensagem. Na maioria das mensagens de erro não será preciso procurar para além da linha indicada; no entanto, tal facto não é admitido pela maioria dos utilizadores.

Veja, por exemplo, o caso da mensagem de erro mais comum, SYNTAX ERROR. Onde quer que apareça esta indicação você sabe que deve ter introduzido de maneira errada a linha em questão. De nada serve examinar a linha e concluir que parece em condições e, de seguida, correr de novo o programa, ou tentar corrigir outras linhas, para pôr o programa a funcionar. Existe algo na linha que o

464 não reconhece como BASIC, e o programa não funcionará até que se corrija a linha.

As mensagens de erro que normalmente apontam a sua localização são: SYNTAX ERROR, LINE DOES NOT EXISTE, TYPE MISMATCH STRING EXPRESSION TOO COMPLEX, UNKNOWN USER FUNCTION, OPERAND MISSING, LINE TOO LONG, UNKNOWN COMMAND.

O modo de actuar com estas mensagens indicadoras de erro em determinada linha é o seguinte:

1) Lista-se o programa até a vizinhança próxima da linha, para deste modo a colocarmos no seu contexto.

2) Lista-se a linha, e só ela, com uma ou duas linhas de intervalo das anteriormente listadas. Procedemos assim porque é surpreendentemente fácil no 464 introduzir duas linhas de tal modo que são recebidas pelo computador como uma só.

Tais erros são praticamente impossíveis de localizar sem listarmos a linha em separado. Repare no seguinte:

```
10 X=10 : LET Y=X+5 : LET Z=X+Y+1234
20 PRINT Z
```

Imagine a frustração que não será correr este programa e aparecer sempre a piscar no visor «Syntax error in 10». A busca pode levar horas ou dias. O que aconteceu foi que no fim da linha 10, em vez de premir RETURN, se introduziu um espaço, levando o cursor para o princípio da linha seguinte, depois introduziu-se a linha 20. Para o 464, a versão definitiva da linha 10 é

```
10 LET X=10 : LET Y=X+5 : LET Z=X+Y+123420 PRINT Z
```

Não é de espantar que o 464 ache isto difícil de interpretar.

3) Se não existirem erros grosseiros na linha listada, teremos de examinar instrução por instrução e carácter por carácter.

Uma maneira de ter a certeza de não caminhar depressa de mais ao longo da linha é colocar o cursor no início da linha e depois, usando a tecla de «cursor para a direita», movê-lo vagarosamente ao longo da linha, examinando cada carácter e cada instrução que dela fazem parte. A maior parte dos erros de sintaxe (e dos outros que resultam da não compreensão pelo 464 da linha tal

como foi concebida) resultam da omissão de caracteres (especialmente parênteses), da escrita incorrecta de palavras-chave, ou da transposição inadvertida de caracteres. Os pontos a ter particular atenção são: a ausência de dois pontos entre instruções, o algarismo «1» substituído pela letra «I» ou «l» e o zero pela letra «O». Por exemplo: GOTO I000 em vez de GOTO 1000 dará origem a um erro de sintaxe.

4) O facto de se esquadrinhar a linha pode não conduzir à revelação do erro. Neste caso, a próxima tentativa depende de querermos, ou não, preservar o valor das variáveis em memória. No caso dos erros de sintaxe, o valor das variáveis é irrelevante e podem fazer-se alterações na linha para ajudar a encontrar o erro, mesmo que isto apague a área das variáveis. Se pressupõe que o valor de uma ou mais variáveis está envolvido na questão, reporte-se ao subtítulo «Mensagens de erro não evidentes», que está mais adiante.

5) Se a linha contém mais de uma instrução, isto é, tem dois pontos, desde que não necessite das variáveis, ponha uma instrução de STOP numa linha nova logo a seguir à linha onde foi assinalado erro. Comece agora na última instrução da linha com defeito e insira REM mesmo no início da instrução. Execute a linha outra vez (por vezes terá de executar todo o programa até esse ponto). Se o erro de sintaxe desapareceu, então é a última instrução que está errada, pois foi o REM que a tirou da linha. Se persistir, tire o REM e coloque-o no princípio da instrução anterior da mesma linha. No momento em que atingir a primeira instrução, dessa linha terá identificado a instrução que contém um erro de sintaxe.

MENSAGENS DE ERRO NÃO EVIDENTES

Algumas mensagens de erro não localizam exactamente o erro no programa, dizem apenas onde se encontra a chave para o localizar. A distinção não é absoluta. Se introduzirmos uma linha do tipo:

```
10 T$ = MID$(A$,0)
```

obtemos a mensagem de erro IMPROPER ARGUMENT, e o motivo é fácil de descobrir. Em geral, as mensagens de erro, não in-

cluídas nas atrás indicadas, podem assinalar um erro numa linha, ou linhas, diferentes das indicadas pelo sistema nas mensagens de erro. Na verdade, algumas mensagens de erro, como por exemplo, DIVISION BY ZERO, não dão indicação da linha onde ocorrem.

O procedimento a adoptar, nesses casos, é menos simples do que descobrir um erro numa única linha. Surpreendentemente, nem sempre a utilização da instrução TRON é a melhor, pela razão simples de que utilizar TRON, executando o programa outra vez, apagará a informação sobre o erro que já existia na memória. Para um erro que não assinala uma linha, TRON é mesmo útil, porque é necessário encontrar a linha assinalada pelo mecanismo de pesquisa mesmo antes da mensagem de erro. Quando estão a ser executadas partes de um programa com que não contávamos, o mecanismo de pesquisa pode também ser útil, mas tem de se estar prevenido contra a eventualidade de, ao executar o programa outra vez, o erro não se repetir e perder-se a oportunidade. Para a maioria dos problemas, a utilização do TRON deve ser o último recurso.

Para a maioria dos casos, o primeiro passo será imprimir o valor de todas as variáveis contidas na linha. Assim, se tivermos uma linha como a que se segue:

```
100 A = X*Y/(T1*T2)
```

devemos fazer:

```
? X
? Y
? T1
? T2
```

e tomar nota dos valores encontrados. Agora, progrida na linha mentalmente, ou no papel, tentando descortinar por que é que os valores encontrados deram origem à mensagem de erro que fez parar o programa. Enquanto não identificar o motivo não pode ir mais além na pesquisa do erro. Normalmente isto não trará problemas; mas se, devido à complexidade da linha, não é capaz de ver como as variáveis trabalham em conjunto, poderá eventualmente ter de recorrer à reintrodução de instruções na linha, mas desta vez distribuídas no meio de duas ou três linhas mais curtas e separadas, e depois executar o programa outra vez. Só deverá fazer isto como último recurso, porque, a menos que consiga reproduzir a cadeia

de acontecimentos que conduziram ao erro, este pode não se repetir e guardar-se para outra ocasião.

Quando descobrimos a variável que deu origem ao erro, o único caminho é seguir mentalmente a execução do programa em sentido inverso, para ver onde a variável poderá ter adquirido o valor errado. Infelizmente isto não é possível nos programas complexos. Neste caso, a solução será alterar o programa para que possam ser feitas verificações regulares ao valor da variável quando o programa estiver em execução. Consegue-se isto inserindo novas linhas no programa depois das secções do programa em que a variável possa ser alterada, ou imprimindo as variáveis, ou com STOP para que possa pedir ao sistema que imprima os valores relevantes durante a paragem do programa e introduzir depois CONT para continuar, se esta tentativa não tiver resultado.

No caso de descobrir a área em que se supõe ter origem o problema e as linhas do programa parecerem correctas, um factor a ter em conta é o da eventual duplicação dos nomes das variáveis. As linhas que se referem a uma variável importante podem, de facto, não estar erradas, mas o programa ter ficado sem sentido devido à utilização de uma variável, com o mesmo nome, para outro fim, noutro lugar.

Toda esta tarefa pode ser feita mais simplesmente se forem observadas duas regras:

1) Quando se pretender manusear conjuntos de dados complexos, um dos primeiros módulos a introduzir deverá ser a rotina de armazenamento de dados no ficheiro, em disco ou fita. Quando introduzir dados, para começar a testar, grave-os regularmente; assim, se o programa parar, será capaz de tornar a carregar o último conjunto de dados a partir da fita, em vez de os introduzir de improviso a partir do teclado.

2) A maior parte dos erros tornar-se-ão mais facilmente evidentes com pequena quantidade de dados. Em vez de martelar grandes quantidades de dados, introduza só três ou quatro assuntos, e depois percorra todas as funções do programa. Se existir um erro será fácil simular de novo a sequência se apenas quatro assuntos tiverem entrado, em especial se tiver introduzido assuntos estilizados ou valores como AAAA, BBBB, CCCC, 1111, 2222 e 3333.

Procurar um erro cravado algures num programa pode ser uma tarefa exigente. Só pode ser executada com sucesso se feita cuidadosamente, seguindo pormenorizadamente a execução do

programa, com conhecimento exacto do que se pretende alcançar com cada parte do programa. Quando lhe aparecerem erros como estes ficará muito grato se tiver seguido os conselhos do cap. 1 e escrito o programa em módulos funcionais, dado que essa estrutura torna consideravelmente mais fácil a pesquisa dos erros.

ERROS EM FUNÇÕES

Existe uma categoria de erros totalmente insolúvel se você não tiver consciência de um facto simples — o de que os erros cometidos na definição de funções só aparecerem quando se utilizar essa função. Por exemplo, se de um programa fizerem parte as linhas seguintes:

```
10 DEF FNX = A + T$
100 A = FNX
```

teremos um erro «Type Mismatch», que é incompreensível e só será evidenciado na linha 100, e não na linha 10. Tudo vai bem quando a situação é do tipo deste exemplo, mas numa linha complexa pode não ser tão claro. Sempre que lhe aparecerem erros esquisitos em linhas que contêm funções definidas pelo utilizador, verifique a definição da função através de uma instrução do tipo ? FN X, que originará um erro se a função estiver incorrectamente definida.

INTERPRETAÇÃO DE MENSAGENS DE ERRO

Os erros são tão variados quanto os programadores, pela simples razão de resultarem do seu trabalho. Eis o motivo por que é impossível dar uma lista exaustiva do significado de todas as mensagens de erro possíveis. Indicarei seguidamente uma lista dos erros mais comuns, bem como algumas sugestões quanto à sua origem:

UNEXPECTED NEXT: A razão evidente é ter esquecido uma instrução FOR, mas, na prática, é mais comum ter começado um ciclo com uma variável de controlo, digamos FOR I = ..., e terminar com outra, seja NEXT J. Se tem tendência para utilizar demasiado GOTO, GOSUB ou qualquer outra instrução que conduza ao

salto para outra linha, então pode ter acontecido ter saltado para o interior de um ciclo que contorne o FOR.

SYNTAX ERROR: O intérprete de BASIC encontrou alguma coisa errada na estrutura da linha em questão. Os erros de sintaxe não têm nada a ver com a execução do programa, estão apenas ligados à forma física da linha assinalada.

UNEXPECTED RETURN: O sistema sabe quantas vezes chamou uma dada sub-rotina com GOSUB e só aceitará o correspondente número de RETURN. Muitas vezes, devido ou a um uso demasiado livre de GOTO ou simplesmente por ter todas as sub-rotinas no fim do programa e se ter esquecido de as separar da parte principal do programa com algo parecido com STOP, acontece que o programa quando acaba, volta à primeira sub-rotina.

DATA EXAUSTED: Só se pode ler — READ — tantos assuntos dos dados — DATA — quantos houver no programa, a não ser que queira voltar ao princípio e ler alguns outra vez. Se tentar ler dez assuntos e só existirem nove, apanhará com este erro. O programa necessita saber quantos DATA existem, ou então necessita de colocar um assunto especial no fim de DATA que, quando for lido, indique que chegámos ao fim dos dados.

IMPROPER ARGUMENT: Um dos pedaços de informação extra que necessita de ser especificado, para um comando, não está correcto. Só podemos saber o significado exacto estudando o contexto da instrução e respectiva sintaxe, no manual. De tudo isto pode concluir-se que não será um erro evidente, encapotado por outra mensagem de erro.

1.7E+38

OVERFLOW: O maior número com que podemos trabalhar é aproximadamente 1.7E+38 (1.70141E+38 no meu sistema) ou 17 seguido de 37 zeros. Se em qualquer altura o resultado de um cálculo excede este limite aparece a mensagem OVERFLOW e o resultado não é de confiar. As funções que convertem os números de vírgula flutuante em inteiros têm um limite inferior (32767) antes de ultrapassarem o limite e não dão um resultado com significado.

MEMORY FULL: Não há suficiente espaço na memória disponível para as necessidades do BASIC. Pode acontecer por o seu programa ser comprido de mais, as matrizes serem grandes de mais, ou porque artificialmente se baixou o limite superior da memória utilizável até um nível impraticável.

LINE DOES NOT EXIST: No sistema existe referência a um número de linha que não corresponde a nenhuma linha do programa real.

SUBSCRIPT OUT OF RANGE: Um dos erros que mais usualmente surpreende as pessoas e, no entanto, é muito simples. Significa que nos referimos a um elemento específico de uma matriz em que um dos seus índices (números são usados para definir a posição do elemento) está para além da dimensão da matriz. Pode acontecer porque:

a) A posição do elemento é calculada pelo programa, e este chegou a valores incompatíveis com a dimensão da matriz.
b) A declaração da matriz foi mal feita (isto é, com dois índices trocados).
c) O nome da matriz foi mal escrito quando a declarámos ou quando tentámos aceder a ela.

ARRAY ALREADY DIMENSIONED: Desde que uma matriz esteja dimensionada não se pode redimensionar sem primeiro limpar a memória.

DIVISION BY ZERO: Foi efectuada uma operação que envolve divisão por um número mais pequeno que 1.7E-38.

INVALID DIRECT COMMAND: Algumas instruções não se podem introduzir directamente a partir do teclado.

TYPE MISMATCH: O sistema esperava um número e recebeu uma cadeia, por exemplo: A = T$ ou, vice-versa, t$ = 5.

STRING SPACE FULL: Por existirem demasiadas cadeias com espaço de memória afectado, já não há espaço para mais. É necessário retirar alguma coisa da memória, antes de prosseguirmos.

STRING TOO LONG: O comprimento máximo de uma cadeia é 255 caracteres e o que você está tentar fazer, quer se aperceba disso quer não, é criar uma cadeia de comprimento superior.

STRING EXPRESSION TOO COMPLEX: Quando se utilizam os operadores MID$, LEFT$, RIGHT$ podem criar-se fórmulas muito complexas, tal como:

MID$(LEFT$(RIGHT$(MID$(T$,2),2,)2,)

É uma expressão perfeitamente válida, mas cada nível de parênteses necessita que seja calculada uma cadeia e guardada na memória até atingir o resultado. Chega-se ao ponto de haver demasiado para lembrar ao mesmo tempo e o intérprete de BASIC fica confuso.

CANNOT CONTINUE: Pediu ao programa que continue e ele não pode fazê-lo. A maior parte das vezes acontece porque foi feita uma alteração ao programa enquanto estava parado, ou porque se limpou a memória de qualquer outra maneira.

UNKNOWN USER FUNCTION: Referiu uma função definida pelo utilizador e o programa não tem conhecimento dela. O 464 necessita de executar a linha que contém a definição da função, para depois a reconhecer. Existem outras máquinas em que é suficiente a definição constar do programa para que seja reconhecida em qualquer parte do programa.

RESUME MISSING: Quando se utiliza ON ERROR, o sistema espera que o informem do RESUME (recomeço) antes que o programa termine.

DIRECT COMMAND: O sistema encontrou um grupo de caracteres sem indicação de número de linha num ficheiro em fita ou disco, normalmente devido a corrupção.

OPERAND MISSING: Pediu-se ao sistema que executasse uma instrução que requer a especificação de assuntos extra, mas um ou mais desses assuntos está em falta, por exemplo, WINDOW sem especificação do número de uma corrente *(stream)*.

LINE TOO LONG: As linhas num programa de BASIC não podem exceder os 255 caracteres. Quando se está a armazenar na memória, o programa faz certas conversões que o encurtam. Se, contudo, o resultado final exceder o limite, gera uma mensagem de erro. A única solução é partir a linha; há poucos motivos que justifiquem a utilização de linhas maciças.

EOF MET: O programa está a tentar ler um assunto de ficheiro contido num disco ou fita, quando de facto o ficheiro foi ultrapassado. É similar à mensagem de erro DATA EXHAUSTED:

FILE TYPE ERROR: Cada ficheiro guardado em disco, ou cassete, tem o seu tipo específico, conforme seja do tipo ASCII ou um programa. Este erro significa que o ficheiro que chamou não pertence à designação que lhe atribuiu. Por exemplo, tentou ler um ficheiro programa quando se tratava de dados ASCII.

NEXT MISSING: Trata-se do oposto a UNEXPECTED NEXT, o sistema não foi capaz de encontrar um NEXT relativo a um FOR. As razões são em tudo semelhantes às dadas naquele caso.

FILE ALREADY OPEN: O ficheiro que pretende abrir já foi anteriormente aberto e não foi subsequentemente CLOSED (fechado).

UNKNOWN COMMAND: O sistema não reconhece uma instrução de introdução de dados externa, isto é, uma instrução precedida do símbolo «|».

WEND MISSING: Começou um ciclo WHILE e o sistema não encontra o correspondente WEND. Veja também UNEXPECTED NEXT.

CONCLUSÃO

A caça ao erro com êxito é adquirida com a experiência e não pouco trabalho e raciocínio. Apesar da melhor boa vontade do mundo, existirão sempre alguns erros que o deixarão embasbacado e que poderão levar dias ou semanas a encontrar. Quando isso

acontecer não há melhor que arranjar outro par de olhos para o ajudar. Outro programador descobrirá rapidamente algo que você não viu por estar demasiado perto do programa. Mesmo assim, o momento de chamar alguém só chega quando formos capazes de descrever exactamente o que vai mal, quais os valores das variáveis em questão e a zona do programa que nos apanhou. Saber que determinada linha do programa dá origem a determinado erro não chega, é apenas o começo do processo.

CAPÍTULO 3

Cadeias

As cadeias são uma maneira fácil e flexível de armazenar informação no Amstrad. Utilizando cadeias, a informação pode ser apagada, adicionada ou alterada quase instantaneamente, mesmo no caso de operações muito complexas. A utilização de cadeias pode beneficiar muito a acção dos programadores. Tal facto não é entendido pela maioria deles, não porque seja difícil manusear as cadeias mas porque é um assunto minucioso e as linhas de programação que permitem a utilização de cadeias parecem complexas à primeira vista.

O Amstrad faculta ao utilizador três funções para lidar com cadeias — LEFT$, RIGHT$, MID$. A maior parte dos utilizadores domina razoavelmente estas funções, mas para os que não estão tão familiarizados com elas faço seguidamente uma breve revisão:

LEFT$(A$,10) significa os primeiros dez caracteres de A$.
RIGHT$(A$,10) significa os últimos dez caracteres de A$.
MID$(A$,10) significa a parte de A$ que começa no carácter número 10 e continua até ao fim da cadeia.
MID(A$,10,5) significa a parte de A$ que começa no carácter número 10 e inclui os cinco caracteres seguintes.

Aplicando estas instruções à cadeia A$ que contenha o alfabeto teremos, respectivamente:

1) ABCDEFGHIJ
2) QRSTUVWXYZ
3) JKLMNOPQRSTUVWXYZ
4) JKLMN

Estas instruções são tão simples como o exemplo evidencia. Para muitas pessoas o problema surge quando é necessário aplicá-las em combinação. Quando isto acontece, os parênteses brotam dentro doutros parênteses num grau que faz parecer estas funções muito complexas e obscuras. A maneira de lidar com este problema é começar sempre com a parte da expressão mais interior dos parênteses e começar a decifrá-la, pouco a pouco, por forma a que fique cada vez mais simples. Por exemplo, suponha que tem uma expressão do tipo:

```
MID$(LEFT$(RIGHT$(A$,10),5),3)
```

Que vamos fazer? Suponhamos que A$ é ainda o alfabeto. Começamos pela expressão mais interior, ou seja, RIGHT$ (A$,10); do que ficou dito, RIGHT$(A$,10) é «QRSTUVWXYZ». Substituindo, temos:

```
MID$(LEFT$(«QRSTUVWXYZ»,5),3)
```

Seguindo o mesmo caminho, vemos que LEFT$ significa «QRSTU», o que conduz a:

```
MID$(«QRSTU»,3)
```

ou seja, «STU». Nas expressões utilizando cadeias, como noutro tipo qualquer de expressões, comece de dentro para fora, e verá que o problema se resolve por si.

Neste capítulo veremos várias maneiras de combinar funções de cadeias, umas com as outras e também com outras instruções de BASIC, para ficarmos com uma grande variedade de úteis e interessantes capacidades para programarmos. Vários capítulos que vêm a seguir utilizam as técnicas que descrevemos aqui pormenorizadamente, diferindo apenas nas aplicações; é sensato ter a certeza de que compreendeu os exemplos dados agora, antes de avançar.

ALTERAÇÃO PARCIAL DE CADEIAS

Uma capacidade adicional da instrução MID$ é permitir alterar parte das cadeias. A instrução

MID$(A$,3,2) = «ABC»

aponta os dois caracteres que começam no terceiro carácter das letras «ABC». Se não existirem caracteres suficientes no lado direito da equação para satisfazer a condição expressa, os remanescentes ficam inalterados. Se houver demasiados caracteres para satisfazer a condição indicada, então só os que forem necessários serão utilizados, por isso:

MID$(A$,3,2) = «ABCDE»

produz o mesmo efeito que a instrução anterior.

A capacidade de MID$ atribuir caracteres a uma cadeia é muito importante porque, desde que exista espaço na cadeia para receber os caracteres, será simples inserir caracteres no meio dela sem ter de recorrer ao corte em fatias da cadeia.

CONCATENAÇÃO OU ADIÇÃO DE CADEIAS

Uma das coisas mais simples que se pode fazer com cadeias é adicioná-las:

```
100 A$ = B$ + C$ + D$
```

dá origem a uma nova cadeia feita com as três cadeias indicadas no lado esquerdo colocadas «cabeça a seguir à cauda». Esta habilidade simples é utilizada para muitas vezes originar cadeias com significado, a partir de pedaços mais pequenos de informação. Indicamos seguidamente um exemplo simples:

```
1000 REM*******************************
1001 REM STRING ADDITION
1002 REM*******************************
1010 INPUT "SURNAME:";SN$
1020 INPUT "FIRST NAME:";CN$
1030 INPUT "SEX (M/F):";Q$
1040 SX$ = "MR." : IF Q$ = "F" THEN SX$ = "MS."
1050 NAME$ = SX$ + " " + CN$ + " " + SN$
1060 PRINT NAME$
```

Nem sempre as aplicações são tão triviais. Assuntos independentes podem ser introduzidos numa única cadeia, separados por marcas, como por exemplo « * », entre cada assunto. Como cada cadeia requer três *bytes* extra de memória, estas entradas independentes «empacotadas» permitem poupar memória. Podem ser facilmente acessíveis utilizando uma rotina de busca simples, que será descrita mais adiante. Outros métodos mais complexos e flexíveis constam do capítulo «Estruturas de dados».

SUBTRACÇÃO DE CADEIAS

Tal como podemos adicionar cadeias, também podemos tirar bocados a uma cadeia. A subtracção não pode ser feita tão simplesmente como a adição, porque A$ = B$-C$ não tem significado para o 464. Subtrair, ou tirar, uma cadeia de outra, significa redefinir a cadeia original de forma a excluirmos os caracteres que pretendemos retirar. O método exacto depende precisamente da posição que os caracteres a retirar ocuparem no interior da cadeia principal:

1) Para retirar LL caracteres da parte esquerda de A$, A$ será redefinida como a porção de A$ que vem depois dos primeiros LL caracteres:

```
100 A$ = MID$(A$,LL + 1)
```

2) Para retirar LL letras do fim de A$, A$ será redefinida como contendo todos os caracteres até ao que fica antes do primeiro a ser retirado. Para conseguirmos isto usamos a função LEN, para sabermos o comprimento da cadeia actual e estabelecer que o comprimento da cadeia final será aquele menos o número de caracteres a apagar:

```
100 A$ = LEFT$(A$,LEN(A$)-LL)
```

3) Para retirar LL caracteres do meio de uma cadeia só é necessário saber onde se começa (SP) e o comprimento da porção a retirar. Uma vez na posse desta informação, resta-nos redefinir a cadeia a partir da concatenação das porções antes e depois dos caracteres a apagar:

```
100 A$ = LEFT$(A$,SP-1) + MID$(A$,SP + LL)
```

A lógica desta linha é que se o começo do grupo a ser apagado é no carácter SP, então desejamos conservar todos os caracteres até ao SP-1, inclusive. A cadeia tem o comprimento actual de LL caracteres; deste modo acabará na posição SP (primeiro carácter) + LL (que é o comprimento) menos um. O segundo grupo de caracteres que desejamos conservar começa no carácter LL + SP e continua até ao fim da cadeia. Para exemplificar, vamos subtrair «CDE» da cadeia «ABCDEFG». A posição inicial de «CDE» é o terceiro carácter, e o seu comprimento é três caracteres. A parte que fica da cadeia é até ao SP-1, que dará «AB» e os caracteres após o grupo a apagar começam em SP + LL, o que dá «FG»; adicionando as duas teremos «ABFG».

INSERÇÃO DE ASSUNTOS DENTRO DAS CADEIAS

Comparando o que foi dito sobre adição e subtracção de cadeias, notamos que até agora, embora possamos tirar um grupo de caracteres do interior duma cadeia, não sabemos ainda inserir assuntos nela. Ao remover um assunto do interior de uma cadeia analisámos o método de identificar as duas cadeias remanescentes a reter no resultado. Utilizamos, aproximadamente, o mesmo método para inserir um grupo de caracteres novo. No exemplo que se segue, o objectivo é inserir uma cadeia nova, B$, dentro de A$, sendo o primeiro carácter de B$ o carácter PP da nova cadeia A$:

```
100 A$ = LEFT$(A$,PP-1) + B$ + MID$ (A$,PP)
```

MOVIMENTAR ASSUNTOS DENTRO DAS CADEIAS

Depois de termos examinado como adicionar ou remover assuntos dentro de uma cadeia, estamos agora em condições de combinar as duas técnicas para movimentar assuntos dentro da cadeia. Em resumo, para movimentarmos um assunto no interior duma cadeia procederemos a duas operações: o grupo de caracteres a ser

movimentado deve ser subtraído da cadeia e depois adicionado noutro lugar. O método é exposto no exemplo que segue, numa cadeia A$ que tem um grupo de caracteres de comprimento LL começando na posição SP. O nosso objectivo é mover o grupo para uma posição nova, com início no carácter que está na posição FP. Um exemplo de como isto tudo se pode passar é dado pelo rearranjo que podemos dar à cadeia «ABGHICDEFJKL» movimentando «GHI», que começa na posição três, para uma posição tal que permita a leitura «ABCDEFGHIJKL». A nova posição inicial do grupo, na cadeia final, será na posição sete.

Não é tão simples como parece chegar à posição correcta. Primeiro devemos ignorar que o grupo na sua posição original será apagado, calculando a nova posição inicial na cadeia existente. No caso acima exemplificado, desejamos que o grupo seja reinserido com posição inicial correspondente à actual do carácter «J», ou seja, posição 10. Temos então duas alternativas:

a) Se a nova posição inicial do grupo de caracteres for antes da posição inicial de partida, então o número a que chegamos não precisa de ser ajustado.
b) Se a nova posição inicial é depois do fim do grupo de caracteres, à partida, então o comprimento do grupo deve ser subtraído do número a que chegámos.

No exemplo já dado, o grupo termina na posição cinco inicialmente, e a posição em que pretendemos reinseri-lo é a posição dez. Temos então de subtrair o comprimento do grupo (3), para chegarmos à posição final, que é sete, conforme concluímos anteriormente com base no senso comum.

Sistematizando, tal como deveríamos ter aplicado no exemplo acima dado, chegamos a algo parecido com o seguinte:

```
50 A$ = "ABGHICDEFJKL"
60 FP = 10
70 SP = 3
80 LL = 3
2000 REM*******************************
2001 REM MOVE CHARACTER GROUP
2002 REM*******************************
2010 IF FP >|(SP + LL-1) THEN FP = FP-LL
2020 TT$ = MID$(A$,SP,LL)
```

```
2030 A$ = LEFT$(A$,SP-1) + MID$(A$,SP + LL)
2040 A$ = LEFT$(A$,FP-1) + TT$ + MID$(A$,FP)
2050 PRINT A$
```

VARIÁVEIS:

A$ — Cadeia em que trabalhamos.
FP — Ponto em que o grupo começaria se fosse reinserido sem primeiro ser apagado.
LI — Comprimento do grupo a ser movimentado.
SP — Posição inicial de partida do grupo a ser movimentado.
TT$ — Armazenamento temporário para o grupo de caracteres a ser movimentado.

As linhas que começam em 2000 podem ser utilizadas para movimentar qualquer grupo de caracteres dentro da cadeia principal, desde que sejam conhecidas a posição inicial (SP), a posição final (FP) e o comprimento (LL).

BUSCA DENTRO DAS CADEIAS

Em tudo o que foi dito, até agora, sobre a manipulação de cadeias, tivemos em conta que o programador já possuía toda a informação necessária para especificar os pontos iniciais e finais de todas as partes das cadeias a movimentar. Normalmente isto não acontece, porque não é o programador, mas sim o programa, que determina onde são feitas as alterações, de acordo com as linhas-mestras nele contidas. Muitas vezes, a maneira como o programa determina como lidar com uma dada cadeia está contida na própria cadeia.

No princípio deste capítulo examinámos um exemplo da adição de cadeias usando partes de nomes. Cada uma das partes estava armazenada numa cadeia em separado, e originava uma cadeia mais comprida, da forma «MS. JANE SMITH». Foi fácil fazer isto — mas... e o contrário, isto é, extrair as partes do todo? Obter «MS.» ou «MR.» será fácil, já que ambos têm o mesmo comprimento, mas nesta altura podemos encontrar um ou dois

«REV.», ou outros títulos que não tenham comprimento normalizado. Mesmo que conseguíssemos tirar facilmente o título do nome, o nome próprio seria já imprevisível no comprimento. Como podemos então desmontar o nome?

A resposta é: toda a informação necessária para separar o nome está na forma dos espaços que separam os três assuntos. São os espaços que usamos mentalmente, quando lemos o nome que o programa pode utilizar, desde que contenha também um método de busca, em relação à cadeia total, da posição do grupo de caracteres que pretendemos. O método mais simples de busca é o que utiliza a instrução INSTR e uma rotina para descobrir uma combinação particular de caracteres, TARGET$, que toma o aspecto seguinte:

```
100 position = INSTR (1,a$,target$)
```

Neste caso a rotina identificará a posição inicial da primeira ocorrência de TARGET$ (o grupo de caracteres que procuramos) na cadeia principal e armazena-o na variável POSITION. Utilizando esta técnica podemos rapidamente conceber uma rotina que «desempacote» qualquer cadeia que contenha várias unidades de informação. No exemplo dado, são as três partes do nome que tem a forma «MR. JOHN BROWN»:

```
5000 REM*********************************
5010 REM UNPACK PARTS OF A STRING
5020 REM*********************************
5030 DIM part$(50)
5040 target$=" "
5050 CLS
5060 INPUT "String with spaces: ";name$
5070 TT=LEN(name$)-LEN(target$)+1
5080 start=1
5090 items=0
5100 WHILE start<=LEN(name$)
5110 finish=INSTR(start,name$,target$)
5120 IF finish=0 THEN finish=LEN(name$)+
1
5130 part$(items)=MID$(name$,start,finis
h-start)
5140 start=finish+1
5150 items=items+1
```

```
5160 WEND
5170 :
5180 FOR i=0 TO items-1
5190 PRINT part$(i)
5200 NEXT i
5210 :
5220 STOP
5230 :
```

VARIÁVEIS:

ITEMS — Número de assuntos descoberto em NAME$
PART$ — Matriz usada para reter os assuntos descobertos em NAME$.
NAME$ — Cadeia principal em que se realiza a busca.
START — Início da secção de NAME$ em que se realiza a busca.
FINISH — Posição do carácter a seguir a um assunto em NAME$.
TARGET$ — Carácter usado como separador entre assuntos.

Neste caso a técnica consiste em começar a busca pela cadeia separadora no carácter número um da cadeia principal, e, todas as vezes que a cadeia separadora é encontrada, colocar a parte da cadeia principal, desde START até à cadeia separadora, na matriz PART$. A busca é então recomeçada no carácter logo a seguir à cadeia separadora. A rotina considera que a cadeia principal não termina com a cadeia separadora.

Esta técnica de busca pode ser utilizada numa grande variedade de aplicações. A informação pode ser arrumada em cadeias da forma que se viu na secção de adição de cadeias e depois utilizada uma rotina de busca para extrair de novo as várias partes. Em programas inteligentes, que analisam o fraseado das instruções que lhe são dadas, o método pode ser utilizado para detectar a presença de certos verbos ou substantivos, uma técnica que é largamente utilizada, por exemplo, nos jogos de aventuras. Noutros casos, a técnica pode ser usada para detectar entradas em programas que contenham determinada palavra.

ESTRUTURAS REGULARES DE CADEIAS

Dado que o manusear de cadeias pode ser executado através de instruções simples, que podem inserir ou apagar porções sem preocupações de movimentar o conteúdo existente, as cadeias são um meio ideal de armazenar assuntos de comprimento regular em que é necessário fazer inserções e apagamentos com regularidade. Uma única cadeia com comprimento de 255 caracteres pode ser utilizada para reter 63 assuntos de quatro caracteres, 50 de cinco caracteres, e assim sucessivamente.

Se o mais importante é ter um lugar certo, a cadeia deve ser aferida para o comprimento total necessário para reter todos os assuntos. A melhor maneira de conseguir isto é usar um ciclo para construir a cadeia, carácter a carácter. Na rotina seguinte, a cadeia é aferida para uma capacidade de armazenar assuntos de quatro caracteres, cuja posição pode ser especificada pelo utilizador:

```
6000 REM*********************************
6010 REM STRING WITH SAME SIZE ITEMS
6020 REM*********************************
6030 STORAGE$=STRING$(252," ")
6040 more$="y"
6050 WHILE more$<>"n"
6060 CLS
6070 INPUT "ENTER FOUR CHARACTER ITEM: "
;t$
6080 INPUT "ENTER POSITION (1-63): ";t
6090 MID$(storage$,4*t-3,4)=t$
6100 PRINT storage$
6110 PRINT : INPUT "More (y/n): ";more$
6120 more$=LOWER$(more$)
6130 WEND
6140 :
```

Os dados podem ser retirados facilmente através de uma localização numerada. Junte as linhas seguintes à rotina, introduza alguns dados e depois responda «n» quando ela lhe perguntar se deseja introduzir mais.

```
6150 REM*******************************
6160 REM RETRIEVE ITEM
6170 REM*******************************
6180 more$="y"
6190 WHILE more$<>"n"
6200 PRINT : PRINT
6210 INPUT "NUMBER OF ITEM TO PRINT: ";t
6220 PRINT MID$(storage$,t*4-3,4)
6230 INPUT "More (y/n): ";more$
6240 more$=LOWER$(more$)
6250 WEND
6260 :
6270 STOP
6280 :
```

Os assuntos podem ser apagados muito simplesmente redefinindo a sua posição como quatro espaços. Estas rotinas não são muito «robustas»; queremos dizer com isto que podem ser facilmente esmagadas. Isto, no entanto, pode ser ultrapassado recorrendo a uma simples verificação de erros, como se descreve no próximo capítulo.

MATRIZES DE CADEIAS DE ELEMENTOS MÚLTIPLOS

Um dos problemas que surgem com a utilização de cadeias para armazenamento de assuntos de tamanho regular é o de o seu comprimento máximo ser fixo. No exemplo dado atrás, o número máximo de assuntos que podiam ser guardados era 63. Se bem que isto possa ser útil numa grande variedade de aplicações, outras haverá em que será necessária uma capacidade maior. Isto pode conseguir-se com bastante facilidade declarando uma matriz de cadeias e calculando a posição de um assunto, não só em termos da sua localização na cadeia mas também em relação à matriz como um todo. Damos a seguir uma adaptação da rotina anterior que resolve este problema em relação a uma matriz de 20 elementos. Des-

te modo poderemos guardar e aceder a 1260 assuntos de quatro caracteres:

```
7000 REM*********************************
7010 REM MULTIPLE ELEMENT ARRAY
7020 REM*********************************
7030 DIM storage$(19)
7040 FOR i=0 TO 19
7050 storage$(i)=STRING$(252," ")
7060 NEXT i
7070 :
7080 more$="y"
7090 WHILE more$<>"n"
7100 CLS
7110 INPUT "ENTER FOUR CHARACTER ITEM: "
;t$
7120 INPUT "POSITION FOR ITEM: ";t
7130 element=INT((t-1)/63)
7140 t=t-63*element
7150 MID$(storage$(element),t*4-3,4)=t$
7160 INPUT "More (y/n): ";more$
7170 more$=LOWER$(more$)
7180 WEND
7190 :


7200 REM*********************************
7210 REM RETRIEVE ITEMS
7220 REM*********************************
7230 more$="y"
7240 WHILE more$<>"n"
7250 PRINT : PRINT
7260 INPUT "ENTER ITEM NUMBER TO EXAMINE
: ";t
7270 element=INT((t-1)/63)
7280 t=t-63*element
7290 PRINT MID$(storage$(element),t*4-3,
4)
7300 INPUT "More (y/n): ";more$
```

```
7310 more$=LOWER$(more$)
7320 WEND
7330 :
7340 :
7350 STOP
7360 :
```

VARIÁVEIS:

ELEMENT — O número de linha em que o novo assunto deve ser colocado, obtido pela divisão de posição desejada pelo comprimento das linhas na matriz.

T — Número original de assuntos a inserir ou examinar, mas transformados na posição de assuntos na linha ELEMENT.

ARMAZENAMENTO DE DADOS EM CADEIAS DE COMPRIMENTO VARIÁVEL

Nas duas secções anteriores supusemos que as cadeias a trabalhar eram de comprimento fixo. Isto tem a vantagem de a posição dos assuntos nas cadeias ser fixa. Há sempre 63 assuntos, e um assunto colocado na posição 23 ficará na posição 23, aconteça o que acontecer às outras posições. No entanto, nem todas as aplicações necessitam do comprimento total da cadeia em memória. Muitas vezes necessitamos de guardar uma lista compacta de assuntos, numa determinada ordem ou não, que deverão ser sempre corridos um a um, podendo ser adicionado, ou apagado, qualquer assunto. Seguidamente damos uma rotina que permite adicionar assuntos de quatro caracteres ao topo de uma lista de 1240, com a possibilidade de os assuntos poderem ser apagados ou procurados.

```
8000 REM*********************************
8010 REM VARIABLE LENGTH STRING ARRAYS
8020 REM*********************************
8030 DIM storage$(19)
8040 items=126
8050 storage$(0)=STRING$(244,"a")
```

```
8060 storage$(1)=STRING$(244,"b")
8070 storage$(0)=storage$(0)+"xxxx"
8080 storage$(1)=storage$(1)+"zzzz"
8090 CLS
8100 choice=0
8110 WHILE choice<>4
8120 PRINT : PRINT
8130 INPUT "1=INSERT/2=FIND/3=DELETE/4=S
TOP: ";choice
8140 ON choice GOSUB 8180,8340,8480
8150 WEND
8160 END
8170 :
8180 REM*********************************
8190 REM INSERT
8200 REM*********************************
8210 IF ITEMS=1239 THEN PRINT "NO ROOM"
: RETURN
8220 INPUT "FOUR CHARACTER ITEM TO INSER
T: ";item$
8230 items=items+1
8240 storage$(0)=item$+storage$(0)
8250 element=0
8260 WHILE LEN(storage$(element))>=252 A
ND element<19
8270 temp$=RIGHT$(storage$(element),4)
8280 storage$(element)=LEFT$(storage$(el
ement),LEN(storage$(element))-4)
8290 storage$(element+1)=temp$+storage$(
element+1)
8300 element=element+1
8310 WEND
8320 RETURN
8330 :
8340 REM*********************************
8350 REM SEARCH
8360 REM*********************************
8370 INPUT "FOUR CHARACTER TO SEARCH FOR
: ";search$
```

```
8380 FOR i=0 TO INT(items/62)
8390 place=1
8400 WHILE place>0
8410 place=INSTR(place,storage$(i),searc
h$)
8420 IF place>0 AND (place-1)/4=INT((pla
ce-1)/4) THEN PRINT 62*i+INT(place/4)+1;
" : ";MID$(storage$(i),place,4)
8430 IF place>0 THEN place=place+4
8440 WEND
8450 NEXT i
8460 RETURN
8470 :
8480 REM*********************************
8490 REM DELETE
8500 REM*********************************
8510 INPUT "POSITION TO DELETE: ";number
8520 IF number>items THEN RETURN
8530 element=INT((number-1)/62)
8540 place=4*(number-62*element)-3
8550 storage$(element)=LEFT$(storage$(el
ement),place-1)+MID$(storage$(element),p
lace+4)
8560 items=items-1
8570 :
8580 FOR i=0 TO 18
8590 done=0 : WHILE LEN(storage$(i))<248
 AND done=0
8600 size=248-LEN(storage$(i))
8610 storage$(i)=storage$(i)+LEFT$(stora
ge$(i+1),size)
8620 storage$(i+1)=MID$(storage$(i+1),si
ze+1)
8630 done=1 : WEND
8640 NEXT i
8650 RETURN
```

VARIÁVEIS USADAS:

CHOICE — Número da função escolhida pelo utilizador.
ITEMS — Número de assuntos de 4 caracteres da matriz.
TEMP$ — Variável temporária usada para transferir um assunto para a linha seguinte quando o comprimento da linha na matriz atinge 252 — isto é, não poderia ser adicionado outro assunto sem ser excedido o comprimento máximo de 255.
ELEMENT — A linha com que estamos a lidar na matriz.
PLACE — Posição do assunto na linha LL da matriz.
SIZE — Comprimento do assunto a retirar da matriz quando estamos a apagar.

Estude bem esta rotina porque, ainda que pareça um pouco duvidosa de início, há qualquer coisa de novo nela. Tudo o que fizemos foi aplicar algumas das técnicas descritas neste capítulo. A novidade é a maneira como os assuntos são transferidos de cadeia em cadeia sempre que o comprimento da cadeia atinge 252. Faz-se isto para que o novo assunto possa sempre ser adicionado sem perigo ao princípio da matriz. O procedimento inverso é usado quando se apaga.

COLECÇÃO DE LIXO

Quando se utilizam rotinas complexas de cadeias que envolvem grande movimentação de cadeias em matrizes, ou redefinição de cadeias na memória, pode notar-se que o Amstrad parece parar. Disse *parece* — mas, de facto, é o que acontece. Quando redifine uma cadeia, o Amstrad não recompõe a memória para optimizar o espaço disponível: as cadeias apenas são movimentadas na memória quando é absolutamente indispensável fazer espaço para algo que aumentou de comprimento. Significa que a redefinição de uma cadeia tornando-se menor não liberta memória na zona da memória afecta à cadeia. O resultado é que quando se manuseiam grandes quantidades de cadeias, a memória se vai enchendo gradualmente e ao fim de pouco tempo o problema torna-se agudo. O

Amstrad fica então embaraçado naquilo que é habitual chamar uma «colecção de lixo», que mais não é do que um enchimento da memória utilizável pelas cadeias.

Pode acabar-se com a colecção de lixo em qualquer altura com a função FRE, a qual retira a colecção de lixo antes de libertar a memória correspondente. Então se fizermos:

```
PRINT FRE (0)
```

(o valor entre parênteses não faz diferença nenhuma), será impresso o número de *bytes* livres da memória. Se descobrir que a colecção de lixo num programa se torna um problema, pode usar FRE atempadamente, sempre que deseje aumentar a velocidade do programa.

CONCLUSÃO

Quando conseguir pôr em prática as técnicas descritas neste capítulo não terá de entrar de novo em desespero quando se lhe depararem programas que utilizem maciçamente técnicas de manuseamento de cadeias repletos de funções LEFT$, MID$, RIGHT$ ligadas por muitas variáveis. Basicamente, as técnicas de manuseamento de cadeias não fazem mais do que identificar uma parte da cadeia, e, com a utilização inteligente de variáveis, pouco haverá que não possa ser feito com cadeias. Abrem-se grandes possibilidades no armazenamento de dados com eficiência e permite-se que o programador escreva programas que darão ao utilizador a possibilidade de introduzir cadeias de forma mais compreensível, deixando ao programa a tarefa de identificar as partes mais importantes de tudo o que entrou. Tal como dissemos no início do capítulo, as técnicas agora descritas serão utilizadas frequentemente nos capítulos seguintes; não passe, pois, em branco o que aqui ficou dito. Será largamente recompensado do esforço que fizer agora para compreender completamente este capítulo.

CAPÍTULO 4

Introdução de informação

Uma das maiores diferenças entre os microcomputadores actuais e os poderosos computadores do passado reside no facto de os micros modernos serem interactivos, responderem imediatamente ao utilizador, e permitirem que se trabalhe com o programa à medida que é executado. Antes de aparecerem os micros, as pessoas lidavam com máquinas a que era necessário fornecer tanto os programas como os dados todos, sem que fossem feitos quaisquer testes do que era introduzido. Depois, o programa era executado. A maior parte das vezes os resultados só eram conhecidos ao fim de várias horas, talvez até no dia seguinte, e os erros do programa poderiam significar a repetição de todo o processo, várias vezes, antes que o utilizador pudesse ter uma pequena ideia do que o programa faria quando estivesse isento de erros.

O resultado era que os programadores, pelo menos os de sucesso, deviam antecipar tudo o que ia acontecer no decorrer da execução do programa. Se existissem vários processos diferentes durante a execução do programa, deviam lá estar todos, na ordem devida e com todos os dados necessários, antes que o programa fosse executado. Se fosse necessário tomar decisões durante a execução do programa, tinham de ser antecipadas, pois não havia maneira de fazer que o programa se reportasse ao utilizador para uma decisão. Se fosse esquecida uma decisão, então o programa teria de ser executado outra vez no dia seguinte.

O micro moderno alterou completamente esta situação. Algumas aplicações actuais dos computadores, como por exemplo os jogos, não seriam possíveis de executar em máquinas que não permitissem ao programa reportar-se ao utilizador à medida que é exe-

cutado. Ainda mais importante é o facto de os utilizadores de aplicações sérias tomarem por certo que, à medida que o programa é executado, podem introduzir informação, tomar decisões sobre as operações a efectuar, sempre com o controlo do programa nas mãos. Esta liberdade de acção traz consigo alguns problemas. Embora fosse sempre uma tarefa difícil, a programação não interactiva fazia que as pessoas tivessem cuidado com o programa e com os dados destinados ao seu funcionamento. A porgramação interactiva tornou-nos descuidados. Como os resultados dos programas estão presentes quase imediatamente, se alguma coisa estiver errada pode rapidamente ser corrigida e executar-se o programa outra vez.

Hoje, um bom programa já não é aquele em que todos os assuntos dos dados são soletrados de uma maneira que não provoque soluços ao programa e em que todas as decisões são antecipadas. O que se pretende é que permita ao utilizador a introdução de informação de maneira flexível e que se reporte ao utilizador para as decisões importantes e para o funcionamento do programa. Dada a facilidade com que podem ser cometidos erros durante tal processo, um bom programa é também o que garante, ao utilizador, que não serão inadvertidamente introduzidos materiais que causem a paragem do programa, ou a corrupção da informação armazenada.

Trataremos neste capítulo de algumas maneiras pelas quais um programa pode aceitar informação. No próximo capítulo examinaremos alguns métodos que protegem o programa contra erros da informação.

INTRODUÇÃO DE INFORMAÇÃO: INPUT

Durante a execução de um programa o Amstrad fornece quatro modos de introduzir a informação: INPUT, LINE INPUT, INKEY e INKEY$. Qualquer dos quatro tem os seus pontos fortes, embora a maioria dos programas caseiros se apoie quase exclusivamente em INPUT, mesmo nos casos em que seria muito mais apropriado o uso de outra alternativa.

A principal vantagem de INPUT é a de ser clara. Aparece o cursor no visor e podem ser introduzidos números e letras logo visualizados. Quase tão importante é a possibilidade de o que está a ser introduzido poder ser emendado, usando as setas do cursor em combinação com a tecla de apagar. Quando o utilizador estiver sa-

tisfeito com o que vê no visor, termina a instrução INPUT premindo ENTER.

INPUT tem, contudo, algumas desvantagens. A principal é que não pode lidar com vírgulas, por motivos que analisaremos seguidamente. Em muitos programas, o utilizador desejará ser capaz de obter uma resposta premindo apenas uma tecla; isso também não é possível com INPUT, porque tudo tem de terminar com ENTER, o que pode tornar-se cansativo em certas circunstâncias. Finalmente, INPUT trava o sistema até que ENTER seja premida, de tal maneira que não é possível ao programa controlar a maneira como o material é introduzido. Por exemplo, não é possível dar qualquer aviso sobre o comprimento de uma cadeia à medida que está a ser introduzida. Apesar destes inconvenientes, INPUT permanece a instrução mais popular nos programas que aceitam informação enquanto estão a ser executados.

ENTRADAS SIMPLES COM INPUT

Eis algumas maneiras simples de utilizar INPUT:
Introdução de uma cadeia simples:

```
10 INPUT «INTRODUZIR UMA CADEIA»; A$
```

Introdução de um único número:

```
10 INPUT «INTRODUZIR UM NUMERO»; A
```

Introdução de várias cadeias:

```
10 INPUT «APELIDO, NOME PROPRIO E SEXO (SEPARADOS POR VIRGULAS)»; SN$, CN$, SX$
```

Visor: APELIDO, NOME PROPRIO E SEXO (SEPARADO POR VIRGULAS)? LAWRENCE, DAVID, MASCULINO <RETURN>

Repare como neste caso são utilizadas as vírgulas para identificar as três cadeias independentes que a instrução INPUT aguarda. Esta é a razão pela qual a instrução INPUT não pode lidar com vír-

gulas. De facto, para a instrução INPUT as vírgulas são separadores entre assuntos. Se tentar introduzir uma cadeia contendo vírgulas quando INPUT só aguarda uma cadeia única, ser-lhe-á pedida a reintrodução do assunto.

Introdução de vários números:

```
10 INPUT «INTRODUZA O VALOR DOS ASSUNTOS 1-3 (SEPARADOS POR VIRGULAS)»;A,B,C
```

Esta porta-se da mesma maneira que a introdução da cadeia anterior. Números e cadeias podem ser misturados na mesma linha de *input* (introdução de dados).

INPUT DE VÁRIOS ASSUNTOS UTILIZANDO A MESMA LINHA DE VISOR

O manuseamento flexível do visor, permitido pelo Amstrad, combinado com a instrução INPUT, pode ser usado para reduzir o intervalo que muitas vezes estraga a aparência do visor. Quando, por exemplo, têm de ser feitas várias entradas em sucessão, e não seja essencial que o utilizador as veja todas de uma vez, é trivial arranjar todas as entradas de forma a caírem na mesma linha, cada uma escrita sobre a anterior:

```
10 LOCATE 1,1
20 PRINT SPACE$(40);
30 LOCATE 1,1
40 INPUT «ASSUNTO 1»;A1$
50 LOCATE 1,1
60 PRINT SPACE$(40);
70 LOCATE 1,1
80 INPUT «ASSUNTO 2»;A2$
```

Este exemplo usa e abusa de LOCATE para garantir que cada INPUT é feito na mesma linha e que a linha é limpa antes do INPUT seguinte.

Um método melhor seria utilizar uma sub-rotina:

```
10 X = 1 : Y = 1 : PROMPT$ = «INTRODUZA O NOME:»
20 A1$ = TEMP$
30 X = 1 : Y = 1 : PROMPT$ = «INTRODUZA ENDEREÇO:»
40 A2$ = TEMP$
:
:
100 LOCATE X,Y
110 PRINT SPACE$(40);
120 LOCATE X,Y
130 PRINT PROMPT$;
140 INPUT «», TEMP$
150 RETURN
```

Agora só é preciso chamar a sub-rotina para cada *input,* especificar as coordenadas de X e Y no visor e transferir TEMP$, introduzida pela sub-rotina, pela variável adequada. Isto torna-se mais comprido para um ou dois indicadores deste programa; mas, nos que tiverem muitas introduções de dados, pode poupar-se muito espaço. Note-se que, para conseguir isto, temos de escrever os indicadores como um assunto separado (linha 130), porque, ao escrevermos o indicador como parte integral de um INPUT, por exemplo, INPUT «ENDEREÇO:»,TEMP$, o indicador deve ser escrito, não pode ser uma variável de cadeia.

COMO FIXAR O COMPRIMENTO DE UMA CADEIA A INTRODUZIR

Dado que o sistema fica suspenso durante a introdução de dados por INPUT, até que se prima ENTER, não há processo de limitar o comprimento dos assuntos à medida que são introduzidos. É uma limitação importante, já que em muitas aplicações se trabalha com base em assuntos de comprimento fixo, ou, pelo menos, de comprimento máximo, para manipulação ou armazenamento. O que *pode* ser feito, apesar de tudo, é dar alguma orientação no que respeita ao comprimento do assunto e verificá-lo logo que ENTER seja premido. A pequena rotina seguinte ilustra como as linhas coloridas podem ser utilizadas acima e abaixo do *input* para indicar

o comprimento. Mostra também como pode ser feita uma verificação para garantir que não é demasiadamente comprida. Durante a utilização só aceitará cadeias que não excedam o comprimento especificado para a variável MAXLEN. Com pequenas alterações, a rotina pode ser utilizada, tal como no exemplo anterior, como sub-rotina.

```
1000 REM*********************************
1010 REM GUIDE TO INPUT LENGTH
1020 REM*********************************
1030 PAPER 0
1040 prompt$="Input here: "
1050 maxlen=20
1060 done=0
1070 WHILE LEN(t$)>maxlen OR done=0
1080 CLS
1090 PAPER 3
1100 PRINT SPACE$(LEN(prompt$)+maxlen)
1110 PAPER 0
1120 PRINT SPACE$(LEN(prompt$)+maxlen)
1130 PAPER 3
1140 PRINT SPACE$(LEN(prompt$)+maxlen)
1150 LOCATE 1,2
1160 PAPER 0
1170 PRINT prompt$;
1180 INPUT "",t$
1190 done=1
1200 WEND
1210 LOCATE 1,10
1220 :
```

Pode evitar-se a maior parte da dificuldade existente no posicionamento dos assuntos através do uso judicioso de janelas no visor. As janelas de uma linha são simples de limpar e de imprimir, sem necessidade de especificar a sua localização, uma vez definidas. As aplicações comerciais reservam, muitas vezes, a primeira ou a última linha do visor para a introdução a partir do utilizador, simplificando-se assim a tarefa de formatação do visor. A pequena rotina que apresento seguidamente realiza uma janela de uma linha, no cimo do visor, na forma de WINDOW # 1. Depois de de-

clarada, a janela pode ser limpa com CLS # 1 e enviar para lá uma instrução de introdução de dados através de INPUT # 1.

```
2000 REM*********************************
2010 REM INPUT TO A WINDOW
2020 REM*********************************
2030 MODE 1
2040 WINDOW #0,1,40,2,25
2050 WINDOW #1,1,40,1,1
2060 PAPER #1,3
2070 CLS #1
2080 INPUT #1,"Input here: ",t$
2090 MODE 1
2100 :
```

É gratificante o trabalho de experimentar as diversas maneiras de apresentação dos seus INPUT no visor. Um visor claramente formatado torna qualquer programa muito mais fácil de utilizar e reduz os erros por falta de cuidado que podem ser feitos na introdução de dados.

ALTERAÇÃO DE UMA CADEIA EXISTENTE COM INPUT

Em comparação com outros micros modernos, o Amstrad tem uma ligeira desvantagem na utilização da instrução de INPUT: não está adaptada a que o utilizador possa confirmar o conteúdo de uma cadeia premindo simplesmente ENTER, nem permite a correcção do conteúdo de uma cadeia pela colocação de um indicador de INPUT antes da cadeia impressa no visor. Felizmente, o sistema de alteração do visor do Amstrad permite ultrapassar isto, tal como se mostra no útil exemplo seguinte:

```
3000 REM*********************************
3010 REM INPUT AND AN EXISTING STRING
3020 REM*********************************
3030 in$="This is a test string to demon
strate how an existing string can be alt
ered using the input command."
```

```
3040 CLS : LOCATE 1,10
3050 PRINT "STRING TO BE EDITED:-" : PRI
NT
3060 PRINT in$
3070 LOCATE 1,2
3080 INPUT "Input here: ",in$
3090 CLS
3100 PRINT "New string is:-" : PRINT
3110 PRINT in$
3120 LOCATE 1,10
3130 PRINT "Press any key to continue."
3140 t$=INKEY$ : IF t$="" THEN 3140
3150 :
```

Neste caso, a versão existente da cadeia IN$ é impressa a meio caminho da parte de baixo do visor, e o INPUT no cimo. O utilizador tem assim a liberdade de utilizar as teclas de fazer o cursor SHIFT para mover o «cursor de cópia», para baixo, para o começo da versão existente da cadeia. A tecla de COPY pode agora ser utilizada para copiar a cadeia existente até à posição a seguir ao INPUT, onde poderá ser *edited* (alterada) da maneira usual, de preferência a reintroduzi-la na totalidade.

LINE INPUT

Na prática, LINE INPUT comporta-se como INPUT, com a excepção de aceitar cadeias que contenham separadores, como, por exemplo, vírgulas. Com excepção dos exemplos anteriores, que necessitam de introduzir assuntos múltiplos a partir de uma única entrada, todos os outros trabalham do mesmo modo com LINE INPUT.

INKEY$

Sem negar a utilidade de INPUT, existem no entanto muitas ocasiões em que as suas limitações são evidentes. Para ultrapassar isto, o BASIC do Amstrad possui outra instrução: INKEY$. É

mais difícil de utilizar do que INPUT, porque a sua única função é ler o teclado, ou seja, detectar quando é que uma determinada tecla foi ou não premida. INKEY$ é, por isso, particularmente apropriada quando se quer controlar o comprimento da entrada a introduzir ou quando for desejável instruções de uma única tecla, sem recurso a RETURN, ou ainda se não desejamos que o programa espere por uma entrada, se o utilizador não estiver a premir uma tecla.

INKEY$ para originar um estado de espera

Uma utilização clássica de INKEY$ é fornecer um estado de espera até que qualquer tecla seja premida:

```
10 A$ = INKEY$ : IF A$ = «»THEN 10
```

Ao deparar com esta linha, o programa executará INKEY$, e descobre quando qualquer tecla está a ser premida. Se nada acontecer, a cadeia A$ tomará o valor nulo e o IF da segunda parte da linha provocará a repetição da linha. O programa esperará, então, indefinidamente, até que uma tecla seja premida. Nesse momento, A$ tomará o valor da tecla que se premiu e o programa passará à linha seguinte.

As aplicações que dão ao utilizador a possibilidade de escolha em diferentes fases do programa fazem muitas vezes uso deste tipo de INKEY$. Quando um grande número de escolhas tem de ser feito, é mais fácil premir uma única tecla do que constantemente recorrer a RETURN depois de cada entrada. Damos a seguir um menu típico de um programa que utiliza INKEY$:

```
1000 IN$ = «» : WHILE IN$ < > «0»
1010 PRINT «0 = QUIT PROGRAM»
1020 PRINT «1 = ENTER NEW DATA»
1030 PRINT «2 = DELETE ITEMS»
1040 PRINT «3 = ALTER ITEMS»
1050 PRINT «4 = DISPLAY DATA»
1060 PRINT : PRINT «WHICH DO YOU REQUIRE (0-4)?»;
1070 IN$ = INKEY$: IF IN$ = «» THEN 1070
1080 TT = VAL(IN$)
1090 ON TT GOSUB 1000, 2000, 3000, 4000, 5000
1100 WEND
```

Neste caso, tudo o que é necessário é que o utilizador prima as teclas 0, 1, 2, 3 ou 4 para chamar a parte respectiva do programa.

Instruções de uma tecla com INKEY$

A utilização de instruções de tecla única não é restrita aos menus. Existem muitas aplicações projectadas para permitir que em pontos particulares se utilizem instruções de tecla única. Um meio económico de conseguir isto é a utilização do «menu cadeia», tal como consta da rotina seguinte:

```
7000 REM*********************************
7010 REM MENU STRING
7020 REM*********************************
7030 menu$="qwertyuiop"
7040 command=0
7050 CLS : WHILE command<>10
7060 PRINT : PRINT
7070 PRINT "Input single key command: ";
7080 in$=INKEY$ : IF in$="" THEN 7080
7090 command=INSTR(menu$,LOWER$(in$))
7100 ON command GOSUB 7130,7140,7150,716
0,7170,7180,7190,7200,7210
7110 WEND
7120 STOP
7130 PRINT 10000 : RETURN
7140 PRINT 11000 : RETURN
7150 PRINT 12000 : RETURN
7160 PRINT 13000 : RETURN
7170 PRINT 14000 : RETURN
7180 PRINT 15000 : RETURN
7190 PRINT 16000 : RETURN
7200 PRINT 17000 : RETURN
7210 PRINT 18000 : RETURN
7220 t$=INKEY$ : IF t$="" THEN 7220
7230 PRINT ASC(t$)
7240 GOTO 7220
```

A primeira linha de letras do teclado está agora disponível, para o programa, para ser utilizada como instrução; «P» dá por finda a rotina.

COMO MOVER O CURSOR COM INKEY$

Muitas vezes, as aplicações utilizam INKEY$ quando é necessário movimentar o cursor, no visor, por cima de listas de assuntos:

```
4000 REM*********************************
4010 REM MOVING CURSOR
4020 REM*********************************
4030 x=1 : y=1
4040 CLS
4050 FOR i=1 TO 20
4060 LOCATE 1-20*(i>10), i+10*(i>10)
4070 PRINT "  Command ";STR$(i)
4080 NEXT i
4090 t$=""
4100 WHILE LOWER$(t$)<>"s"
4110 t$=""
4120 WHILE t$=""
4130 t$=INKEY$
4140 LOCATE x,y
4150 PRINT "*"
4160 FOR i=1 TO 50 : NEXT
4170 LOCATE x,y
4180 PRINT " "
4190 FOR i=1 TO 50 : NEXT
4200 WEND
4210 IF t$=CHR$(13) THEN command=y+MAX(0
,x-10) : LOCATE 1,20 : PRINT "Command: "
;command
4220 IF ASC(t$)=240 THEN y=y-1 : IF y<1
THEN y=1
4230 IF ASC(t$)=241 THEN y=y+1 : IF y>10
 THEN y=10
```

```
4240 IF ASC(t$)=242 THEN x=x-19 : IF x<1
 THEN x=1
4250 IF ASC(t$)=243 THEN x=x+19 : IF x>2
0 THEN x=20
4260 WEND
4270 :
```

A rotina estabelece uma lista de instruções no visor, ao lado do cursor a piscar, que pode ser colocado em frente de qualquer delas, com a ajuda das teclas de comando normal do cursor. Premindo ENTER, conseguimos que a variável «COMMAND» seja colocada no valor da instrução adequada. É uma maneira extremamente suave de proporcionar ao utilizador uma escolha variada.

INKEY$ utilizada como resposta diferida

Muitas vezes pode ser útil proporcionar ao utilizador um tempo de reflexão antes da resposta, em vez de parar o programa indefinidamente. Utilizando INKEY$ em conjugação com um ciclo apropriado, isso torna-se uma tarefa simples:

```
5000 REM*********************************
5010 REM TIMED RESPONSE
5020 REM*********************************
5030 CLS
5040 PRINT "You have 10 seconds to press
 a key otherthan ENTER."
5050 AFTER 500 GOSUB 5140
5060 done=0 : t$=""
5070 WHILE t$="" AND done=0
5080 t$=INKEY$
5090 WEND
5100 IF t$<>"" THEN PRINT "Key pressed w
as >";t$;"<"
5110 PRINT : PRINT "Press any key."
5120 t$=INKEY$ : IF t$="" THEN 5120
5130 GOTO 6000
5140 done=1 : RETURN
5150 :
```

Neste caso, AFTER conclui o ciclo alternando o valor de DONE depois do período especificado.

INKEY$ e caixas inversas

Quando, à partida, definimos a localização de dados numa determinada posição, ou seja, utilizamos caixas inversas. Uma maneira interessante consiste no uso de INKEY$, o que pode mesmo ser determinante na solução de problemas deste tipo. De facto, INKEY$ pode detectar o comprimento de uma entrada à medida que se está a introduzir. A rotina seguinte aceita qualquer entrada até 10 caracteres, de comprimento, inclusive, mas não permitirá que além desse comprimento se introduza qualquer coisa:

```
6000 REM*********************************
6010 REM ENTRY TO INVERSE BOX
6020 REM*********************************
6030 in$=""
6040 prompt$="Item 1: "
6050 PAPER 0 : PEN 1
6060 CLS
6070 LOCATE 1,2
6080 PRINT prompt$;
6090 PAPER 1 : PEN 0
6100 PRINT SPACE$(10)
```

Neste caso desenhámos a caixa, colocando o indicador na sua frente. A posição de impressão fica LOCATE (localizada) no início da caixa. Podem ser batidas as letras até ao comprimento máximo (10, neste caso). Quando se atingir este máximo, a rotina não consentirá que o utilizador introduza mais caracteres, a não ser que retire primeiro outros.

CONCLUSÃO

A introdução de dados é uma área em que se pode estragar um programa até aí excelente. Quando o visor que vai receber a informação não está adequadamente demarcado, esta poderá ser introduzida erradamente e será cansativo repetir todo o processo. Uma formatação ajustada, utilizando mesmo a cor, por exemplo, com diferentes cores ligadas a indicadores sucessivos, e com cada introdução feita em cores contrastantes, para que o conteúdo sobressaia no visor em relação aos indicadores, tornará o programa totalmente diferente.

O método de formatação é um assunto de gosto pessoal, mas, com as técnicas expostas neste capítulo, não há razão para que os programas feitos por si sejam uma enfadonha lista de indicadores uns atrás dos outros, no visor. Falta-nos ainda saber como podemos garantir que a informação que obtemos está correcta.

CAPÍTULO 5

Caça ao erro

Não há nenhum programa à prova de idiotices. Pode dizer-se, a propósito, que ainda não apareceu um idiota suficientemente criativo. Mesmo assim, não há nada mais aborrecido que termos um programa, muito atractivo e útil, que pára num ponto crucial porque o utilizador fez uso de uma introdução de dados que não pode ser tratada da maneira usual. Neste capítulo examinaremos algumas maneiras de tornar o programa mais «robusto», ou seja, menos sujeito a parar quando qualquer dado insignificante for introduzido. Encontrará poucas técnicas complexas, porque a caça ao erro é quase sempre um assunto de senso comum. Faremos a tentativa de compreender as causas dos erros que a maioria das pessoas comete e de estar sempre um passo à frente, em relação a elas.

EVITAR ERROS — PRECAUÇÕES DO SENSO COMUM

Por que será que as pessoas insistem em cometer erros na utilização dos melhores programas? Existem muitas respostas possíveis, mas a mais provável é que o nosso programa favorito não é tão bom como pensámos. A maioria dos erros cometidos na introdução de dados surgem porque o programa não é suficientemente claro no modo como a informação é solicitada ao utilizador. Pode acontecer que os indicadores que deu às suas instruções de introdução de dados, INPUT, sejam demasiado breves, ou que o visor esteja demasiado cheio, para que o utilizador seja capaz de se concentrar eficazmente em cada introdução de dados. Pode também acontecer que alteremos as convenções de partida depois de percor-

rermos metade do caminho durante a execução do programa, como, por exemplo, à espera, a maior parte do tempo, de uma resposta numérica e subitamente solicitar uma introdução de dados alfabética, sem previamente ter clarificado este ponto. Qualquer que seja a razão, não há motivo para criticar as pessoas que destroem os seus programas; os programas fizeram-se para serem utilizados, e isso só será possível se forem concebidos de tal modo que tornem claros para o utilizador o que se pretende que aconteça minuto a minuto.

Com estas ideias na mente, podemos começar por dois princípios do senso comum que eliminarão a maior parte dos erros:

Arranjo do visor

Dê-se ao cuidado de formatar as introduções de dados de forma adequada, com utilização das técnicas descritas em pormenor no capítulo anterior. A disposição dos assuntos no visor deve ser de molde a conduzir a vista do utilizador para o indicador correcto e evitar qualquer coisa que o possa distrair. Deve evitar-se encher o visor.

Indicadores de linha («prompts»)

Arranje alguém que olhe para os indicadores de linha que utilizou, antes de dar o programa por terminado. Depois de passar dias ou semanas a desenvolver o programa, não se duvida que você saiba qual é o objectivo de cada introdução de dados. Os outros utilizadores não farão a menor ideia para que serve o programa, nem a que se destina cada introdução de dados, a não ser que lhes seja dito. Isto aplica-se também quando se pretende que ninguém mais mexa no programa. Se deixarmos o programa sem lhe mexermos durante algumas semanas, quando lhe pegarmos podemos ficar tão intrigados como qualquer outra pessoa.

Formatação

Sempre que haja dúvidas, especifique o formato das introduções de dados: é em algarismos ou letras, trata-se de um valor máximo, é limitado o comprimento em caracteres, deverão utilizar-se

vírgulas entre os assuntos...? Considere o exemplo de um menu que aparece no visor como segue:

0 — Fim do programa
1 — Introdução de novos assuntos
2 — Apagar assuntos
3 — Busca de assuntos
4 — Ver dados
— Qual é que pretende?

Embora pareça suficientemente claro, pode estar certo de que existirão pessoas que não percebem por que é que, introduzindo as letras «Ver dados», nada acontece, ou o programa pára. A solicitação deve especificar que é necessário introduzir um número. Por outras palavras, se o programa pressupõe que a introdução de dados se processará de determinada maneira, deverá primeiro dar conhecimento disso ao utilizador.

Pratique a simplicidade

Não faça combinações complicadas ou demasiado compridas para a introdução de dados. Se necessita mesmo de introduzir dez assuntos em linha, parta-os visualmente no visor, por forma a obter grupos afins, e talvez seja melhor limpar o visor entre cada grupo. Paradoxalmente cometem-se mais erros com introduções complexas quando o utilizador já se tornou familiar ao programa dado que menos cuidado se põe na leitura dos indicadores. O facto de haver alguém que já utilizou o programa correctamente algumas vezes, não significa que não comecem a responder erradamente aos indicadores quando estiverem mais habituados ao programa.

Torne-o consistente

Tenha a certeza de utilizar as mesmas convenções ao longo de todo o programa. Se a introdução de um «0» normalmente provoca o abandono da função em curso e se o utiliza noutro ponto para apagar assuntos, não fique surpreendido se alguém perder a informação importante, quando pensava sair da função e regressar ao menu principal.

Tenha especial cuidado com as introduções de dados em que a ordem é por vezes alterada. Se, normalmente, ao longo do programa pede nome, endereço e idade, mas, além disso, outra informação é necessária, por exemplo, se a idade for superior a 65 anos, não coloque apenas mais um indicador no visor, sem aviso prévio. O utilizador foi habituado a introduzir três assuntos e é mais que certo que carregará ENTER outra vez, quando os dados ainda não tiverem desaparecido do visor, após a terceira introdução. Se pretender alterar a ordem a que o utilizador se acostumou, ponha uma cor diferente a piscar no visor, ou introduza um sinal sonoro, para recordar que qualquer coisa diferente está a acontecer.

Confirme as introduções de dados

É boa ideia assumir que, apesar de todas as nossas precauções, cometemos erros: devido a cansaço, por saturação ou ainda por descuido. Podemos, no entanto, reduzir ao mínimo esses erros imprimindo toda a informação que o utilizador acabou de introduzir, com outro formato, e pedindo confirmação sobre a sua exactidão. Pode perfeitamente acontecer que o utilizador detecte que a idade e o número de telefone foram trocados. A colocação no visor de uma réplica da entrada implica que nada está concluído, em relação a determinada introdução de dados, até que o utilizador confirme que a informação é a pretendida. A colocação directa de dados na matriz principal torna muito difícil remediar a situação quando o utilizador descobre que foi cometido um erro.

Independentemente de ser desejável ter uma réplica dos dados no visor, para confirmação, é sempre sensato proteger as matrizes principais de dados, através da aceitação antecipada de dados em variáveis temporárias. Só depois de a introdução de dados ter sido confirmada, ou feitos alguns cálculos que possam levar a concluir que os dados introduzidos estão correctos, se devem introduzir os dados nas matrizes principais, para não correrrmos o risco de corromper a principal estrutura de armazenamento de dados.

CAÇA AO ERRO NA INTRODUÇÃO DE DADOS — ALGUMAS TÉCNICAS DE PROGRAMAÇÃO SIMPLES

O uso das regras do senso comum como as descritas anteriormente reduz o número de erros cometidos proporcionalmente ao esforço desenvolvido nesse sentido. No entanto continuaremos a cometer erros, e vamos agora examinar algumas maneiras de tornar os programas mais imunes aos erros pela introdução de verificações e conferências no programa.

Estabelecer limites

Uma das fontes de falha de programas ocorre quando se concebe o programa para trabalhar com dados que variam entre certos limites e se introduz um assunto que cai fora desses limites. Vejamos, por exemplo, o que acontece se escrevermos um programa cujas introduções sucessivas são de dois números e em que, posteriormente, se divide o primeiro pelo segundo. O programa parará se o segundo número for zero. É claro que o indicador deveria orientar o utilizador no sentido de só introduzir números superiores a zero, para obviar à inevitável mensagem de erro. Mesmo com esta precaução, existe ainda a possibilidade de erro por digitação incorrecta, ou porque o utilizador é mais obtuso que o habitual. Se o programa só pode funcionar dentro de certos limites, é sensato introduzir-lhe algumas verificações, para que a informação introduzida não saia desses limites. O comprimento de uma cadeia e o valor de um número são as duas limitações mais usuais. Deve ainda prever-se a hipótese de incorrectas introduções de dados.

Valor de um número

No desenvolvimento de um programa, e posteriormente à caça ao erro, uma das coisas a determinar é a amplitude de variação dos valores numéricos toleráveis pelo programa. No exemplo acima tipificado, zero é um caso óbvio de introdução não válida. Limites como estes podem facilmente ser controlados em verificações de erros de uma linha. Suponhamos que a introdução de dados necessária é um número entre 1 e 10, inclusive:

```
1000 REM*********************************
1010 REM RANGE CHECK FOR NUMBER
1020 REM*********************************
1030 nn=-1
1040 CLS
1050 WHILE nn<1 OR nn>10
1060 INPUT "Enter a number in the range
1-10: ";nn
1070 IF nn<1 OR nn>10 THEN PRINT "Number
 out of range." : PRINT
1080 WEND
1090 :
1135 CLS
```

Esta é uma rotina simples, que não necessita de explicação, mas note que ela informa o utilizador *por que é que* a introdução de dados foi rejeitada, se for caso disso. Há poucas coisas mais frustrantes que uma introdução de dados que é rejeitada sem se saber porquê.

Comprimento de uma cadeia

Não é só o facto de os números estarem fora dos limites desejados que entope o programa. Se o programa espera uma cadeia de um determinado comprimento e não a recebe por excesso de comprimento, também pode parar. Evitar isto é um pouco diferente da técnica utilizada para números:

```
2000 REM*********************************
2010 REM CHECKING STRING LENGTH
2020 REM*********************************
2030 t$=""
2040 WHILE LEN(t$)<1 OR LEN(t$)>10
2050 INPUT "Enter string (1-10 character
s): ";t$
2060 IF LEN(t$)<1 OR LEN(t$)>10 THEN PRI
NT "String too long." : PRINT
2070 WEND
2080 :
```

Neste caso, a verificação é de que o comprimento da cadeia cai dentro da amplitude 1-10, e as linhas correspondem exactamente às linhas da técnica de verificação de números, exposta anteriormente.

Entradas trocadas

Quer através de um erro de digitação, quer por deficiente leitura do indicador de entrada, acontece que o utilizador faz uma introdução de dados que resulta, não tanto fora dos limites esperados, mas sobretudo incompreensível para o programa. No caso de entradas numéricas, acontece muitas vezes introduzir-se uma letra em vez de um número. Felizmente o próprio sistema pode resolver a situação com uma mensagem de erro «Redo from start». Para o caso das cadeias, acontece que o programa pode interpretar como instruções uma lista de cadeias — e a cadeia introduzida não corresponder a nenhuma delas.

Nos casos em que o programa for concebido para aceitar instruções constituídas por uma cadeia, entre muitas possíveis, pode estabelecer-se confusão se uma instrução for mal digitada na introdução. Consideremos o exemplo de um programa que permite a introdução de dados em todos os meses do ano. O utilizador pode ser levado a introduzir o número do mês — e isso conduzirá ao erro. Em vez disso pode decidir-se que apenas seja permitido que o utilizador introduza o nome do mês em causa. Para este caso deve haver uma verificação de que a entrada era uma das permitidas:

```
3000 REM*********************************
3010 REM UNKNOWN STRINGS
3020 REM*********************************
3030 DIM month$(11)
3040 DATA january,february,march,april,m
ay,june,july,august,september,october,no
vember,december
3050 CLS
3060 FOR i=0 TO 11 : READ month$(i) : NE
XT i
3070 found=0
3080 WHILE found=0
3090 INPUT "Name of month: ";t$
```

```
3100 FOR i=0 TO 11
3110 IF LOWER$(t$)=month$(i) THEN found=
1 : i=11
3120 NEXT i
3130 IF found=0 THEN PRINT "Invalid mont
h name." : PRINT
3140 WEND
3150 PRINT : PRINT : PRINT "Press any ke
y."
3160 t$=INKEY$ : IF t$="" THEN 3160
3170 :
```

Neste caso, os nomes dos meses são guardados na matriz MONTH$, quando o programa começa pela primeira vez. A rotina verifica o que foi introduzido, em correspondência com os nomes dos meses registados, e, se não houver coincidência, imprime uma mensagem de erro.

Uma segunda observação

Quando analisámos as medidas de bom senso atrás mencionadas, verificámos que uma das medidas mais efectivas para reduzir o nível de erros numa introdução de dados é forçar o utilizador a olhar outra vez para os dados introduzidos, para confirmar se estão correctos. Se desejar fazer isto durante a execução normal de um programa, pode ser uma ajuda recorrer a uma sub-rotina que poupa espaço e, ao mesmo tempo, garante que a apresentação é uniforme ao longo do programa. Um exemplo de uma sub-rotina desse tipo é apresentada seguidamente:

```
4000 REM*********************************
4010 REM THE SECOND LOOK
4020 REM*********************************
4030 DIM query$(20),prompt$(20)
4040 prompt$(0)="Item 1"
4050 prompt$(1)="Item 2"
4060 prompt$(2)="Item 3"
4070 nq=3
```

```
4080 CLS
4090 GOSUB 4120
4100 STOP
4110 :
4120 WINDOW #1,1,40,10,10+nq+3
4130 q$="" : WHILE LOWER$(q$)<>"y"
4140 PAPER #1,0 : PEN #1,1 : CLS #1
4150 FOR i=0 TO nq-1
4160 PAPER #1,1 : PEN #1,0
4170 PRINT #1,prompt$(i)": ";
4180 PAPER #1,0 : PEN #1,1
4190 INPUT #1,"",query$(i)
4200 NEXT i
4210 :
4220 CLS #1
4230 FOR i=0 TO nq-1
4240 PRINT #1,prompt$(i)": ";query$(i)
4250 NEXT i
4260 PRINT #1: INPUT #1,"Are these corre
ct (Y/N): ";q$
4270 WEND
4280 CLS #1
4290 RETURN
4300 :
```

Com esta rotina consegue-se colocar uma série de questões baseadas na variável NQ, ou (N)úmero de (Q)uestões. Antes de chamarmos a sub-rotina, as questões são guardadas na matriz PROMPT$. As respostas são armazenadas em QUERY$. Quando todas as perguntas foram colocadas, tornam a ser impressas no visor com as respostas em *inverse*. Se o utilizador não responder «Y» (sim), à pergunta «ARE THESE CORRECT» (Está tudo correcto), as perguntas e respostas são retiradas e apresentadas de novo.

A formatação do visor é executada através de uma janela que destina uma área para os indicadores e, quando desnecessários, os elimina de maneira simples.

A formatação inicial dos indicadores, e a maneira como serão apresentados de novo ao utilizador, é assunto de bom gosto, somente deverá ter-se em conta que alterar o seu aspecto torna mais

fácil para o utilizador concentrar-se neles, quando aparecerem de novo.

É evidente que não há inconveniente em utilizar esta rotina num pequeno programa ou num onde só haja um ou dois indicadores. No entanto, em programas que tenham muitas perguntas, uma sub-rotina destas pode aumentar a precisão das respostas e, ao mesmo tempo, reduzir a necessidade de incluir linhas separadas para executar verificações, sempre que um indicador é utilizado.

CONTROLO DAS MENSAGENS DE ERRO: UMA SOLUÇÃO ELEGANTE

Até aqui considerámos as rotinas que podem ser introduzidas nas várias partes do programa, para fornecer uma série de armadilhas para os erros, principalmente devidos a incorrectas introduções de dados. Tudo isto depende de sabermos o que é uma incorrecta introdução de dados logo que seja feita. Nem sempre é possível. Por vezes serão feitas introduções que, não sendo compridas de mais, e estando dentro dos valores aceitáveis, ainda causarão problemas ao programa.

Para mais, corremos o risco de encher o programa com rotinas para impressão de mensagens e execução de testes, rotinas que ocuparão memória valiosa. A solução é termos à mão um método de lidar com erros numa secção do programa, poupando espaço noutros lados, e, ao mesmo tempo, possuirmos uma ferramenta mais robusta, à prova de ruptura do programa («crash»).

Felizmente o Amstrad tem excelentes características construtivas, que encorajam o programador a controlar os erros, de uma forma económica, com a instrução ON ERROR e o sistema de variáveis ERR e ERL. A pequena rotina que de seguida se apresenta mostra como ON ERROR pode ser utilizada para eliminar uma determinada verificação de erros:

```
5000 REM*********************************
5010 REM ON ERROR
5020 REM*********************************
5030 CLS
5040 ON ERROR GOTO 5130
```

```
5050 nn=0 : WHILE nn>=0
5060 INPUT "Input a number: ";nn
5070 temp=100/nn
5080 result=temp
5090 PRINT temp : PRINT
5100 WEND
5110 STOP
5120 :
5130 IF ERR=11 THEN PRINT "Cannot divide
 by zero."
5140 temp=0
5150 RESUME NEXT
5160 :
```

Para pôr o sistema a trabalhar, apenas necessitamos de garantir que qualquer resultado que conduza a uma divisão por zero fica retido na variável temporária TEMP, por forma a ser colocado em zero ou outro valor predeterminado.

O processo de controlo de erros não necessita de parar neste ponto. A rotina que se escreve a seguir é o esqueleto de algo que permite ao programa controlar as mensagens de erro e as acções que conduzem à limitação dos prejuízos que os erros causam:

```
6000 REM*********************************
6010 REM COMPLETE ERROR MESSAGE SET
6020 REM*********************************
6030 DIM MESSAGE$(50)
6040 message$(6)="Cannot cope with inter
mediate or final  result."
6050 message$(11)="Cannot divide by zero
."
6060 message$(40)="String too long for r
equired format."
6070 :
6080 CLS
6090 ON ERROR GOTO 6300
6100 INPUT "Input a number: ";nn
```

```
6110 temp=100/nn
6120 result=temp
6130 PRINT temp
6140 GOSUB 6490
6150 :
6160 FOR i=1 TO 40
6170 temp=10^i
6180 PRINT temp
6190 NEXT i
6200 GOSUB 6490
6210 :
6220 maxlen=10
6230 INPUT "Input a string: ";temp$
6240 IF LEN(temp$)>maxlen THEN ERROR 40
6250 result$=temp$
6260 PRINT result$
6270 :
6280 STOP
6290 :
6300 done=0
6310 WHILE done=0 AND (ERR=6 OR ERR=11 O
R ERR=40)
6320 PRINT message$(ERR)
6330 IF ERR=6 OR ERR=11 THEN GOSUB 6380
6340 IF ERR=40 THEN GOSUB 6400
6350 done=1 : WEND
6360 IF done=1 THEN RESUME NEXT ELSE ERR
OR ERR
6370 :
6380 temp=0
6390 RETURN
6400 temp$=LEFT$(temp$,maxlen)
6410 RETURN
6420 FOR i=1 TO 2
6430 PRINT 1000*1E+38
6440 NEXT i
6450 STOP
6460 PRINT "error "ERR
6470 RESUME NEXT
```

```
6480 :
6490 PRINT : PRINT "Press any key."
6500 t$=INKEY$ : IF t$="" THEN GOTO 6500
6510 CLS
6520 RETURN
```

Aqui, a matriz MESSAGE$ é usada para imprimir qualquer mensagem de erro que desejamos retirar da alçada do sistema. Neste caso, a rotina assume a responsabilidade de duas mensagens, lidando com o problema do mesmo modo que a rotina anterior, atribuindo o valor zero a uma variável temporária. É criada uma terceira mensagem de erro para tratar das cadeias demasiado compridas e que necessitam de ser truncadas. O corpo principal da rotina representa simplesmente o tipo de linhas do programa principal que utilizam o sistema de caça ao erro. No caso destes três erros (ou outros que tenha o cuidado de incluir), o sistema lidará com eles da maneira habitual.

TRATAMENTO DOS ERROS QUE SURGEM TARDIAMENTE

Antes de terminarmos o problema dos erros necessitamos de fazer referência aos erros que aparecem no meio da execução de cálculos complexos, em que nem desejamos parar o programa, originando talvez a perda de grande quantidade de trabalho já realizado, nem atribuir um valor qualquer a uma entidade qualquer, para continuarmos, sujeitando-nos à corrupção dos dados.

Suponhamos que um certo programa aceita uma introdução de dados e que depois passa o controlo a cinco sub-rotinas sucessivas, cada uma executando tarefas diferentes, terminando com o armazenamento definitivo dos dados. Suponha, além disso, que se descobre que na quarta das cinco sub-rotinas existe alguma coisa errada em relação aos dados introduzidos, nada que provoque a ruptura do programa no momento presente, mas, mesmo assim, algo que poderá tornar disparatado o resultado final. Que podemos fazer?

Poderíamos certamente inserir uma sub-rotina de verificação de erros na quarta sub-rotina do programa. Ficamos no entanto com a quinta sub-rotina, que felizmente processará os dados e criará um disparate. Não é sensato terminar a quarta sub-rotina

quando detectamos o erro, mas desse modo a quinta sub-rotina será executada e, como já vimos, isso será desastroso.

Devemos em primeiro lugar registar o facto de ter sido detectado um erro e utilizar esse registo para garantir que não será efectuado mais trabalho com base naqueles dados.

A solução será registar todos os erros encontrados no decurso do programa numa variável única, a que chamaremos PROBLEM (problema), que será testada regularmente, e que será usada para se concluir da possibilidade de execução de determinadas tarefas. Com este método podemos inserir novas verificações de erros e identificar outros erros; mas, em vez de trabalharmos com eles imediatamente, podemos simplesmente registar a situação em PROBLEM com uma linha do tipo:

```
1070 IF DAY>365 THEN PROBLEM = 7
```

ou podemos utilizar ON ERROR para evitar que o programa termine num ponto inconveniente. A rotina que lida com o erro tomará o seguinte aspecto:

```
1070 TEMP = 0
1080 PROBLEM = 11
1090 RESUME NEXT
```

As secções do programa, tais como as cinco sub-rotinas atrás citadas, a título de exemplo, serão colocadas no interior de ciclos que as isolem, tais como:

```
100 DONE = 0
110 WHILE DONE = 0 AND PROBLEM = 0 : DONE = 1
:
:
:
200 WEND
```

Sempre que PROBLEM regista um erro de qualquer tipo, as secções do programa dentro dos ciclos respectivos não serão executadas. Finalmente, quando tivermos saído das séries de cálculos, podemos usar o valor de PROBLEM para decidir qual a acção final a desencadear, tal como desprezar o último assunto introduzido e a impressão de uma mensagem de erro. Nos casos em que não for muito importante a paragem do programa, PROBLEM pode

muito simplesmente ser traduzido para um sistema de erro, para ser trabalhada por uma rotina, como a indicada no exemplo anterior, através de uma linha do tipo:

```
1070 ERROR PROBLEM
```

As possibilidades destas técnicas de tratamento de erros são intermináveis e fazem parte integrante dos mais complexos programas de aplicações.

CONCLUSÃO

Utilizando as técnicas estudadas neste capítulo será capaz de produzir programas que sobreviverão ao uso abusivo dos utilizadores. Até onde se deseja ir na protecção dos programas dependerá do seu grau de complexidade e da importância dos dados que eles manuseiam. Mesmo um programa simples pode trabalhar com informação que leve muito tempo a introduzir, implicando graves consequências quando há uma ruptura do programa depois de meia hora de trabalho. Talvez, no entanto, a maior satisfação advenha do facto de ter concebido um programa que parece controlar os acontecimentos, em vez de se ter de manusear com cuidado, com receio de que pare no meio da maior confusão.

Mesmo assim, esteja alerta! Não se vanglorie de que o seu programa favorito é muito robusto. Pode ser que a pessoa que o está a ouvir carregue numa única tecla, em que você nunca tinha pensado, e a sua reputação se desfaça em fumo.

CAPÍTULO 6

Como armazenar e recuperar a informação

Chegará o dia em que utilizaremos uma geração de computadores com memórias tão vastas que terão capacidade para todos os programas e informação que desejarmos. Além disso, a informação e programas farão parte integrante da memória, sempre disponíveis quando se ligar a máquina. As possibilidades ao nosso dispor com essas máquinas representarão um importante salto em frente, comparável ao que representou a introdução dos microcomputadores domésticos há alguns anos. Até que esse dia chegue, os possuidores de micros têm de aprender a viver com máquinas que apenas comportarão uma parte da soma total de informação que se utiliza dia a dia.

Para obviar a este facto, de uma máquina como o Amstrad só poder conter, de cada vez, um programa e os dados associados a ele, o Amstrad prevê uma extensão de memória. É o papel desempenhado pelo gravador de cassetes, ou *drive* de disco. Se compararmos a velocidade de transferência de dados a partir dos *chips* da memória RAM com a do gravador de cassetes, e até mesmo do *drive* de disco, estes parecem andar a passo de caracol. A programação concreta, e especialmente a que se utiliza na maior parte das aplicações úteis, aprenderá, contudo, a fazer uso da capacidade adicional do gravador de cassetes ou do *drive* de disco, para ultrapassar tanto as limitações de memória do Amstrad como o facto de a presente geração de micros não poder conter a informação quando a máquina está desligada.

Estranhamente, embora muitos possuidores de micros pareçam preparados para despender grandes quantias com extras que aumentam a capacidade de memória das suas máquinas, muitas ve-

zes com quantias mais pequenas poderiam utilizar, com benefício, a capacidade maciça de armazenamento de um pequeno disco, ou até mesmo de um humilde gravador de cassetes, para muitos dos programas feitos em casa. Neste capítulo examinaremos algumas técnicas necessárias para utilizar melhor qualquer deles. Não se trata de um capítulo exaustivo do assunto, especialmente no que diz respeito ao *drive* de disco. Mesmo assim, utilizando as técnicas descritas nestas páginas, ficará apto a armazenar a informação de forma mais fiável e em maiores quantidades. Poderá permutar informação com a que está armazenada no disco, ou fita, e fazer melhor uso da memória disponível.

COMO GRAVAR PROGRAMAS

O primeiro e mais evidente uso da gravação externa é guardar o seu programa para utilização futura. É surpreendente como a maior parte das pessoas tem pouco cuidado com a gravação das últimas actualizações dos programas que estão a desenvolver, esquecendo-se de verificar se um programa foi devidamente gravado, limitando-se a conservar apenas uma cópia de programas importantes, prejudicando fitas e discos por os deixarem de qualquer maneira sujeitos ao mau tempo. Damos seguidamente uma ou duas regras de bom senso para a gravação de programas:

Grave (SAVE) com regularidade

Quando estiver a desenvolver novos programas, grave-os regularmente. Como qualquer outro microcomputador, o Amstrad pode perder os programas se houver uma falha momentânea de energia eléctrica, se alguém desliga a ficha ou até porque na sua programação arranjou maneira de transtornar o equilíbrio do Amstrad. A quantidade de trabalho perdida dependerá do tempo decorrido desde a última vez que gravou. Se um programa entra rapidamente, não se deverá estar mais de quinze minutos sem gravar. Se estivermos a caçar erros, de modo que relativamente poucas alterações são feitas, pode aumentar-se o período para meia hora. Na realidade depende da quantidade que se está preparado para perder. Se não gravar programas regularmente, tarde ou cedo perderá um programa importante, que levou muito tempo a introduzir.

Introduza-lhe uma rotina

Para tornar mais fácil gravar um programa e para me encorajar a fazê-lo, incluo sempre três linhas no começo dos meus programas, linhas essas que permitem gravar sem ter de escrever o nome do programa todas as vezes:

```
1 GOTO 3
2 SAVE «NOME DO PROGRAMA» : SAVE «!NOME DO PROGRAMA» : STOP
3 REM
```

A inclusão de uma rotina deste tipo num programa tem a virtude de evitar que se grave o programa com um nome errado, devido a má digitação. Pode simplesmente gravar-se o programa introduzindo «GOTO 2». Como bónus adicional, todos os programas podem ser iniciados com um uniforme «GOTO 1», se desejar que as variáveis não sejam apagadas com a utilização de RUN e apagar quaisquer variáveis armazenadas. O programa é gravado duas vezes, o que é sempre sensato, e o uso de «!» antes da segunda referência ao nome do programa suprime as mensagens do sistema referentes à fita. A única precaução adicional que se recomenda é verificar regularmente se o processo SAVE foi executado com sucesso, através da instrução CAT.

Quando se utiliza um disco pode incluir-se uma rotina similar à data anteriormente:

```
1 GOTO 4
2 ERA, «NAME.BAK» : REN,«NAME.BAK»,«NAME.BAS»
3 SAVE «NAME.BAS» : CAT : STOP
4 REM
```

A técnica que empregamos aqui é a de guardar duas cópias do programa — uma é a última versão gravada, outra é a versão anterior. Pretende-se, deste modo, garantir que se fizermos um erro, por exemplo o apagamento de uma importante parte do programa, antes de o ter gravado, a versão primitiva esteja ainda disponível. Note que as primeiras duas vezes que utilizar esta rotina vão aparecer duas mensagens de aviso dizendo que tanto os ficheiros «.BAK» como «.BAS» não foram encontrados. Não se preocupe com estas mensagens.

Um programa por cassete

Conquanto as fitas mais compridas sejam mais adequadas ao desenvolvimento dos programas, não há duvida de que o melhor caminho para armazenar programas permanentemente é usar cassetes especiais para esse fim — e usar uma cassete para cada programa. Será um pouco mais caro, mas também significará que todos os programas estarão instantaneamente disponíveis, sem ter de procurar no meio de grandes cassetes a posição inicial correcta. Reduz também o risco de sobreposição inadvertida de gravações sobre programas existentes.

Limpe as cabeças de gravação

Mantenha limpas as cabeças de gravação e reprodução do seu gravador. Os conjuntos de limpeza das cabeças são baratos e fáceis de utilizar, comparados com a frustração de perder um programa porque as cabeças ficaram revestidas de óxido proveniente das suas cassetes.

Faça várias cópias

Guarde mais de uma cópia dos seus programas, com a segunda cópia em sítio totalmente diferente das fitas e discos que normalmente utiliza. Há sempre o risco de a cópia de trabalho ficar estragada por qualquer motivo, devido a calor ou às brincadeiras de uma criança com um íman. Se não quiser pôr os duplicados dos programas cada um em sua cassete, poderá armazená-los todos em cassetes mais compridas e regravá-los em pequenas cassetes se estiverem sempre a ser precisos.

COMO GRAVAR E CARREGAR (LOAD) DADOS

Muitos programas que estamos a utilizar de momento necessitam de grande quantidade de dados para trabalho. Nos casos em que esses dados não são fixos, por forma a terem de ser escritos no programa, temos duas maneiras de proceder: podemos reintroduzir

os dados todas as vezes ou ultrapassar o problema de gravar dados em fita ou disco e introduzi-lo no programa mais tarde.

Quando decidimos gravar dados em fita ou disco, o primeiro problema com que deparamos é o de identificarmos o que desejamos gravar. Não tem qualquer utilidade gravar o conteúdo de uma matriz e esquecermo-nos de gravar a variável associada à matriz e gravar todos os assuntos que lá havia. Primeiro que tudo, faça uma lista completa das variáveis essenciais do seu programa. Claro que *não são* todas as variáveis do programa. Algumas variáveis terão valores quando o programa corre. Só é necessário gravar as que são necessárias para pôr o programa de novo operacional quando dele necessitarmos. Um ponto a ter presente é só gravar a parte das matrizes que no momento contribuem para alguma coisa. Pode acontecer que tenha definido uma matriz de cadeias de 500 linhas na qual estamos a fazer gradualmente o armazenamento de dados. Se até aí só utilizou 170 linhas da matriz, é melhor utilizar uma variável para gravação apenas das linhas usadas, e só dessas. A razão deste procedimento é que gravar e carregar dados leva tempo, e quanto menos gravarmos menos tempo levará e menos fita ou disco será ocupado.

Depois de termos identificado as variáveis e partes de matriz que pretendemos gravar, devemos colocá-las num módulo que mais eficazmente as gravará. A Amstrad teve grande cuidado na concepção dos sistemas de fita e disco, para garantir a compatibilidade dos dois. Enquanto os possuidores de outros micros têm de fazer grandes alterações aos programas se mudam de fita para disco, os donos do Amstrad 464 podem comprar um *drive* para disco ou *upgrade* (aumentar a capacidade) para o 664 sem problemas. Para armazenar material em fita ou disco, tudo o que se precisa é dizer primeiro ao sistema que se deseja comunicar com o dispositivo de armazenamento de dados e conhecer o nome do grupo de assuntos que pretendemos gravar. Isto é conhecido por «abrir um ficheiro» *(opening a file)*. Existem dois comandos para isto, um para enviar informação para o disco ou fita, outro para a retirar OPENOUT e OPENIN.

A partir desse momento, os assuntos podem ser enviados, ou recuperados, da fita, utilizando as instruções de extracção *(output)* ou introdução *(input)*, tais como PRINT, WRITE, INPUT, LINE INPUT.

COMO ESCREVER PARA UM FICHEIRO

Depois de abrir o ficheiro é necessário começar a colocar a informação dentro dele. Isto faz-se usando combinações das instruções PRINT e WRITE. Qualquer que seja precedida de «PRINT#9,» será colocada no ficheiro da mesma maneira que uma instrução PRINT o colocaria no visor. Pode certamente escrever-se da mesma maneira no visor através da abertura de uma janela. Esta parecença familiar entre os diferentes usos de PRINT tem certas implicações.

Primeiro, o Amstrad não faz uma separação automática entre variáveis que são impressas pela mesma linha, utilizando uma instrução do tipo:

```
PRINT#9, A$,B$,C$
```

Uma linha destas daria como resultado que, quando os dados fossem chamados da fita, ficávamos com uma cadeia única, constituída pelas cadeias A$,B$ e C$, em conjunto com espaços entre elas.

No caso de matrizes consegue-se a separação dos assuntos com a utilização de um ciclo para gravar um assunto de cada vez, por exemplo:

```
1710 FOR I = 0 TO ITEMS
1720 PRINT#9,A$(I)
1730 NEXT I
```

Os assuntos únicos necessitam de ser escritos no ficheiro, seguidos por um carácter de «return» (CHR$(13)). Isto faz-se habitualmente definindo uma cadeia, seja R$ igual a CHR$(13) quando o programa é iniciado a primeira vez, e separando todos os assuntos com R$ para evitar escrever «CHR$(13)» todas as vezes:

```
1740 PRINT#9,TT$ R$ CD$ R$ IT R$ NN R$ QQ$
```

Repare na ausência de pontuação entre os assuntos. Pode pôr, se quiser, ponto e vírgula entre os assuntos, mas ao Amstrad é-lhe indiferente, tal como sempre que se utiliza PRINT com vários assuntos numa linha.

A outra alternativa é usar WRITE, que é especialmente concebido para tornear o problema, colocando aspas nos extremos das

cadeias à medida que são gravadas. Tornando a carregar com INPUT dá a cadeia original, enquanto LINE INPUT também inclui as aspas na cadeia. Devido a esta separação, uma linha como:

```
1740 WRITE#9,A$,B$,C$
```

produziria um resultado sensato quando o material fosse reintroduzido.

Em segundo lugar, tal como em quase todos os micros, o 464/664 em BASIC não é capaz de recarregar todos os caracteres que grava. As razões para isto são complexas, mas andam à volta do facto de que alguns caracteres representam o que irá ser tomado por separadores entre assuntos ou sinais de fim de ficheiro. Os caracteres que não são recarregados dependem das combinações de comandos de introdução e extracção utilizados, mas não há nenhuma combinação que grave e recarregue os 256 caracteres desde 0 até 255. Se deseja recarregar com um mínimo de fiabilidade cadeias que contenham caracteres de controlo, isto é, com um valor compreendido entre 0 e 32, deve traduzir, pelo menos, parte da cadeia, carácter por carácter, para números e depois armazenar esses números todos de uma vez:

```
1710 PRINT#9,LEN(A$)
1720 FOR I = 1 TO LEN(A$)
1730 PRINT#9, ASC(MID$(A$,I,1))
1740 NEXT I
```

Note que se armazena o comprimento da cadeia para, quando recarregarmos o programa, sabermos quando parar a tradução da cadeia de caracteres.

Terceiro: a ordem pela qual se gravam os dados pode ser vital para sabermos se é possível recarregar com sucesso. Voltando a um exemplo anterior, suponha que tem uma matriz de cadeias de 500 elementos e uma variável chamada «ITEMS» para registar quantos elementos vão ser usados de momento. Quando quiser gravar o conteúdo da matriz utilizará um ciclo no género do seguinte:

```
1710 FOR I = 0 TO ITEMS
```

Quando recarregarmos o programa já deverá possuir um valor para ITEMS antes de poder retirar os dados da fita. Uma regra simples é: se forem utilizadas variáveis do programa para controlo

da gravação dos dados, então essas variáveis devem ser armazenadas *antes* dos dados.

Finalmente, quando todos os dados tiverem sido armazenados, o ficheiro deverá ser «fechado». Esquecermo-nos disto significará que qualquer futura tentativa de abrir um ficheiro com o mesmo número esbarrará com uma mensagem de erro. Para fechar um ficheiro utiliza-se:

CLOSEOUT — não há necessidade de pôr o nome do ficheiro.

COMO CARREGAR A PARTIR DE UM FICHEIRO

O processo para carregar de novo os dados na memória é de certo modo a imagem num espelho do modo de proceder para a gravação. As diferenças principais são:

1) As instruções principais para retirar a informação são INPUT#9 e LIN E INPUT#9. Dos dois, LINE INPUT é o que é capaz de recarregar cadeias que contenham vírgulas — INPUT vê a vírgula como o fim de um assunto que esteja a ser introduzido.

2) Desde que tenha sido feita a separação adequada entre os assuntos dos dados quando foram armazenados, não há necessidade de fazer qualquer coisa para detectar a separação entre eles:

```
1840 INPUT#9,TT$,CD$,IT,NN,QQ$
```

será suficiente para recuperar os dados armazenados na fita no exemplo anterior da utilização de R$ como separador de assuntos.

3) As cadeias que foram armazenadas na forma de valores numéricos devem ser reconstituídas como cadeias:

```
1810 A$=«» : INPUT=//9,LS
1820 FOR I=1 TO LS
1830 INPUT=//9,T:A$=A$+CHR$(T)
1840 NEXT I
```

4) Em geral, o melhor lugar para o módulo que recarrega os dados é mesmo a seguir ao que os grava. Isto porque a maneira mais segura de escrever a rotina de carregar que segue passo a passo a rotina de armazenamento é chamar *(edit)* os números de linha da rotina de gravação e mudar PRINT ou WRITE para INPUT ou LINE INPUT. Mesmo assim, as duas rotinas podem ser chamadas pelo programa de diferentes maneiras.

A rotina de gravar deve ser uma função normal do programa chamada a partir do menu, talvez com o lembrete para o utilizador de que é melhor gravar os dados antes que se percam, antecedendo a paragem do programa (que de novo deve ser uma opção do menu em vez de simplesmente premir «ESC»). Se estamos a introduzir grandes quantidades de dados, o utilizador deverá preferir gravar dados regularmente, prevenindo-se dessa maneira contra a eventualidade de problemas que possam surgir no Amstrad ou no programa.

A rotina de carregar pode ser chamada de maneira muito diferente com a utilização de uma função de «autocarregamento» colocada mesmo no início do programa, de certo modo semelhante à técnica de «auto-inicialização» descrita no cap. 1. No caso da «auto-inicialização», o utilizador tem duas hipóteses: fazer o RUN do programa, que conduz à auto-inicialização dos valores iniciais das funções, ou começando com GOTO para que a «auto-inicialização» não seja executada, nos casos em que já existem dados armazenados no programa. A função de «autocarregamento» do programa permite, adicionalmente, se não existirem dados armazenados no programa, especificar se a introdução de dados se vai fazer a partir da fita ou do teclado. Se for escolhida a introdução a partir da fita, então partes da rotina de «auto-inicialização» serão executadas, nomeadamente as que preparam as matrizes para receberem dados, mas as variáveis não serão respostas nos valores iniciais, já que, de qualquer modo, serão carregadas a partir da fita. Damos a seguir um exemplo de um módulo deste tipo:

```
1000 WHILE BASE=0
1010 DIM A$(100), B$(25),A%(100),
B%(25) : LIMIT=100 : R$=CHR$(13)
1020 INPUT «DESEJA CARREGAR A PARTIR DA FITA
(Y/N):»;Q$
1030 ITEMS=0 : NN=0 : CT=12 : BASE=2
1040 IF LEFT$(Q$,1)=«Y»THEN GOSUB CARREGADORA
1050 WEND
```

A linha de «auto-inicialização» é 1000. BASE é a variável que toma um valor diferente de zero quando o programa possui dados. Assim, se o programa torna a arrancar com GOTO, o ciclo não é executado. A segunda linha dá a dimensão de quatro matrizes e declara uma variável cujo valor não sofre alteração no decurso do programa, e representa o número de assuntos permitidos. Desde que estes sejam declarados, o utilizador pode especificar quando é que os dados são carregados a partir da fita. Tenha presente que, se os dados *são* carregados a partir da fita então *devem* incluir todas as variáveis da linha 1030, que são necessárias à execução do programa. Por esta razão é ainda mais importante que habitualmente soletrar todas as variáveis, mesmo aquelas que têm valor inicial nulo, para servir de lembrança quando estiver a escrever os módulos de gravação e carregamento.

ROTINAS DE SAVE (GRAVAR) E LOAD (CARREGAR): EXEMPLO FUNCIONAL

Apresento seguidamente um exemplo de rotinas de *save/load* tirado de um programa meu. Não há necessidade de procurar compreender as funções das variáveis; o exemplo somente pretende ilustrar o método empregue com conjuntos de dados complexos:

```
8000 REM*******************************
8010 REM Save To Tape
8020 REM*******************************
8030 CLS
8040 PRINT "Save data:":PRINT
8050 INPUT "Name of file:",file$
8060 OPENOUT file$
8070 PRINT #9,nn$
8080 PRINT #9,cu
8090 PRINT #9,it
8100 FOR i=0 TO cu-1
8110 PRINT #9,te$(i,0)
8120 PRINT #9,te$(i,1)
8130 PRINT #9,te(i)
8140 NEXT i
```

```
8150 FOR i=0 TO it-1
8160 PRINT #9,array$(i,0)
8170 PRINT #9,array$(i,1)
8180 PRINT #9,array(i)
8190 NEXT i
8200 CLOSEOUT
8210 RETURN
8220 :

9000 REM*********************************
9010 REM Load From Tape
9020 REM*********************************
9030 CLS
9040 PRINT "Load data:":PRINT
9050 INPUT "Name of file:",file$
9060 OPENIN file$
9070 INPUT #9,nn$
9080 INPUT #9,cu
9090 INPUT #9,it
9100 FOR i=0 TO cu-1
9110 INPUT #9,te$(i,0)
9120 INPUT #9,te$(i,1)
9130 INPUT #9,te(i)
9140 NEXT i
9150 FOR i=0 TO it-1
9160 INPUT #9,array$(i,0)
9170 INPUT #9,array$(i,1)
9180 INPUT #9,array(i)
9190 NEXT i
9200 CLOSEIN
9210 RETURN
```

Neste caso o utilizador tem a opção de escolher o nome do ficheiro em que os dados são gravados ou a partir de que vão ser recarregados. Dois conjuntos de matrizes vão ser gravados ou carregados, com a quantidade de dados para manuseamento baseados no conteúdo de duas variáveis, CU e IT. Finalmente o ficheiro é fe-

chado. Repare que a disposição das duas metades dos módulos é idêntica. Na realidade, a rotina de carregamento foi copiada da de gravação por EDIT (chamamento de linhas), o que garante que os assuntos são chamados na mesma ordem por que foram gravados.

Os possuidores de *drives* de disco devem ter presente que, quando abrem um ficheiro para saída (output), as cópias anteriores do ficheiro ficam destruídas sem aviso prévio. Podem preferir o método descrito na secção de gravação de programas, que consiste em dar um nome diferente a cada versão do ficheiro existente, mesmo que isto implique o aparecimento da mensagem de aviso de que não existe no disco nenhum ficheiro com esse nome.

COMO UTILIZAR O EOF

Antes de darmos por findo o tópico do armazenamento devemos mencionar a variável do sistema EOF *(end of file),* que pode utilizar-se para determinar quando o programa recarregou toda a informação contida no ficheiro, já que, se tentarmos chamar mais, ocorreria um erro.

EOF permite que um programa chame a informação do ficheiro sem sabermos exactamente quantos assuntos existem; é o que se faz no exemplo seguinte:

```
1710 NN=0: OPENIN «TEMP»
1720 WHILE NOT EOF
1730 INPUT=9,T
1740 A(NN)=T
1750 NN=NN+1
1760 WEND
1770 CLOSEIN
```

Esta rotina continuará a aceitar assuntos da fita ou do disco até que o sinal de fim do ficheiro seja encontrado e a execução do programa passe adiante. No entanto, na maior parte dos casos, a marca de EOF não é necessária, porque qualquer programa decente deve registar o número de assuntos de dados que tem armazenados em cada momento e, além disso, quantos têm de ser gravados. Se isto for conhecido, as variáveis representando as quantidades podem ser colocadas no início de cada bloco de dados e o programa

concebido para recarregar esse número específico de assuntos, sem utilização de um sinal EOF. A moral é: não utilize EOF a menos que necessite dele.

CONCLUSÕES

Não há dúvida de que, embora as linhas gerais de armazenar e recuperar dados sejam simples, conseguir acertar o módulo de dados do ficheiro referente a programas complexos é muitas vezes assunto delicado. No entanto, os programas que se destinam a armazenar quantidades importantes de informação necessitam de unidades de armazenamento externo. Os discos e as cassetes são fiáveis e constituem modos relativamente rápidos de armazenar a informação útil. Quaisquer que sejam as suas limitações, é preferível do que trabalhar com os programas que servem só para um conjunto de informação contida no interior do programa ou à informação que é introduzida em cada sessão através do teclado. O esforço de armazenar a informação em boas condições, para uso futuro, vale tanto mais a pena quanto mais valiosa ela for.

CAPÍTULO 7

A selva das ordenações

Este capítulo surge na sequência do meu começo como escritor para utilizadores de microcomputadores. Há alguns anos atrás, uma revista de computadores do Reino Unido publicou um programa concebido para permitir aos utilizadores armazenarem nomes e moradas dos amigos. Quando os nomes fossem todos introduzidos e se carregasse num botão, o programa faria a respectiva ordenação alfabética para utilização futura. Suponho que o programa foi publicado por ser claro e curto, mas logo que o vi desconfiei do método utilizado para a ordenação.

Em vez de introduzir os cem nomes permitidos, escrevi uma rotina simples capaz de gerar os cem nomes e endereços sem sentido. Depois fiz correr o programa. *Meia hora depois* acabou a ordenação. Se tivesse introduzido só noventa nomes e mais tarde lhe juntasse mais um ou dois, levaria mais meia hora só para colocar esses dois no lugar. Um pequeno e útil programa falhou, para todos os fins práticos, por utilizar um método inadequado.

Utilizando um molho de apontamentos de um curso de informática de uma universidade, escrevi um pequeno artigo demonstrando como o programa em questão podia ser acelerado de um factor à volta de 40. Foi o primeiro artigo de fundo que publiquei. Desde essa altura nunca mais parei de ficar embaraçado com o elevado número de aplicações, escritas por possuidores de microcomputadores, que ficam frouxas devido ao desconhecimento dos seus autores dos variados métodos de ordenação disponíveis para diferentes fins, e a correspondente diferença de duração entre eles. Admitamos que o programa da revista foi escrito para a máquina mais lenta existente no mercado. Com uma máquina tão sofis-

ticada como o CPC464 terá de se esforçar *muito* para que a ordenação seja suficientemente lenta para que leve meia hora só com cem assuntos. O princípio continua, no entanto, a ser aplicável. Um método apropriado de ordenação pode distinguir um programa terrivelmente lento de outro suficientemente rápido para que possa ter alguma utilidade.

Neste capítulo examinaremos três métodos de ordenação, mostrando como e porquê trabalham, e por que podem diferir tanto os tempos de execução. No entanto, antes de olharmos para a ordenação no computador, vamos dar uma espreitadela mais simples no que sucede quando um ser humano ordena assuntos, com o intuito de conseguir mais facilmente fazer o mesmo quando se vira para o teclado.

PRINCÍPIOS DA ORDENAÇÃO

Para começar a experiência devemos dispor de dez pedaços de papel. Pequenas fichas de cartolina seria o ideal, mas, não as tendo, arranje simplesmente dez pedaços de papel em forma de quadrado com 8 cm de lado. Escreva nos dez quadrados os algarismos de 0 a 9, não se esquecendo de pôr um traço por baixo do 6 e do 9 para evitar trocá-los entre si quando inadvertidamente lhes inverter a posição. Arranje uma mesa com suficiente espaço livre e disponha os quadrados na seguinte ordem:

7 3 5 1 6 9 4 8 0 2

(Não há nada premeditado nesta ordem; mas, se não a utilizar, os comentários seguintes não farão sentido.)

O objectivo do exercício é ordenar os quadrados por ordem crescente, de 0 a 9, a começar pela esquerda. As únicas limitações são não existirem mais de onze espaços possíveis, sendo dez ocupados pelos quadrados e um de reserva. Significa que, antes de movimentar qualquer quadrado de um lado para outro, se deve retirar primeiro um quadrado da sequência e colocá-lo no espaço de reserva. Desta forma obterá um espaço na sequência, que poderá ocupar com outro quadrado, e um quadrado retirado da sequência depositado no espaço de reserva. O objectivo é obter no fim do processo todos os quadrados por ordem e o espaço de reserva vazio.

Faça as movimentações que entender, mas tente analisar o que vai fazendo.

Ao proceder como foi indicado terá possivelmente feito o seguinte:

1) Analisado a sequência, um a um, para localizar o maior e o menor valor.

2) Retirado o quadrado correspondente à mais alta ou mais baixa posição para o espaço de reserva e movimentado o quadrado certo para o espaço que ficou vazio na sequência.

3) Pode depois escolher entre colocar o quadrado que está no espaço de reserva na sequência e começar outra vez de 1) para o segundo valor mais alto ou mais baixo, ou identificar o local correcto para o quadrado na posição de reserva, retirar o quadrado que está nessa posição e colocar correctamente o quadrado que estava na posição de reserva, economizando assim o número total de deslocações a fazer.

Talvez você tenha procedido de maneira muito diferente, mas se utilizou a massa cinzenta e os olhos, muitas coisas serão verdadeiras no modo como fez a ordenação:

1) Como os quadrados ocupavam eventualmente posições correspondentes aos seus números, foi capaz de ver rapidamente para onde um determinado quadrado devia ir.

2) Como havia um relativamente reduzido número de quadrados pôde facilmente identificar o mais elevado e o menor dos valores da sequência e agir em conformidade.

3) Com um simples olhar foi capaz de ver a sequência no conjunto e ajustar as suas acções em concordância.

Agora tente colocar-se na posição do microcomputador no início de um processo de ordenação: é similar à que *você* encontraria frente a cem quadrados cada um com, por exemplo, o nome de cem pessoas diferentes:

1) Não pode contar com uma ordem regular, que lhe permitiria identificar instantaneamente onde um determinado quadrado ficaria. *Deve* haver uma ordem, mas não há maneira de saber qual é. Claro que seria possível escrever uma ordenação muito rapidamente, supondo que se houvesse cem assuntos então os seus valores seriam espaçados regularmente, de tal modo que, olhando para um

quadrado, fosse possível dizer qual a sua posição. Na realidade uma ordenação deste tipo *só* funcionaria com uma lista ordenada desse modo. Um programa de ordenação normal deve estar apto a lidar com todos os tipos de dados e, de modo diferente do cérebro humano, não pode dizer: «Ah, uma sequência pacífica de números, que me permitirá colocar cada assunto na sua posição correcta, logo à primeira.»

2) Como não existe nenhuma regularidade particular dos dados, não seríamos capazes de identificar o mais elevado ou mais baixo assunto dessa eventual sequência, e depois o segundo mais alto e assim sucessivamente. Se desejarmos encontrar o quadrado de valor mais elevado teremos de examinar quadrado a quadrado, e só no fim poderemos decidir qual é o de valor mais elevado.

3) Como só pode executar uma tarefa de cada vez, e examinar um facto de cada vez, não seria capaz de ter uma visão global de toda a lista. Teria de comparar um assunto aqui, outro acolá, sem ter a visão do que se passa em volta. Se dermos a um ser humano uma lista com a seguinte ordenação:

0 1 2 3 4 5 6 7 8 9

e a instrução de colocá-los por ordem crescente de zero a nove teremos imediatamente a resposta: «Já está.» O micro nunca poderia dar esta resposta sem primeiro observar a lista assunto por assunto.

Tendo presente estas diferenças entre um ser humano e um computador, podemos classificar os métodos de ordenação numa escala que tem num extremo os métodos que aceitam as limitações do computador na totalidade e, no outro extremo, os que tentam imitar alguns dos atalhos que o raciocínio humano comporta. O mais simples de todos os métodos comuns e o mais próximo do funcionamento do computador é o *Bubble Sort*.

A ORDENAÇÃO «BUBBLE SORT»

A essência deste método é a confiança total na capacidade do micro de comparar dois assuntos adjacentes e decidir qual deles é maior. O nome vem da maneira como, à medida que a ordenação prossegue, os valores mais elevados parecem «borbulhar para cima» *(bubble up)* na lista, do mesmo modo que as bolhas nas paredes de um copo que contenha um líquido efervescente.

Como demonstração do que se passa, voltemos aos quadrados de papel que devem dispor-se outra vez na ordem:

7 3 1 5 6 9 4 8 0 2

Tendo em conta as mesmas limitações, nomeadamente que há um espaço de reserva que deve ficar vazio no final, observe o seguinte procedimento:

Comece com o primeiro quadrado, o 7 no extremo esquerdo, compare-o com o segundo, que é o 3. Como o 7 é maior que o 3, retire o 3 e coloque-o no espaço de reserva. Movimente o 7 para a posição original do 3 e depois coloque o 3 no lugar original do 7. Conseguimos assim mover o 7 um lugar para cima na sequência.

Continue a subir com o sete na sequência, com o mesmo procedimento cada vez que encontra um número mais pequeno à direita. Eventualmente terminará com o 7 na quinta posição da sequência, com o 9 à direita.

Porque encontrámos um número maior que 7, ou seja, o 9, transferimos a nossa atenção para este número e procedemos como fizemos com o 7, trocando-o com o número que está à sua direita, se for menor. Como o 9 é o maior número da sequência, o processo continua até que o 9 esteja colocado na posição mais elevada.

Regresse ao princípio da lista e comece outra vez com o 3. Se o número à direita for menor, então troque-o, se não for esqueça-se do 3 e continue com o número que é maior. No fim do processo teremos o 8 na posição 9, à esquerda do 9. Como já descobrimos que o 9 é o maior da lista, quando percorremos a sequência, não há necessidade de comparar o 8 com o 9.

Regresse ao princípio e comece com 1. Terá imediatamente de o trocar pelo 3. Continue pela sequência fora, recordando que desta vez pode deixar os dois últimos assuntos de fora, já que foram correctamente colocados nas operações anteriores.

Repita o processo até passar todos os elementos da lista sem fazer nenhuma troca. Lembre-se que, tal como o computador, você não sabe que a lista está em ordem até ter feito isto, porque não pode ver a lista toda.

Se seguiu o processo correctamente, os quadrados, após cada operação, devem ter um aspecto semelhante aos seguintes:

7 3 1 5 6 9 4 8 0 2 : POSIÇÃO INICIAL
3 1 5 6 7 4 8 0 2 9
1 3 5 6 4 7 0 2 8 9
1 3 5 4 6 0 2 7 8 9
1 3 4 5 0 2 6 7 8 9
1 3 4 0 2 5 6 7 8 9
1 3 0 2 4 5 6 7 8 9
1 0 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9

Quando tiver executado este procedimento e terminado com uma sequência correctamente ordenada, pode agradar-lhe tentar outras sequências ao acaso da sua invenção, para se familiarizar com o método. Há, no entanto, algo mais para aprender, de importância vital para compreender esta ou qualquer outra espécie de ordenação.

Volte à sequência original e comece a ordenação usando o método de *Bubble Sort,* mas desta vez, tome nota numa folha de papel, em colunas separadas, cada vez que faz uma comparação entre dois assuntos (sem ter em conta o modo como a comparação termina em termos de dimensão relativa) e cada vez que troca dois dos quadrados. Pelas minhas contas, o resultado para as oito passagens em que houve alguma alteração, mais uma para nos dizer que a sequência já está em ordem, é o seguinte:

1) 9	comparações	8 trocas
2) 8	comparações	4 trocas
3) 7	comparações	3 trocas
4) 6	comparações	3 trocas
5) 5	comparações	2 trocas
6) 4	comparações	2 trocas
7) 3	comparações	2 trocas
8) 2	comparações	1 trocas
9) 1	comparações	0 trocas

TOTAL: 45 comparações 25 trocas

Aprendeu agora alguma coisa de muito importante para a compreensão do funcionamento de todas as ordenações existentes. Ordenação implica *trocas* e *comparações,* e as diferenças entre estas dependem inteiramente do seu número. Devemos ter em consideração que as trocas levam mais tempo que as comparações. É

possível conhecer as diferentes ordenações do ponto de vista matemático, de acordo com os respectivos totais. Enquanto não fazemos isso, os resultados para a *Bubble Sort* mostram que na melhor das hipóteses (isto é, quando a lista dada está já ordenada), fazer a ordenação de N assuntos requer 0 trocas e N – 1 comparações. Na pior hipótese (isto é, quando se tem uma lista em que os assuntos estão inteiramente em ordem inversa) a *Bubble Sort* necessita de 0,5 * N * (N – 1) trocas e o mesmo número de comparações.

Exemplificando para o caso de cem e mil assuntos, teremos:

100 assuntos: 4950 trocas, 4950 comparações.
1000 assuntos: 499 500 trocas, 499 500 comparações.

Donde se conclui que, à medida que a lista aumenta, a *Bubble Sort* se torna horrivelmente lenta em relação ao número de operações a efectuar.

É claro que têm de ser feitas, em cada caso, trocas que muitas vezes caem no melhor ou pior dos casos. Vimos no caso da ordenação da nossa pequena sequência que tínhamos, como número máximo de comparações, (0,5 * 10 * (10 – 1) = 450, e este será muitas vezes o caso com qualquer lista. Em termos de trocas, no entanto, tínhamos apenas ultrapassado metade do número teórico máximo. As listas diferem umas das outras, mas, em geral, quanto maior for a lista pior será o trabalho executado pela *Bubble Sort* numa rápida ordenação. Em relação a pequenas listas, qualquer diferença de tempo será pouco significativa e pode ser ultrapassada pela simplicidade com que se pode programar a ordenação.

A PROGRAMAÇÃO DA «BUBBLE SORT»

Tendo dado uma vista de olhos no que são as ordenações e em particular como a *Bubble Sort* trabalha, é possível agora aprofundar e mostrar como se pode programar (não se esquecendo do que ficou agora dito). A seguir dá-se o esqueleto de um programa curto que usaremos para testar três tipos diferentes de ordenações. O programa consiste de:

1) Um gerador de palavras ao acaso para fornecer uma lista de cem combinações de letras sem sentido, armazenadas em ARRAY1$.

2) Uma rotina de cópia que copiará a lista original para uma segunda matriz, ARRAY2$, para que tenhamos a possibilidade de usar a mesma lista para todas as ordenações e assim comparar os resultados.

3) Uma rotina de verificação de ordem que verifica se a lista, tal como é processada por qualquer das ordenações, está em ordem correcta.

4) Uma rotina para imprimir a lista e podermos verificar visualmente a sua ordem. Pode não desejar chamar esta rotina, a não ser que lhe aconteça qualquer problema, já que inibirá o controlo dos tempos das três ordenações, que assim serão impressas no visor ao mesmo tempo.

5) Uma rotina que executará uma ordenação *Bubble Sort* numa lista desordenada, colocando-a por ordem alfabética.

Repare que neste capítulo estaremos sempre a ordenar cadeias. Para ordenar números apenas necessitamos de trocar os nomes das matrizes para matrizes numéricas. Poderá descobrir, se fizer essa alteração, que as ordenações serão mais rápidas, especialmente as que requerem um grande número de trocas. As trocas constantes de cadeias no Amstrad resultam em pausas para arrumação da memória em intervalos regulares.

Nas ordenações dadas estaremos sempre a trabalhar para ordenar as nossas listas alfabeticamente. Se desejar ordenar uma lista em sentido inverso (isto é, começando no Z e acabando no A), só necessitará de alterar as linhas que têm «<» e «>» para as relações opostas.

```
1000 REM*********************************
1010 REM CONTROL ROUTINE
1020 REM*********************************
1025 CLS
1030 GOSUB 4000
1040 GOSUB 5000 : seconds=0 : minutes=0
: hours=0 : EVERY 50,3 GOSUB 6000 : GOSU
B 8000 : t=REMAIN(3) : GOSUB 2000 : GOSU
B 7000 : PRINT hours;"/";minutes;"/";sec
onds
1120 STOP
2000 REM*********************************
2010 REM PRINT LIST
```

```
2020 REM*********************************
2025 INPUT "Display list (y/n): ";q$
2027 IF LOWER$(q$)="n" THEN RETURN
2030 FOR i=0 TO items-1
2040 PRINT array2$(i)
2070 IF i/10 =INT(i/10) THEN t$=INKEY$ :
 IF t$="" THEN 2070
2100 NEXT i
2110 RETURN
4000 REM*********************************
4010 REM GENERATE LIST
4020 REM*********************************
4025 PRINT "Generating list."
4030 ITEMS=100
4040 DIM array1$(items-1),array2$(items-
1)
4050 FOR i=0 TO items-1
4060 T$=""
4070 FOR j=1 TO 4+INT(9*RND(1))
4080 t$=t$+CHR$(65+INT(26*RND(1)))
4090 NEXT j
4100 array1$(i)=t$
4110 NEXT i
4120 RETURN
5000 REM*********************************
5010 REM COPY LIST
5020 REM*********************************
5025 PRINT "Copying list."
5030 FOR i=o TO items-1
5040 array2$(i)=array1$(i)
5050 NEXT i
5060 RETURN
6000 REM*********************************
6010 REM TIME
6020 REM*********************************
6030 seconds=seconds+1
6040 IF seconds>59 THEN seconds=0 : minu
tes=minutes+1
6050 IF minutes>59 THEN minutes=0 : hour
```

```
s=hours+1
6060 RETURN
7000 REM*********************************
7010 REM CHECK ORDER
7020 REM*********************************
7030 FOR i=0 TO items-2
7040 IF array2$(i)>array2$(i+1) THEN PRI
NT "Out of order at ";i
7050 NEXT i
7060 RETURN
8000 REM*********************************
8010 REM BUBBLE SORT
8020 REM*********************************
8025 PRINT "Bubble sort started."
8030 FOR i=items-1 TO 1 STEP -1
8040 FOR j=0 TO i-1
8050 WHILE array2$(j)>array2$(j+1)
8060 t$=array2$(j)
8070 array2$(j)=array2$(j+1)
8080 array2$(j+1)=t$
8090 WEND
8100 NEXT j
8110 NEXT i
8120 RETURN
```

Desde que tenha seguido o caminho indicado, estas linhas não devem constituir problema para si, já que elas seguem exactamente o método que utilizámos ao exemplificar manualmente o funcionamento.

A rotina de ordenação, tal como está listada, pode ser retirada do programa e usada quando quiser nas suas aplicações em que pequenas quantidades de dados necessitarem de ordenação. Só será preciso alterar os números de linha e adaptá-los ao programa, bem como os nomes das matrizes, de acordo com os que desejar utilizar.

Para utilizar o programa deverá fazer RUN e esperar que ele confirme que a lista gerada ao acaso está correctamente ordenada. Terá de esperar o tempo que medeia entre a cópia da lista a ordenar ser enviada para a ordenação e a ordenação ser concluída. Pode tentar adaptar o programa para trabalhar com listas maiores mudando o valor de ITEMS, desde que não tenha objecções em ir dar

uma voltinha enquanto a ordenação se arrasta. Se definitivamente decidiu fazer testes com listas largamente superiores a cem assuntos, pode ser útil inserir uma linha extra no princípio de cada ordenação, incorporando FRE (0), que garantirá que a colecção de lixo do Amstrad seja retirada no mesmo ponto em todas.

A «ORDENAÇÃO POR SUBSTITUIÇÃO ATRASADA» — UM ATALHO SIMPLES

Anteriormente comentei que um método utilizado por muitas pessoas seria identificar o mais elevado ou mais baixo valor da sequência e colocá-lo imediatamente no seu lugar correcto. Poderia ter escolhido começar por colocar o terceiro quadrado na posição correcta, depois o sétimo, depois o primeiro; na realidade poderia ter escolhido a ordem que quisesse. Isto é assim porque a lista tem intervalos regulares entre os valores e deste modo pode dizer-se para onde vai cada quadrado depois de se ver o seu número. Se o intervalo fosse irregular seria mais difícil começar por descobrir o quadrado correcto a colocar na terceira posição. Em tais listas, e muitas são as que têm intervalos irregulares entre os assuntos, o mais simples, se desejamos identificar a posição de um assunto sem ambiguidade, é encontrar primeiro o mais elevado ou o mais baixo, depois o segundo mais elevado ou mais baixo, e assim sucessivamente.

De facto é o que a *Bubble Sort* faz. Em cada passagem apanha o valor mais elevado que ainda não está correctamente colocado e deposita-o no local próprio. Procedendo deste modo, a ordenação também introduz alterações na ordem do resto da sequência. A questão que se põe é saber se todas estas trocas são necessárias. A resposta é negativa.

Voltemos aos cartões para examinar quantas destas trocas poderiam ter sido eliminadas. Primeiro disponha os cartões na mesma ordem que utilizámos para testar a *Bubble Sort:*

7 3 1 5 6 9 4 8 0 2

Agora arranje uma moeda pequena e coloque-a no quadrado situado mais à esquerda, o 7. A moeda representa a posição do

quadrado com valor mais elevado que encontrou. Procede-se como se indica seguidamente:

Compare o valor do quadrado que tem a moeda com o que está à sua direita. O próximo quadrado, o três, tem um valor mais baixo, por isso a moeda fica onde está.

Transfira a comparação para a casa seguinte à direita, que no nosso caso é o 1. Vá transferindo a comparação sucessivamente para a direita enquanto o valor que está a comparar for inferior ao que tem a moeda.

Quando chegar a um quadrado que tem um valor superior ao que nessa altura tem a moeda, no nosso caso o 9, desloque a moeda para esse quadrado. Depois comece a comparar o quadrado que tem a moeda com o que está à sua direita, deixando a moeda onde está se o valor for inferior e deslocando-a se o seu valor for superior. No nosso caso a moeda fica sobre o nove até ao fim da passagem dos quadrados.

Agarre no quadrado em que terminou a passagem e coloque-o na posição de reserva. Coloque o cartão que tem a moeda no espaço vazio e depois tire o quadrado que está na posição de reserva e coloque-o no espaço do que tinha a moeda.

Coloque a moeda no quadrado mais à esquerda e repita todo o processo apenas com a excepção de que, tal como na *Bubble Sort,* pode reduzir-se o número de comparações em cada passagem, levando em conta o facto de que as passagens anteriores reduziram a parte da sequência que está desordenada uma a uma.

Se seguiu o processo acima descrito deverá obter as sequências seguintes em cada passagem:

```
7 3 1 5 6 9 4 8 0 2 : POSIÇÃO INICIAL
7 3 1 5 6 2 4 8 0 9
7 3 1 5 6 2 4 0 8 9
0 3 1 5 6 2 4 7 8 9
0 3 1 5 4 2 6 7 8 9
0 3 1 2 4 5 6 7 8 9
0 3 1 2 4 5 6 7 8 9
0 2 1 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
```

Se sentir que «agarrou» o método, então a verdadeira lição desta ordenação está «madura» para ser aprendida. Por agora vamos contar as trocas e comparações da mesma maneira que fizemos para a *Bubble Sort.* Neste caso, e para sermos honestos em re-

lação à *Bubble Sort,* devemos contar não só o número de vezes que a moeda teve de ser movimentada mas também a vez que correspondeu à colocação da moeda no início no quadrado mais à esquerda. Os números tais como eu os conto são como segue:

1) 9	Comparações	1	Trocas	2	movimentações
2) 8	Comparações	1	Trocas	2	movimentações
3) 7	Comparações	1	Trocas	1	movimentações
4) 6	Comparações	1	Trocas	4	movimentações
5) 5	Comparações	1	Trocas	3	movimentações
6) 4	Comparações	0	Trocas	3	movimentações
7) 3	Comparações	1	Trocas	2	movimentações
8) 2	Comparações	1	Trocas	2	movimentações
9) 1	Comparações	0	Trocas	2	movimentações

Totais: 45 comparações 7 trocas 21 movimentações

Fizemos claramente maior economia no conjunto de trocas e movimentações da moeda. Mesmo com a simulação manual, que consiste em movimentar a moeda, é muito mais simples que movimentar dois quadrados. Quando o método é computadorizado, torna-se significativa a economia que resulta de trocar apenas o valor da variável (a que é usada para registar a posição do valor mais alto até aí encontrado), comparada à tripla operação de trocar duas cadeias.

Os números para a «Ordenação por substituição atrasada» no caso pior tornam a posição ainda mais clara. Para o caso pior de listas (pelo menos nesta ordenação), o número de comparações será, tal como com a *Bubble Sort,* $0{,}5 * N * (N-1)$. O maior número de trocas possível é no entanto só $N-1$, ou seja, 9, no nosso caso. O maior número possível de movimentações consegue-se agora quando a lista está quase em ordem. Por exemplo, uma passagem da lista quando está em perfeita ordenação conduzirá a nove movimentações ponteiro em direcção ao valor mais alto. Se registarmos o número de movimentações feitas em cada passagem e o número de comparações a fazer na mesma passagem, então todas as vezes que os números forem os mesmos temos a certeza de que a parte inferior da sequência está em perfeita ordem. Se arranjarmos os nossos quadrados com a ordenação

1 2 3 4 5 6 7 8 9 0

descobriremos que há 45 movimentações, que parece suspeitamente a nossa velha amiga $0{,}5 * N * (N-1)$ outra vez. Podemos ainda concluir que o pior caso para a «Ordenação por substituição atrasada» será:

Para cem assuntos: 4950 comparações, 99 trocas e 4950 movimentações.

Para mil assuntos: 499 500 comparações, 999 trocas e 499 500 movimentações.

Se for verdade que as movimentações são muito mais rápidas que as trocas, então a perda de 500 000 trocas para o número mais elevado deve produzir uma considerável economia de tempo. Isto só poderá ser testado comparando as duas ordenações em acção, e, para tal, do que necessitamos é de introduzir a rotina seguinte, que encaixa no programa já dado:

```
1050 GOSUB 5000 : seconds=0 : minutes=0
: hours=0 : EVERY 50,3 GOSUB 6000 : GOSU
B 9000 : t=REMAIN(3) : GOSUB 2000 : GOSU
B 7000 : PRINT hours;"/";minutes;"/";sec
onds
9000 REM********************************
9010 REM DELAYED REPLACEMENT SORT
9020 REM********************************
9025 PRINT "Starting DR sort"
9030 FOR i=items-1 TO 1 STEP -1
9040 biggest=0
9050 FOR j=1 TO i
9060 IF array2$(j)>array2$(biggest) THEN
 biggest=j
9070 NEXT j
9080 t$=array2$(i)
9090 array2$(i)=array2$(biggest)
9100 array2$(biggest)=t$
9110 NEXT i
9120 RETURN
```

A execução repetida da ordenação demonstrará, numa estimativa grosseira, que obteremos uma economia da ordem dos dois terços em relação à simples *Bubble Sort* no caso de cem assuntos. Ao mesmo tempo fica claro que o que na verdade fizemos foi elaborar o método, retirando algumas trocas redundantes.

A «ORDENAÇÃO DE SHELL-METZNER»: COMO UTILIZAR A POTÊNCIA DE DOIS

Para se conseguir uma capacidade de ordenação real deveremos deixar para trás o relativamente seguro mundo das repetidas passagens de dados de uma maneira ordenada à procura dos valores mais baixos e mais altos. Com a busca binária devemos lançar-nos num método aparentemente sem algoritmo, com a confiança de que os métodos binários trarão alguma ordem ao que aparentemente parece o caos. Neste método de ordenação serão feitas trocas aparentemente desvairadas nos estágios iniciais e, no entanto, a ordem correcta aparecerá no meio da confusão em muito menos tempo (para listas substanciais) que para qualquer dos métodos que empregámos.

A compreensão do andamento da ordenação não vai ser fácil; mas, como sempre, tentaremos executá-la nos nossos quadrados desordenados. Desta vez, contudo, serão necessárias quatro moedas de diferentes valores e um pedaço de papel para acrescentar aos quadrados. Chamaremos às quatro moedas, por ordem ascendente do seu valor, A, B, C, D. No pedaço de papel registaremos os valores sucessivos de uma variável a que chamaremos GAP (intervalo).

O método básico da ordenação é começar a fazer trocas entre assuntos que têm um intervalo entre eles da maior potência de 2 que encaixa na lista de dados a ordenar. Quando tivermos esgotado todas as trocas desta potência de 2, procederemos às trocas correspondentes à potência que tem metade daquele valor, e assim sucessivamente. No nosso caso o valor do intervalo em que vamos fazer trocas é 8; nesse caso, escreva no pedaço de papel «GAP = 8». Agora coloque as moedas A e B próximas do quadrado mais à esquerda (7). Durante a ordenação usaremos a moeda A (a de valor mais baixo) para indicar o quadrado mais à esquerda dos dois que estamos a trocar e a moeda «B» para registar o espaço percorrido na passagem pela matriz para cada valor do intervalo (GAP). A moeda «C» é colocada agora no quadrado que está mais afastado para a direita a partir da qual uma troca de GAP pode ser feita. No nosso caso será a posição dois, já que dois mais o valor de GAP leva-nos para o extremo mais afastado da nossa matriz de quadrados. A moeda «D» será movida ao longo da ordenação para mostrar a posição do quadrado que está GAP lugares à esquerda do quadrado com a moeda «A».

Agora siga o procedimento que a seguir se expõe:

1) Coloque os quadrados na ordem 7 3 1 5 6 9 4 8 0 2, tal como antes.

2) «A» está na posição 1, então «D» deve ser colocada na posição 1 × GAP (= 9). Estes dois quadrados são o 7 e 0; que são então trocados. As moedas ainda não são deslocadas.

3) Por pateta que pareça nesta altura, examinamos se é possível mover «A» GAP lugares para a esquerda. Explicarei isto mais tarde. De momento retenha na ideia o que fizemos, mesmo que ache impossível desloque «B» um lugar para a direita.

4) Sempre que «B» é deslocada um lugar para a direita, «A» é automaticamente deslocada para a mesma posição; por isso coloque «A» na posição 2 com «B». Esta é também a posição ocupada por «C».

5) «D» é deslocada agora para A + GAP, que significa posição 10. Os quadrados são 3 e 2; então são trocados.

6) Mais uma vez consideramos a hipótese de deslocar «A» GAP lugares para a esquerda. Não podemos; por isso, deslocamos «B» um lugar para a direita.

7) Reparará agora que «B» foi movimentado em avanço em relação a «C». Isto significa que a primeira passagem de dados está completa. Sempre que uma passagem está completa devemos reduzir o valor do GAP para metade e recolocar as moedas. Risque 8 no pedaço de papel e substitua-o por 4. Desloque «B» de novo para a posição mais à esquerda, seguida de «A». «C» deve ser colocada na posição 10 − GAP, que é agora 6, representando a posição mais à direita a partir da qual uma troca de GAP(= 4) pôde ser feita.

8) Agora começamos outra vez a tentar fazer trocas. Desloque «D» para «A» + GAP, que será a posição 5. Os quadrados são 0 e 6, então não podem ser trocados. Sempre que uma troca não pode ser feita, «B» é deslocada um lugar para a direita, com «A» seguindo-a para a sua nova posição. «D» é deslocada para a posição 6 (A + GAP).

9) O processo de deslocar B, A e D repete-se até que B e A estejam na posição 6. Até aqui foi impossível fazer quaisquer trocas de GAP 4.

10) Com B e A na posição 6, os dois quadrados indicados por A e D têm os valores de 9 e 3, então podem ser trocados.

11) Como foi feita uma troca, movimentamos «A» GAP lugares para a esquerda. Desta vez foi possível e por isso o fizemos. «A» está agora na posição 2, então movimentamos «D» para

«A + GAP», que dá 6. Os dois quadrados são 2 e 3 e não podem ser trocados. A razão por que tentámos isto é que, sempre que um quadrado é movimentado para baixo na sequência, temos de testar quando é que ele podia ter sido trocado se estivesse na nova posição desde o início. Se a troca foi possível teríamos tentado mover «A» GAP lugares para a esquerda outra vez. Enquanto isto acontece, «B» fica onde está.

12) Quando se atinge um ponto em que não é possível a troca ou em que não podemos movimentar «A» GAP lugares para a esquerda, movimenta-se «B» e «A» logo atrás. Isto significa que «B» está à frente de «C» e a passagem está por isso terminada.

13) Reduza em metade o valor de GAP, para 2. Substitua «A» e «B» na posição 1. Coloca-se «C» na posição 10 − GAP (8). Coloca-se D na posição A + GAP (3).

14) As trocas podem tentar-se sem sucesso até que «A» e «B» alcancem a posição 4, com «D» em 6. Troque o 5 e o 3. «A» movimenta-se de volta à posição 2, mas não se pode fazer outra troca: por isso, desloque «B» um lugar para a direita.

15) Agora com «B» e «A» na posição 5 pode trocar-se o 6 e o 4. Não são possíveis trocas quando «A» for desviado para a esquerda, então desloca-se «B».

16) Não são possíveis trocas nas posições 6, 7, 8, por isso a passagem termina sem que sejam possíveis trocas posteriores.

17) Reduz-se o GAP a 1, «A» e «B» são de novo colocados na posição 1, «C» é colocado na posição 10 − GAP(9) e «D» colocado na posição A + GAP(2).

18) De facto, para GAP = 1, deslocando os ponteiros produzirá uma passagem final do tipo da utilizada pela BUBBLE SORT. Neste momento já deve ser capaz de executar as movimentações neste caso, sem mais ajudas.

19) Se tudo correu bem, a lista está agora ordenada, uma indicação disto é o facto de que GAP teria de ser menor que 1 para a próxima passagem.

20) Como retoque final para ilustrar por que é que o ponteiro «A» é movido para a direita sempre que se faz uma troca, você pode gostar de tentar fazer aquele final, uma passagem do tipo *Bubble Sort* em cinco dos quadrados dispostos da forma 0 1 3 4 2. Tal como fizemos mais atrás, utilize as moedas e faça GAP = 1. Descobrirá que não são possíveis trocas até que «B» esteja na posição 4 e que parando aqui não se consegue a ordenação final dalista. Deslocando «A» GAP lugares para a esquerda (para a posição 3) fornece a última troca que é necessária para colocar 2 na posição correcta.

Para o caso da «Ordenação de Shell-Metzner» não tentaremos analisar o número de trocas, deslocamentos e comparações, em parte porque seria muito complexo e também porque, para esse número de assuntos, nos diria muito pouco, porque as vantagens deste método de ordenação só se tornam aparentes quando grandes corpos de dados são ordenados. Isto não significa que a «Ordenação de Shell-Metzner» não possa ser utilizada para pequenas quantidades de dados; trata-se, simplesmente, de saber até que ponto compensa o trabalho de programação adicional para um pequeno aumento da velocidade.

Quando introduzir a «Ordenação de Shell-Metzner» no programa teste de ordenação, na forma da rotina que se segue, descobrirá que para grandes corpos de dados a economia é dramática. Em geral, o tempo necessário para a *Bubble Sort* varia aproximadamente de acordo com a fórmula (N*N)/2, o que significa que os aumentos de N são multiplicados por si próprios. Com a «Ordenação de Shel-Metzner», o tempo necessário para a ordenação de N assuntos acompanha a fórmula:

N*(LOG N/LOG 2)

Isto significa que, no pior caso, se a ordenação *Bubble Sort* para dez assuntos demorar 1 segundo, cem assuntos demorarão 100 segundos e mil levarão 10 000 segundos.

No caso da «Ordenação de Shell-Metzner», também para o pior caso, se a ordenação de dez assuntos demorar 1 segundo, então cem assuntos demorarão 20 segundos e mil necessitarão de 300 segundos.

Não pretendo que estes números sejam rigorosos, mas ilustram as imensas diferenças resultantes para o Amstrad quando cresce o número de dados a ser ordenado. Pode verificar por si próprio brincando com o tamanho da lista a ordenar no programa teste.

```
1060 GOSUB 5000 : seconds=0 : minutes=0
: hours=0 : EVERY 50,3 GOSUB 6000 : GOSU
B 10000 : t=REMAIN(3) : GOSUB 2000 : GOS
UB 7000 : PRINT hours;"/";minutes;"/";seconds
10000 REM*******************************
10010 REM SHELL-METZNER
10020 REM*******************************
10025 PRINT "Starting S_M sort"
```

```
10030 gap=2^(INT(LOG(items-1)/LOG(2))+1)
10040 WHILE gap>1
10050 gap=gap/2
10060 b=0
10070 c=items-gap-1
10080 WHILE b<=c
10085 a=b
10087 WHILE a>=0
10090 IF array2$(a)>array2$(a+gap) THEN
GOSUB 10200 ELSE a=-1
10140 WEND
10150 b=b+1
10160 WEND
10170 WEND
10180 RETURN
10200 t$=(array2$(a))
10210 array2$(a)=array2$(a+gap)
10220 array2$(a+gap)=t$
10230 a=a-gap
10240 RETURN
```

Na prática, quando aplicar à lista de cem assuntos a «Ordenação de Shell-Metzner» descobrirá uma economia típica de dois terços em relação à de *Bubble-Sort*. Aumentando o tamanho da lista, no entanto, obtêm-se resultados mais dramáticos. Para uma lista de duzentos assuntos a «Ordenação de Shell-Metzner» pode precisar apenas de 15% do tempo levado pela *Bubble-Sort*.

CONCLUSÃO

Ainda não acabámos com o tópico da ordenação — na verdade, ainda mal começámos. Existem outros tipos de ordenações que podem trazer economias significantes, ainda que não astronómicas, em relação à de Shell-Metzner. Em relação a elas há o problema de quase invariavelmente necessitarem de um espaço de armazenamento em paralelo para que os dados possam ser baralhados entre a lista principal e uma ou mais listas paralelas. Para fins práticos,

isto significa que para o tipo de quantidades em que elas começam a ser vantajosas em relação à de Shell-Metzner, desperdiçam tanto espaço de memória que não têm utilidade prática nos computadores domésticos. Utilizando as ordenações dadas neste capítulo será capaz de manusear quase todas as quantidades de dados, da mais pequena à maior, com métodos apropriados à tarefa e, muitas vezes, com um aumento de velocidade dramático em relação à velocidade dos seus programas.

CAPÍTULO 8

Estruturas de dados — I

Um dos axiomas deste livro é que a maioria dos programas úteis armazenam dados. O armazenamento de dados e o seu processamento constituem as melhores vantagens de um microcomputador. São áreas em que excedem a utilização de qualquer outro método. No entanto, algumas vezes, a maneira fácil com que o micro manuseia a informação pode ser uma desculpa para o desmazelo no processo de decisão da maneira como essa informação terá de ser armazenada. Tudo parece correr bem enquanto o programa pode conservar na memória os dados necessários.

A desvantagem deste tipo de aproximação é que uma estrutura de dados errada pode originar um enorme buraco nas capacidades do programa, tornando-o vagaroso, reduzindo a quantidade de material que pode ser usado ou complicando desnecessariamente a estrutura funcional do programa. Neste capítulo examinaremos algumas das muitas maneiras de estruturar dados para responder às necessidades do programa e maximizar a velocidade e espaço de armazenamento.

ESTRUTURAS DE DADOS SIMPLES: A MATRIZ À VONTADE DO FREGUÊS

De longe, a mais simples estrutura que pode ser usada para armazenar qualquer espécie de informação é uma matriz em que as dimensões são adequadas à maneira como a informação se reparte.

Um exemplo pode ser o programa destinado a registar o movimento, lucro, taxa de investimento para uma série de companhias. Neste caso pode ser declarada uma matriz, ARRAY(X,3), em que X é o número de companhias e dados a introduzir como se mostra na rotina seguinte:

```
100 INPUT «COMPANY NUMBER: »;CY
110 INPUT «TURNOVER: »;ARRAY(CY,0)
120 INPUT «PROFIT: »;ARRAY(CY,1)
130 INPUT «TAX: »;ARRAY(CY,2)
140 INPUT «INVESTMENT: »;ARRAY(CY,3)
```

Num programa de ficheiro para armazenar nomes, endereços, e números de telefones pode ser declarada uma variável de cadeia como ARRAY$(500,2), com as entradas dadas por uma rotina similar à apresentada atrás.

Estas estruturas podem ser tão complexas quanto o programa necessite. Por exemplo, pode ser que, tal como o programa dado acima para o caso dos números da companhia, a necessidade seja registar todos os números durante um período de cinco anos. Neste caso a matriz seria declarada como ARRAY(X,3,4) e cada uma das categorias separadas teria um ciclo para permitir cinco entradas, de tal modo que a linha 110 torna-se por si só uma rotina:

```
110 FOR I=0 TO 4
112 PRINT «TURNOVER FOR»;1977+I;«:»:INPUT
ARRAY(CY,0,1)
114 NEXT I
```

A vantagem destas matrizes pronto-a-vestir é que tornam o armazenamento e a retirada de dados uma tarefa imensamente directa: todas as coisas têm um lugar sem ambiguidade e encontrar um item dos dados é uma tarefa que só necessita de conhecer o número da companhia, a categoria da informação (por exemplo, taxa) e o ano. Os dados neste tipo de matriz são também fáceis de obter de outras maneiras. Por exemplo, se o utilizador desejasse saber como cada companhia obteve resultados do ponto de vista do lucro em 1978, um ciclo simples como:

```
100 FOR I=0 TO ITEMS : PRINT ARRAY(I,2,1) : NEXT
```

extrairia a informação sem problemas.

Outra vantagem das matrizes pronto-a-vestir é a maneira como todos os conjuntos de matrizes podem ser interligados contendo conjuntos de informação diferentes mas paralelos. No caso da nossa matriz-exemplo bastaria simplesmente declarar outra matriz contendo os nomes das companhias e estes nomes poderiam ser acedidos exactamente pela mesma variável que foi utilizada para especificar o conjunto de números a ser impressos. Em programas que implicam um manuseamento complexo de dados não é pouco comum ter uma gama de diferentes matrizes, armazenando todas informação paralela. Nos casos do armazenamento de informações diversas em que a maior parte da estrutura é semelhante, por exemplo, dizendo toda respeito aos meses do ano, então as matrizes tipo pronto-a-vestir podem ser o único meio eficaz de controlar o número de variáveis que têm de ser usadas para obter a informação de muitos lugares diferentes. Se todos os dados fossem relacionados com os meses do ano, então, nalguns casos, uma única variável, representando o mês em questão, seria suficiente para tirar informação importante de todas as matrizes.

No entanto as matrizes pronto-a-vestir têm desvantagens. Uma delas é a quantidade de memória de que necessitam, mesmo no caso de uma matriz aparentemente inócua. Uma matriz numérica com as dimensões (100,10,10) pode não parecer muito assustadora em termos de dimensão, mas necessitaria de 50 000 *bytes* de memória para a conter — mais do que existe disponível em BASIC no Amstrad. Para calcularmos o espaço de memória necessário para a matriz bastará simplesmente multiplicar uns pelos outros os números que estão entre parênteses na definição da matriz e depois multiplicá-los ainda por:

2 no caso de uma matriz de números inteiros;
3 no caso de uma matriz de cadeias;
5 no caso de uma matriz numérica de vírgula flutuante.

Recordamos que, no caso de matrizes de cadeias, o número a que se chega é um valor estimado e tudo o que ponha dentro da matriz será adicionado à quantidade de memória necessária. No caso de matrizes numéricas, todos os elementos são colocados a zero; por isso, não haverá aumento na dimensão da memória necessária se mudarmos os elementos.

Quando se declaram matrizes grandes e complexas é, portanto, vital determinar se todo o espaço da matriz é necessário. Há muitas ocasiões, na introdução de informação em ficheiros, em que um

grande número de categorias necessitam de ser declaradas, mas em que nem todos os assuntos do ficheiro vão ter informação dentro de cada categoria. Tais matrizes são chamadas «escassas», sendo muito reduzido o número de aplicações que podem comportar este tipo de memória desperdiçada.

A conclusão disto tudo é que pode dispor de uma estrutura de dados exactamente à medida da sua informação — e se comportar toda a informação que pensa necessitar sobre esse assunto, então aproveite-a! Sem dúvida simplificará imenso o programa. Muitas vezes, no entanto, cairá em aplicações em que o número absoluto de assuntos ou a complexidade da estrutura é tal que a matriz pronto-a-vestir desperdiça demasiado e terá de examinar algum dos outros tipos de armazenamento deste capítulo.

ESTRUTURAS DE DADOS PARA NÚMEROS

Números de um só «byte» em matrizes de inteiros

O Amstrad fornece um meio muito económico de armazenamento para o tipo de números mais usados nos programas que manuseiam dados. Muitas vezes, a maior parte dos valores utilizados em tais programas com o fim de controlar o programa são números inteiros (sem vírgula), num alcance mais restrito reflectindo coisas tais como dimensão das matrizes e comprimento das cadeias. Para a maior parte deste tipo de valores, utilizar uma matriz numérica normal é uma absoluta perda de memória, porque cada elemento de uma matriz de vírgula flutuante necessita de cinco *bytes* de memória, comparado com os dois necessários para um elemento de uma matriz inteira.

Matrizes inteiras podem, por isso, ser usadas para reduzir a quantidade de espaço necessário para armazenar pequenos valores numéricos, até mesmo armazenando mais que um valor numérico em cada elemento da matriz. Isto é possível porque cada elemento numa matriz de inteiros pode armazenar um número entre -32768 e $+32768$, efectivamente um domínio de 65536. O domínio disponível representa o que pode ser armazenado num número de 16 *bit* que utiliza o *bit* mais à esquerda para o sinal do número (+ ou −). O que é importante para nós é que um elemento numa matriz de inteiros comporta um número até $256 \times 256 - 1$ (não considerando a

parte negativa do domínio); por isso, se tivermos dois números no domínio 0 – 255, multiplicando um deles por 256 e adicionando o segundo, ficam efectivamente guardados dois números dentro de um número maior. Por exemplo, se quisermos guardar 237 e 76 desta maneira, podemos fazer a seguinte linha em BASIC:

```
100 NN = 256*237 + 76 : REM A RESPOSTA É 60748
```

Contudo, não é assim tão simples. Supondo que NN vai ser definida como um inteiro, o domínio dos números que podem ser armazenados é de – 32768 até + 32768, em vez de 0 – 65535; por isso, se o número a ser armazenado for superior a 32767, primeiro terá de ser tornado negativo subtraindo 65536, por forma a tornar a linha de BASIC em:

```
100 TEMP = 256*237 + 76 : IF TEMP > 32767 THEN
TEMP = TEMP – 65536
110 NN% = TEMP
```

Para descodificar um par de números que foi armazenado num elemento de uma matriz de inteiros como fizemos atrás, precisamos de outra linha de BASIC:

```
100  NN = ARRAY%(X) + 65536*(ARRAY%(X) < 0)  :  N1
= INT(NN/256): N2 = NN – 56*N1
```

Para guardar números no domínio 0 – 255 pode usar-se uma rotina como a seguinte:

```
1000 REM*********************************
1010 REM SINGLE BYTES IN INTEGER ARRAYS
1020 REM*********************************
1030 DIM array%(39)
1040 more$="y"
1050 WHILE more$<>"n"
1060 CLS
1070 pp=-1 : WHILE pp<0 OR pp>79
1080 INPUT "Position: ";pp
1090 WEND
1100 nn=-1 : WHILE nn<0 OR nn>255
1110 INPUT "Value: ";nn
```

```
1120 WEND
1125 p2=INT(pp/2)
1130 temp=array%(p2)-65536*(array%(p2)<0
)
1140 IF pp AND 1 THEN temp=256*INT(temp/
256)+nn ELSE temp=temp-256*INT(temp/256)
+256*nn
1150 array%(p2)=temp+65536*(temp>32767)
1160 INPUT "More (y/n): ";more$
1170 more$=LOWER$(more$)
1180 WEND
1190 :
```

A linha 1080 extrai o valor actual do elemento correcto da matriz ARRAY% e lida com o problema dos números negativos. A posição correcta em ARRAY% é determinada por PP/2, por forma a que os dois primeiros valores serão guardados no elemento 0 (zero), os dois segundos no elemento 1 e assim sucessivamente. Na linha 1140 o «IF PP AND 1» detecta os assuntos com numeração ímpar, porque PP AND 1 só será verdade para valores ímpares de PP. Se o valor de PP é ímpar, então o valor a ser inserido num dado elemento da matriz é o menor de dois valores que serão armazenados nesse elemento. Além disso conservamos o valor superior na forma de 256*INT(NN/256), de outro modo a subtracção de uma quantidade equivalente conserva o valor inferior e elimina o superior.

«Desembalar» uma estrutura como esta é mais simples que criá-la:

```
2000 REM*********************************
2010 REM UNPACK SINGLE BYTES
2020 REM*********************************
2030 CLS
2040 FOR i=0 TO 79
2050 nn=array%(INT(i/2))-65536*(array%(I
NT(i/2))<0)
2060 IF i AND 1 THEN nn=nn-256*INT(nn/25
6) ELSE nn=INT(nn/256)
```

```
2070 PEN 3 : PRINT " "; : PRINT USING "#
#";i; : PRINT ":";
2080 PEN 1 : PRINT USING "######";nn;
2090 NEXT i
2100 STOP
2110 :
```

A vantagem deste método, em termos de economia de memória, é óbvia, embora se deva notar que tem os seus custos em termos do tempo que se gasta a armazenar e a extrair valores, se estiverem constantemente a serem chamados pelo programa. Haverá uma vantagem em termos de velocidade no carregamento e gravação em fita ou disco, pois apenas necessita de ser processado metade do número de elementos.

Armazenamento directo na memória disponível

Para os que gostam, ou precisam, de utilizar todos os gramas de memória disponível, existe a possibilidade de armazenar directamente na memória, utilizando PEEK e POKE. Não será uma técnica que se goste de aplicar frequentemente, mas dá uma flexibilidade total em relação às estruturas a empregar e, adicionalmente, permite que os dados sejam armazenados como uma área de memória usando a opção «,B» e a instrução SAVE. O método envolve a utilização da instrução MEMORY para tornar mais baixa a área da memória normalmente usada pelo BASIC e quase todas as estruturas de dados ordenadas de números, e até de cadeias, se podem armazenar nela, desde que esteja disposto a ter cuidado na preparação da estrutura e variáveis a serem usadas no controlo de acesso aos dados.

A forma mais simples de dados a armazenar numa área livre de memória são os valores numéricos de um único *byte,* ou seja, números no domínio 0–255. Para simular uma matriz de 50 por 50, por exemplo, os assuntos podem ser armazenados por uma instrução do tipo:

```
100 POKE AREA_START+50*X+Y,NN
```

em que X é o número de linha da matriz, contando a partir de zero, e Y é o número de coluna, contando sempre a partir de zero, e NN

será o número a armazenar. Os valores são retirados por uma instrução que é a imagem desta num espelho, utilizando agora PEEK:

```
100 NN=PEEK(AREA_START+50*X+Y)
```

Estruturas de matrizes mais complexas, com mais que duas dimensões, podem ser criadas desde que se observem as regras seguintes:

1) Faça como se a matriz fosse declarada em BASIC, isto é, A(20,10,5).

2) Começando da direita, calcule o número de elementos em cada unidade de cada uma das dimensões. No exemplo dado acima há um elemento em cada uma das unidades da dimensão mais à direita, cinco em cada uma das do segundo a partir da direita e cinquenta no elemento mais à esquerda.

3) Quando especificar uma localização na hipotética matriz, primeiro ponha a localização da área de memória que está a ser utilizada, depois multiplique cada dimensão específica pelo número de elementos que calculou e adicione ao elemento a localização inicial. Então o elemento 14,7,3 no exemplo dado seria localizado em AREA_START+50*14+5*7+3.

Podem ser definidas estruturas mais complexas para números que necessitem de dois ou mais *bytes,* embora seja necessário utilizar técnicas como as descritas na secção anterior para dividir os números em *bytes* únicos. Por exemplo, utilizando números de dois *bytes,* a regra deveria ser multiplicar por dois o endereço obtido das dimensões e depois fazer POKE no interior do endereço resultante e no que se segue, os dois *bytes* perfazendo o número a ser armazenado. Para armazenar um número no domínio 0–65535 na localização 14,7,3 numa matriz de dois *bytes* por elemento, como no exemplo dado acima, a rotina seguinte será suficiente:

```
3000 REM*********************************
3010 REM STORING IN FREE MEMORY
3020 REM*********************************
3030 realtop=HIMEM
3040 newtop=HIMEM-2000
3050 MEMORY newtop
3060 more$="y"
3070 WHILE more$="y"
```

```
3080 CLS
3090 INPUT "Input or output (i/o): ";q$
3100 GOSUB 3310
3110 done=0
3120 WHILE LOWER$(q$)="i" AND done=0
3130 PRINT
3140 INPUT "Value to be input (0-65535):
 ";nn
3150 n1=INT(nn/256) : n2=nn-256*n1
3160 POKE place,n1 : POKE place+1,n2
3170 done=1
3180 WEND
3190 WHILE LOWER$(q$)="o" AND done=0
3200 PRINT
3210 PRINT "Value at that position is: "
;256*PEEK(place)+PEEK(place+1)
3220 PRINT : PRINT "Press any key to con
tinue."
3230 t$=INKEY$ : IF t$="" THEN 3230
3240 done=1
3250 WEND
3260 PRINT : INPUT "More (y/n): ";more$
3270 more$=LOWER$(more$)
3280 WEND
3290 MEMORY realtop
3300 STOP
3310 :
3320 PRINT : PRINT "There are three dime
nsions to the       simulated array: 20,
10,5" : PRINT
3330 INPUT "Position in dimension 1 (1-2
0): ";p1
3340 INPUT "Position in dimension 2 (1-1
0): ";p2
3350 INPUT "Position in dimension 3 (1-5
): ";p3
3360 place=(50*p1+5*p2+p3)*2
3370 RETURN
3380 :
```

Números em cadeias

Já vimos, no cap. 3, que armazenar dados em cadeias de comprimento variável pode ser ao mesmo tempo rápida e económica. Utilizando este tipo de método, novos itens de dados podem ser adicionados no começo, meio e fim de um bloco de dados, sendo automaticamente recolocados todos os outros assuntos que estão armazenados. Também vimos que os dados não ficam limitados ao máximo de 255 *bytes* imposto pelo comprimento máximo possível de uma cadeia.

Esta descoberta não é só relevante para o problema de armazenar pequenos assuntos em cadeias de dados. As cadeias podem muitas vezes ser muito úteis para armazenarem valores numéricos. O motivo que torna isto possível é que todos os caracteres pertencentes ao Amstrad têm um único valor chamado «valor ASCII», um número pertencente ao domínio 0 – 255. É nesta forma numérica que o Amstrad os armazena. O BASIC fornece ao programador duas funções tradutoras, ASC e CHR$, entre os números e os caracteres, por forma a que os valores possam ser traduzidos em caracteres, armazenados na forma de uma cadeia e subsequentemente reconvertidos novamente em números.

As técnicas para transformarem um valor num número são seguidamente ilustradas:

```
100 NN = 65
110 A$ = CHR$(NN)
120 PRINT A$
```

Se executar estas linhas, o que lhe vai aparecer é a letra «A», cujo valor em código ASCII é 65. A retroversão pode ser ilustrada, adicionando a linha seguinte:

```
130 PRINT ASC(A$)
```

Disto podemos concluir que a função CHR$ cria um carácter com o valor do código especificado, enquanto ASC extrai o valor do código de uma cadeia de caracteres existente. Podemos combinar as duas para executar formas flexíveis e económicas de armazenar pequenos valores numéricos:

```
100 A$ = «  »
110 INPUT NN
```

```
120 IF NN > = 0 THEN A$ = A$ + CHR$(NN):GOTO 110
130 FOR I = 1 TO LEN (A$)
140 PRINT ASC(MID$(A$,I))
150 NEXT I
```

Esta pequena rotina permitir-lhe-á introduzir até 255 números entre 0 e 255. Se introduzir um número negativo, a segunda metade da rotina actuará e imprimirá os dados na ordem em que eles foram introduzidos. Note que, na linha 140, a função MID$ não especifica que a parte da cadeia a partir da qual um valor ASC será extraído tem só um carácter. MID$(A$,I) significa toda a cadeia A$ começando em I. Isto pode fazer-se porque a função ASC trabalha sempre, e só, sobre o primeiro carácter de qualquer cadeia.

A utilidade desta técnica reside no facto de o manuseamento das cadeias não nos limitar à adição de caracteres no extremo duma cadeia existente. A linha 120 poderia muito simplesmente ter sido:

```
120 IF NN > = 0 THEN A$ = CHR$(NN) + A$:GOTO 110
```

A rotina poderia ter colocado os números por ordem inversa, o que teria sido muito difícil numa matriz sem ter de constantemente desviar dados na matriz.

Igualmente podíamos usar algumas técnicas retiradas do cap. 3 para inserir valores em qualquer parte da cadeia existente:

```
4000 REM*********************************
4010 REM SINGLE BYTE NUMBERS IN STRINGS
4020 REM*********************************
4030 a$=""
4040 nn=0
4050 WHILE nn>=0
4060 CLS
4070 INPUT "Number to be inserted (0-255
): ";nn
4080 INPUT "Position in string: ";pp
4090 IF pp<1 THEN pp=1
4100 IF nn>=0 THEN a$=LEFT$(a$,pp-1)+CHR
$(nn)+MID$(a$,pp)
4110 WEND
4120 FOR i=1 TO LEN(a$)
```

```
4130 PEN 3 : PRINT " "; : PRINT USING "#
#";i; : PRINT ":";
4140 PEN 1 : PRINT USING "######";ASC(MI
D$(a$,i));
4150 NEXT i
4160 STOP
4170 :
```

Quando for necessário considerar valores superiores a 255 não há vantagem, do ponto de vista de economia de memória, na utilização de cadeias para guardar tais valores; mas a flexibilidade que se obtém com a capacidade de introduzir valores no meio de dados existentes, sem desviar nada, pode ainda tornar atractiva a utilização de cadeias. A rotina seguinte inserirá um número de dois *bytes* (0 – 65535) em qualquer parte de uma cadeia existente:

```
5000 REM*********************************
5010 REM TWO BYTE NUMBERS IN STRINGS
5020 REM*********************************
5030 a$=""
5040 nn=0
5050 WHILE nn>=0
5060 CLS
5070 INPUT "Number to be inserted (0-655
35): ";nn
5080 n1=INT(nn/256) : n2=nn-256*n1
5090 PRINT "Position 1 to ";LEN(a$)/2+1;
": "; : INPUT pp
5100 IF pp<1 THEN pp=1
5110 pp=pp*2
5120 IF nn>=0 THEN a$=LEFT$(a$,pp-2)+CHR
$(n1)+CHR$(n2)+MID$(a$,pp-1)
5130 WEND
5132 OPENOUT "temp" : FOR i= 1 TO LEN(a$
) : PRINT #9,ASC(MID$(a$,i)) : NEXT i :C
LOSEOUT
5134 a$=""
5136 OPENIN "temp" : WHILE NOT EOF : INP
UT #9,t : a$=a$+CHR$(t) : WEND : CLOSEIN
```

```
5140 FOR i=1 TO LEN(a$) STEP 2
5150 PEN 3 : PRINT " "; : PRINT USING "#
#";INT(i/2)+1; : PRINT ":";
5160 PEN 1 : PRINT USING "######";256*AS
C(MID$(a$,i))+ASC(MID$(a$,i+1));
5170 NEXT i
5180 STOP
5190 :
```

As técnicas descritas no cap. 3 permitem apagar dados, e as técnicas de cadeias múltiplas do mesmo capítulo podem ser empregues para aumentar a capacidade em relação a uma única cadeia. Isto será discutido mais adiante neste capítulo, na secção de matrizes-ponteiro.

De tudo isto pode ver-se que armazenar números em cadeias é uma opção real para os programas que necessitam de inserir ou apagar valores regularmente. Neste ponto, a desvantagem principal consite em nem todos os caracteres poderem ser armazenados praticamente em fita ou disco. O carácter CHR$(0), isto é, o carácter com um código ASCII zero, é usado pelo sistema como marca de fim de ficheiro e, se for encontrado nos ficheiros de dados do tipo dos criados por um programa de BASIC, o sistema assume que o ficheiro terminou, mesmo se no ficheiro houver mais dados para encontrar. Por causa disto, quando se grava uma cadeia de caracteres contendo o valor zero, a rotina de gravação deve incluir algo do tipo:

```
100 PRINT #9,LEN(A$)
110 FOR I = 1 TO LEN(A$)
120 PRINT # 9, ASC(MID$(A$,I))
130 NEXT I
```

e a cadeia necessitará de ser recriada carregando algo como:

```
100 INPUT #9,LL
110 FOR I = 1 TO LL
120 INPUT #9,TT: A$ = A$ + CHR$(LL)
130 NEXT I
```

Isto junta-se consideravelmente ao tempo de carregar e gravar, por isso deve tomar-se uma decisão sobre a utilização do armazena-

mento em cadeia, confrontando as economias na gravação e carregamento com as vantagens do método quando o programa está a correr.

Pilhas

Uma aplicação da técnica de armazenar em cadeias é a criação de «pilhas». O princípio da pilha é o mesmo do espigão dos escritórios. À medida que as cartas chegam ao escritório são espetadas no espigão até que chegue o tempo de tratar delas. Devido à maneira como são armazenadas, as letras são sempre processadas na ordem inversa, ou seja, a última é a primeira.

As pilhas são muito importantes nos computadores, já que muitas das operações que se executam são ditadas pela informação contida nos dados armazenados numa área, conhecida como *pilha (stack)*. Veja o exemplo das GOSUB. Em qualquer cadeia de GOSUB, o endereço a que o programa volta é sempre o do último GOSUB. Por isso, o endereço de cada GOSUB é armazenado no cimo da pilha e, quando se encontra um RETURN, somos remetidos para o endereço do cimo da pilha.

Em BASIC, uma pilha pode ser usada sempre que for necessário retirar vários assuntos ao mesmo tempo de uma estrutura de dados e modificá-los. À medida que se retira, cada um pode ser colocado numa matriz em separado e o seu lugar na matriz principal de dados armazenado numa pilha de valores numéricos de dois *bytes*. Quando se substituírem os assuntos na matriz principal, os lugares correctos obtêm-se pelo princípio da pilha.

Um exemplo comum de utilização da pilha é o registo da posição de um grupo de assuntos que possuem uma mesma característica, especialmente grupos de assuntos que são frequentemente somados ou subtraídos. Mais adiante, neste capítulo, veremos como isto pode ser aplicado para lembrar quantos espaços vazios existem numa matriz e onde é que eles estão. Cada vez que um assunto é apagado de uma matriz, o seu endereço é armazenado numa pilha com valores de dois *bytes:*

```
100 P1 = INT(PP/256):P2 = PP - 256*P1
110 STACK$ = CHR$(P1) + CHR$(P2) + STACK$
```

em que PP é o endereço dos assuntos a serem apagados. Para recuperar o endereço de um espaço vazio só é necessário:

```
100 PP = 256*ASC(STACK$) + ASC(MID$(STACK$,2)):STACK$ =
= MID$ (STACK$,2)
```

Isto permite que os assuntos sejam inseridos numa matriz sem terem de desviar todos os outros assuntos para conseguirem espaço no fim. Quando STACK$ ficar reduzida a uma cadeia vazia é uma indicação de que não há mais espaços disponíveis.

ESTRUTURAS DE DADOS EM CADEIA

Cadeias empacotadas

No capítulo referente ao manuseamento de cadeias já examinámos algumas maneiras simples de utilizar cadeias para armazenamento de informação, e como mais informação pode ser armazenada em menos espaço. Um exemplo disto foi a «cadeia empacotada» simples, utilizando um separador de um carácter para identificar cada parte, por forma a que a totalidade de uma entrada única de nome e morada pudesse ficar armazenada na forma:

```
SILVA*JOÃO ANTÓNIO*11 RUA DO LÁ VAI UM*
ALGUIDARES DE BAIXO* DISTRITO QUALQUER*PO0
0CO*0909 11111
```

A vantagem de uma estrutura destas é que cada assunto, em vez de ter um valor atribuído de 3 *bytes* de memória como no caso de um elemento completo de uma cadeia de matrizes, requer só um *byte,* o que contém «*». No princípio isto pode não parecer uma grande economia, mas para ficheiros contendo entradas que tenham cada uma oito assuntos separados, como no exemplo dado acima, poderá representar uma economia de 8×3 (por cada elemento separado de cada assunto) $-3 + 7$ (três *bytes* ou a cadeia única mais sete separadores), ou 14 *bytes* para uma entrada única. Para um ficheiro de 500 entradas a economia seria de 7000 *bytes,* o que poderá ser uma economia de peso quando a memória total para o programa e dados for menor que 43 000 *bytes.*

O inconveniente de tal método de empacotar cadeias é o tempo necessário para aceder a cada assunto específico. Para desempacotar o exemplo dado acima necessitamos de utilizar uma rotina de busca, tal como se descreveu no capítulo de manuseamento de cadeias, para identificar as partes separadas. O processo de busca num ficheiro de 500 entradas de um assunto específico, principalmente se ficar para o fim da cadeia, pode levar um tempo dolorosamente longo. Uma maneira de contornar isto consiste em dar à informação um formato mais fácil de apanhar do que os asteriscos quando se percorre o percurso ao longo do qual a cadeia foi empacotada. Tal técnica é conhecida por utilizar «ponteiros» e vamos examinar mais tarde uma utilização mais complexa dos «ponteiros», mas também mais eficaz, na estruturação de ficheiros. Por agora, a técnica consiste muito simplesmente em ligar, à frente da cadeia, números que indicam onde os caracteres devem partir-se:

```
6000 REM*********************************
6010 REM PACKED STRINGS
6020 REM*********************************
6030 CLS
6040 FOR i=1 TO 8
6050 ptr$="" : in$=""
6060 PRINT "Item no. ";i; : INPUT temp$
6070 in$=in$+temp$
6080 ptr$=ptr$+CHR$(LEN(in$))
6090 NEXT i
6100 in$=ptr$+in$
6110 :
```

O que acontece neste caso é que, à medida que cada uma das oito cadeias é introduzida, o programa leva o comprimento de IN$, que é a entrada global, e depois adiciona um carácter com o código ASCII correspondente àquele comprimento, à cadeia-ponteiro PTR$. No fim do ciclo, todos os assuntos estão guardados em IN$, sem qualquer indicação de onde acaba um e começa outro, enquanto em PTR$ estão armazenados oito caracteres cujo código ASCII é o ponto término de cada um dos assuntos. Estas duas são reunidas para produzirem uma entrada na forma pela qual será guardada na memória.

Tendo guardado os assuntos desta maneira, só é necessária uma rotina simples para os desempacotar outra vez:

```
6120 p1=8
6130 FOR i=1 TO 8
6140 p2=ASC(MID$(in$,i))+8
6150 PRINT MID$(in$,p1+1,p2-p1)
6160 p1=p2
6170 NEXT i
6180 :
```

Neste caso, os valores dos caracteres-ponteiro do princípio da cadeia são usados para passar através da cadeia, determinando o princípio e o fim de cada assunto. Necessitamos apenas de saber no início da rotina o número de caracteres-ponteiro, para que a posição do primeiro carácter do primeiro assunto seja determinada (isto é, o primeiro carácter depois dos ponteiros). Depois disto, cada carácter-ponteiro diz-nos onde termina um assunto, e nós sabemos que um carácter depois começa o próximo assunto. Esta rotina que demos supõe uma estrutura regular de oito assuntos — mas pode não ser este exactamente o caso. Os assuntos podem ser introduzidos em qualquer número até ao limite imposto pelo comprimento máximo da cadeia e, depois, um carácter extra adicionado à frente de PTR$, que regista o número de assuntos existentes. Este primeiro carácter pode então ser usado pelo ciclo que apanha os assuntos para determinar quantos caracteres-ponteiros se devem esperar.

Esta técnica é consideravelmente mais rápida do que esperar que o programa passe pela cadeia, carácter por carácter, à procura de separadores entre os assuntos. A sua principal limitação é limitar o comprimento de cada entrada a 255 caracteres menos o número de caracteres necessários aos ponteiros. Para a maior parte dos ficheiros, 255 caracteres são mais que suficientes, e esta limitação pode ser ultrapassada utilizando duas ou mais cadeias empacotadas, para cada entrada.

Cadeias empacotadas usando ponteiros de matrizes numéricas

Utilizar as cadeias empacotadas tal como as utilizámos mais acima tem a desvantagem de que obter o código ASCII de caracteres particulares pode levar tempo se grandes quantidades de assun-

tos estão a ser processados — e, ainda mais importante, como mencionámos atrás, ao tratar da gravação e carregamento de assuntos em fita ou disco. Por exemplo, para armazenar uma única entrada empacotada no formato especificado atrás, numa cassete, poderá ser necessária uma rotina como a seguinte:

```
100 FOR I = 1 TO 8:PRINT #1,ASC(MID$ (INS$, I):NEXT I
100 PRINT #1, MID$(IN$,9)
```

enquanto para recuperar os dados será necessário:

```
100 IN$ = «» :FOR I = 1 TO 8:INPUT# 1, TT:IN$ =
= IN$ + CHR$(TT):NEXT
110 INPUT #1, TT$:IN$ = IN$ + TT$
```

Toda esta tradução entre números e caracteres leva tempo durante a gravação e o carregamento. Esta demora pode ser reduzida armazenando os ponteiros numa matriz numérica, tal como foi indicado na secção referente ao empacotamento de números numa matriz de inteiros. Isto só pode ser feito quando o número de assuntos em cada cadeia empacotada é conhecido e regular, mas pode simplificar consideravelmente a gravação e carregamento. Se desejar armazenar ponteiros numa matriz de inteiros, então tem de se determinar quantos assuntos se desejam empacotar em cada cadeia, e depois dimensionar uma matriz de inteiros, com cada linha igual a metade daquele número (mais um se o número for ímpar). Então, exemplificando com um ficheiro de matrizes empacotadas contendo cada uma nove assuntos, teríamos de declarar uma matriz como POINTER%(99,7) e usar uma rotina como a que a seguir se indica:

```
7000 REM*********************************
7010 REM NUMERIC POINTERS
7020 REM*********************************
7030 DIM pointer%(99,7),temp%(7)
7040 DIM array$(99)
7050 more$="y"
7060 WHILE more$<>"n"
7070 in$=""
7080 CLS
7090 INPUT "Position for new item: ";pp
```

```
7100 PRINT
7110 FOR i=1 TO 8
7120 PRINT "Item no. ";i;": "; : INPUT t
emp$
7130 in$=in$+temp$
7140 temp(i-1)=LEN(in$)
7150 NEXT i
7160 array$(pp)=in$
7170 FOR i=0 TO 7
7180 pointer%(pp,i)=temp(i)
7190 NEXT i
7200 INPUT "More (y/n): ";more$
7210 more$=LOWER$(more$)
7220 WEND
7230 :
7240 :
```

Desempacotar uma estrutura deste tipo é também mais complexo que utilizar caracteres no interior da cadeia como ponteiros:

```
7250 FOR i=0 TO 99
7260 p1=0
7270 PEN 3 : PRINT i;": "; : PEN 1
7280 FOR j=0 TO 7
7290 PRINT MID$(array$(i),p1+1,pointer%(
i,j)-p1); : PEN 3 : PRINT "/"; : PEN 1
7300 p1=pointer%(i,j)
7310 NEXT j
7320 PRINT
7330 IF (i/20=INT(i/20)) AND i<>0 THEN t
$=INKEY$ : IF t$="" THEN 7330
7340 NEXT i
```

Pode executar estas rotinas ao mesmo tempo, introduzindo e tornando a chamar assuntos de uma matriz, porque o valor de «X» é indefinido e então será zero. Nas utilizações normais, outra parte qualquer do programa poderia ditar o lugar no qual o novo assunto estava a ser colocado.

A decisão quanto à utilização deste método depende da relação entre a quantidade de carregamento e gravação e a quantidade

de processamentos a efectuar em determinados assuntos. O processamento de um assunto por este método, extraindo os ponteiros ou substituindo-os, leva aproximadamente duas vezes mais tempo que usar valores armazenados em cadeias, embora dificilmente se note no processamento de qualquer assunto. É no entanto, mais uma útil aquisição para a panóplia de métodos, quanto mais não seja por amor à variedade. No próximo capítulo continuaremos a examinar estruturas de dados bastante mais difíceis de programar mas que podem reduzir o tempo de execução e a memória ocupada.

CAPÍTULO 9

Estruturas de dados — II

No último capítulo vimos métodos simples e claros de armazenar dados que permitiram resolver alguns problemas. Neste capítulo daremos atenção a algumas estruturas de dados que, embora não sejam menos úteis a resolverem problemas, exigem mais esforço de programação. O principal objectivo do capítulo é familiarizá-lo com um tipo de estruturas que reduzem o tempo necessário para inserir e remover dados, pela colocação dos assuntos nos locais mais convenientes, em vez de serem postos de acordo com o que à primeira vista pareceria mais lógico ao nosso raciocínio.

LISTAS LIGADAS

Anteriormente, quando discutimos estruturas de dados para números, deve ter reparado que, muitas vezes, o que fizemos foi empregar técnicas para reduzir a quantidade de dados desviados. Inserir um valor de um *byte* na primeira posição de uma matriz numérica implica o deslocamento de todos os dados correntes um espaço para a frente na matriz. Utilizar uma cadeia para armazenar valores, tal como vimos, resulta em que tudo o resto será desviado automaticamente sempre que se introduz um novo assunto no princípio. Algumas vezes, porém, tanto para números como para cadeias, os dados só podem ser armazenados praticamente numa matriz tão directa como A$(500), em que cada entrada necessita de uma linha da matriz. Supondo que os dados começam na linha zero

da matriz, inserir um novo assunto no princípio vai implicar o desvio de grandes quantidades de dados para fazer espaço para ele, o que pode levar muito tempo e, além disso, levantar o problema da colecção de lixo, tal como vimos no cap. 3.

A solução para este tipo de problema é, muitas vezes, armazenar os dados à medida que forem introduzidos e conservar um registo separado do local em que cada assunto *estaria* se a matriz estivesse numa ordem previamente escolhida, tal como a ordem alfabética para as cadeias. Uma estrutura deste tipo é a «lista ligada», em que o material parecerá estar numa ordenação totalmente aleatória mas em que cada assunto na lista terá ligado a ele o endereço do assunto a seguir na matriz na ordem que se pretende. Para exemplo, considere o conjunto de três assuntos em cadeia:

1) AAA
2) CCC
3) DDD

Se desejarmos adicionar um novo assunto, BBB, na ordem alfabética correcta, não há necessidade de deslocar CCC e DDD para criar espaço. O que necessitamos é, de alguma maneira, ligar ao assunto AAA um indicador que diga ao programa que o próximo assunto na ordem alfabética não é o assunto 2 mas sim o 4. Depois de encontrado o assunto 4 deve haver um indicador que lhe fique ligado *que* diga ao programa que o próximo assunto se encontra na posição 2. Desde que os indicadores correctos estejam ligados a cada assunto, o programa processará os assuntos na ordem 1,4,2,3 e obter-se-á uma ordenação alfabética com estes «ponteiros».

Para demonstrarmos o método que utilizaremos, corte seis pedaços de papel aproximadamente quadrados, de 10 cm de lado, e divida-os ao meio com um traço. A parte superior do papel será usada para o «ponteiro», indicando o destino seguinte da lista, a parte inferior regista o assunto a armazenar. Num dos quadrados escreva «1» na parte superior (ligeiramente mais pequeno, já que irá ser substituído), e «START» na parte de baixo. Noutro quadrado escreva «65535» na parte de cima e «STOP» na parte de baixo. Coloque estes quadrados um a seguir ao outro, com espaço para os restantes quatro ficarem a seguir em linha. A partir de agora colocaremos os quadrados por ordem alfabética e, quando compararmos qualquer quadrado novo, contaremos START como vindo antes no alfabeto e STOP como vindo depois.

Agora agarre noutro quadrado e escreva «BBB» na metade in-

ferior. Agora compare com START. De acordo com a regra acima, BBB vem depois de START; assim, olhe para a parte de cima de START e desloque-se para o quadrado indicado (estão numerados a partir de zero). Obviamente, o quadrado indicado é STOP e BBB deve vir antes. Já encontrámos a posição correcta para BBB imediatamente a seguir a START e antes de STOP.

Agora surge a parte crucial. *Não* desloque os dois quadrados existentes, coloque apenas o quadrado BBB na terceira posição (segunda posição a contar de zero). Risque o «1» que está na parte superior de START e substitua-o por «2». Agora START aponta para o quadrado novo BBB. No cimo do novo quadrado escreva «1», porque o próximo quadrado, na ordem alfabética, é STOP. Se agora seguir os ponteiros que estão na parte de cima dos quadrados, começando em START, a ordem será START, BBB, STOP.

Agora, num quarto quadrado, escreva «DDD» na metade inferior. Vá seguindo os ponteiros até ao ponto em que um quadrado está antes de DDD e outro está depois. Neste caso estará depois de BBB e antes de STOP. Coloque DDD na quarta posição. Risque o «1» que está em BBB e substitua-o por «3». Ponha «1» na parte superior de DDD para indicar que o quadrado que vem a seguir é STOP. Os ponteiros indicam a ordem START, BBB, DDD, STOP.

Deve agora ser capaz de lidar sozinho com os dois quadrados remanescentes, um dos quais deve ser «AAA» e o outro «CCC». No fim do processo, os ponteiros deverão indicar a ordem START, AAA, BBB, CCC, DDD, STOP. Repare que isto não tem nada a ver com a ordem dos quadrados na mesa, é puramente um reflexo da ordem alfabética e é alcançado sem ser necessário mover nada que já estivesse no lugar. Se compreendeu isto, está agora em condições de acompanhar o mesmo método em BASIC. Desta vez trabalharemos com cadeias em matrizes, utilizando ainda uma matriz de inteiros em paralelo para armazenar os ponteiros.

A rotina seguinte criará uma lista ligada por ordem alfabética:

```
1000 REM*********************************
1010 REM CREATE LINKED LIST
1020 REM*********************************
1030 DIM a$(499),a%(499)
1040 items=2
1050 holes=0
1060 a$(0)=CHR$(0)
1070 a$(1)=CHR$(255)
```

```
1080 a%(0)=1
1090 a%(1)=32767
1100 :
1110 IN$=""
1120 WHILE LOWER$(in$)<>"stop"
1130 CLS
1140 INPUT "Input new item or 'stop': ";
in$
1150 done=0
1160 small$=LOWER$(in$)
1170 IF small$="stop" OR small$="delete"
 OR small$="print" THEN done=1
1180 WHILE done=0
1190 address=0
1200 FOR i=1 TO items-1
1210 temp=address
1220 address=a%(address)
1230 IF a$(address)>=in$ THEN i=items-1
1240 NEXT i
1250 adress=a%(address)
1260 a%(temp)=items
1270 a$(items)=in$
1280 a%(items)=address
1290 items=items+1
1300 done=1
1310 WEND
1320 IF small$="print" THEN GOSUB 2000
1330 IF small$="delete" THEN GOSUB 3000
1340 WEND
1350 STOP
1360 :
```

Esta rotina necessita de alguma explicação; vamos examiná-la linha por linha.

1000-1050: A$ é a matriz principal onde os dados são guardados, A% será utilizada para armazenar os ponteiros, e ITEMS representa o número de assuntos já armazenados. ITEMS é inicialmente colocada em 2, já que a matriz será colocada com duas en-

tradas mudas para marcar o começo e o fim da lista ligada. A variável HOLES será explicada mais tarde.

1060-1070: Estas são duas entradas mudas, tal como START e STOP no nosso exemplo. Na primeira posição (endereço 0) das matrizes fica uma entrada cujo ponteiro indica a segunda posição (endereço 1), sendo o conteúdo do assunto CHR$(0). Isto significa que para todas as cadeias normais esta entrada será sempre o primeiro assunto da matriz em ordem alfabética. A segunda posição da matriz tem uma entrada cujos *bytes* de ligação apontam para 32767 (este valor nunca será usado, já que o último assunto não tem para onde apontar) e cujo conteúdo principal é um *byte* com o valor 255. Este assunto será sempre o último em ordem alfabética. A razão para estas duas entradas é que, tal como muitas outras estruturas de dados, é mais fácil começar com o que interessa que estar a colocar condições especiais no funcionamento de um ficheiro. Deste modo não precisamos de testar quando determinado assunto cai, ou não, no princípio ou fim da lista ligada. O primeiro e o último assunto da lista têm de ser tratados de maneira diferente, porque um não tem nenhum assunto para apontar e o outro não tem nenhum assunto que aponte para ele. Muitos métodos de armazenamento de dados têm problemas com o primeiro e último assuntos, e na maior parte dos casos é simples ultrapassar o problema e arranjar assuntos artificiais para o primeiro e último lugares quando se define a matriz.

1170: Esta linha leva em conta opções a adicionar um pouco mais tarde.

1190: A variável endereço (ADDRESS) será usada para armazenar a posição, na matriz, do assunto que no momento está a ser comparado: começa em zero.

1200: O número máximo de vezes que temos de comparar o assunto introduzido e um assunto existente é igual ao número de assuntos da matriz.

1210: TEMP será normalmente a posição do assunto examinado em último lugar na busca alfabética. Necessitamos de o ter presente para o caso de o novo assunto necessitar de ser introduzido entre aquele assunto e o seguinte; nesse caso, o ponteiro do assunto em TEMP deverá ser alterado.

1220: ADDRESS está agora afinada para a posição indicada pelo ponteiro do assunto que está em TEMP, isto é, o próximo assunto na lista ligada.

1230: Se o assunto que está em ADDRESS é maior que o novo assunto, então encontrámos o lugar correcto, por isso o ciclo terminou.

1250-1280: O ponteiro para o assunto na posição TEMP, que é o que fica antes do lugar para o novo assunto, deve ser agora feito para indicar a posição em que o novo assunto será inserido, isto é, a posição ITEMS. O ponteiro do novo assunto deve ser feito para indicar o assunto seguinte, aquele que era inicialmente indicado pelo ponteiro de TEMP.

Para verificarmos se a rotina está a trabalhar adequadamente, introduzimos as linhas seguintes, que imprimirão a lista por ordem alfabética:

```
2000 REM*********************************
2010 REM PRINT LINKED LIST
2020 REM*********************************
2030 CLS
2040 address=0
2050 FOR i=1 TO items-holes-2
2060 PEN 3: PRINT i;": "; : PEN 1 : PRIN
T a$(a%(address))
2070 address=a%(address)
2080 NEXT i
2090 PRINT : PRINT "Press any key."
2100 t$=INKEY$ : IF t$="" THEN 2100
2110 RETURN
2120 :
```

Neste caso, o ponteiro para cada assunto é empregue para obter o endereço do próximo. Mais uma vez é utilizada a variável HOLES, que será explicada mais tarde.

A rotina para apagar segue o mesmo tipo de exposição. Apaga o assunto numa posição especificada da lista — será fácil especifi-

car a cadeia e procurar, através da lista, o assunto antes de o apagar. Utilizaremos posteriormente este método numa versão nova:

```
3000 REM*********************************
3010 REM DELETE FROM LINKED LIST
3020 REM*********************************
3030 CLS
3040 INPUT "Number of item to delete: ";
nn
3050 address=0
3060 FOR i=0 TO nn-1
3070 temp=address
3080 address=a%(address)
3090 NEXT i
3100 a$(a%(temp))=""
3110 a%(temp)=a%(address)
3120 holes=holes+1
3130 RETURN
3140 :
```

Repare que neste caso nada se deslocou, tal como com a inserção. O que acontece é que o ponteiro do assunto anterior ao que se quer apagar é tornado igual ao do assunto a ser apagado. O assunto antes do que vai ser apagado indica agora o assunto a seguir ao que vai ser apagado, e o programa não mais se preocupará com o assunto apagado — mesmo se ele ainda lá estiver.

Nesta rotina, a variável HOLES será incrementada todas as vezes que um assunto for apagado. Isto pode parecer estranho: mais sensato seria simplesmente abater um valor a ITEMS. O problema é que cada novo assunto é acrescentado ao fim da lista, e ITEMS reporta-se ao fim da lista. Porque nesse momento não alteramos os endereços dos assuntos a ser inseridos ou apagados, reduzir ITEMS significaria que cada assunto novo apagaria no mesmo momento outro existente. Mais adiante, neste capítulo, examinaremos a maneira de tratar estes «buracos» que desperdiçam parte da matriz.

As listas ligadas constituem um método útil em programas onde as listas necessitam de ser regularmente compiladas, mas têm a desvantagem de só permitirem acesso a partir de um extremo — neste caso, só um dos extremos, o começo. É possível que sejam li-

gadas pelos dois extremos, dispondo de dois ponteiros por assunto, um indicando o assunto anterior e outro o que vem depois do assunto que estivermos a considerar; mas, de uma forma sensível, só podemos também começar de um dos extremos. Se saltarmos no interior de uma lista ligada não há maneira de se determinar em que ponto da lista se está, apenas se conhece, o endereço do assunto seguinte. Colocando o problema de outro modo: se desejarmos o trigésimo assunto de uma lista ligada não há maneira de saltar até lá, *tem* de se seguir pelo caminho tortuoso dos vinte e nove ponteiros que o antecedem. No próximo capítulo veremos a maneira de contornar este problema.

CADEIAS DE PONTEIROS

Sabemos que é possível criar uma lista ordenada, sem ter de andar constantemente a desviar assuntos no interior da matriz. Os assuntos podem simplesmente juntar-se no fim da matriz e a tarefa de determinar a sua localização correcta pode ser confiada a uma matriz de ponteiros. O problema é não haver maneira de saltar no interior da lista e determinar a posição de, por exemplo, o trigésimo assunto. Contudo, tanto no cap. 3 como no anterior, vimos que existe um tipo de estrutura de dados que torna possível inserir assuntos na sua posição correcta, sem desviar os assuntos existentes: é a cadeia. Sabemos também que é possível armazenar números em cadeias, e temos a experiência das técnicas que *não* devemos utilizar numa lista ligada, ou seja, saltar dentro da cadeia para uma posição determinada. Juntando tudo com o que aprendemos sobre ponteiros, acabamos na cadeia de ponteiros, que é a técnica que nos permite saltar no interior de uma matriz conservando as vantagens de inserir novos assuntos sem desviar os existentes para criar espaço.

Para conseguir tudo isto só necessitamos de uma técnica nova: como inserir números no domínio 0 – 65535 no interior de uma matriz de cadeias múltipla, em que o comprimento total das cadeias varia. Isto é similar à técnica que examinámos no cap. 3, e pode conseguir-se com algo parecido com a rotina seguinte:

```
1000 REM*********************************
1010 REM POINTER STRING -- INITIALISE
1020 REM*********************************
```

```
1030 DIM pointer$(19)
1040 items=0
1050 holes=0
1060 FOR i=0 TO 1
1070 FOR j=1 TO 125
1080 pointer$(i)=pointer$(i)+"01"
1090 NEXT j
1100 NEXT i
1110 items=250
1120 :

2000 REM*********************************
2010 REM INPUT ITEM
2020 REM*********************************
2030 done=0
2040 WHILE done=0
2050 CLS
2060 INPUT "Number in range 1-32767: ";n
n
2070 IF nn<0 THEN GOSUB 6000 : done=1
2080 WHILE done=0
2090 nn$=CHR$(nn/256)+CHR$(nn-256*INT(nn
/256))
2100 GOSUB 3000
2110 pointer$(ll)=LEFT$(pointer$(ll),2*lp
p-2)+nn$+MID$(pointer$(ll),2*lp-1)
2120 items=items+1
2130 GOSUB 4000
2140 done=1
2150 WEND
2160 PRINT : INPUT "More (y/n): ";more$
2170 IF LOWER$(more$)="n" THEN done=1 EL
SE done=0
2180 WEND
2190 STOP
2200 :

3000 REM*********************************
3010 REM OBTAIN POSITION
```

```
3020 REM*********************************
3030 -
3040 PRINT "Position (1 to ";items+1+(nn
<0);"): "; : INPUT pp
3050 ll=INT((pp-1)/125)
3060 lp=pp-125*ll
3070 RETURN
3080 :
4000 REM*********************************
4010 REM RE-ARRANGE FOR INSERTION
4020 REM*********************************
4030 FOR i=0 TO 18
4040 done=0
4050 WHILE done=0 AND LEN(pointer$(i))>2
50
4060 pointer$(i+1)=RIGHT$(pointer$(i),2)
+pointer$(i+1)
4070 pointer$(i)=LEFT$(pointer$(i),LEN(p
ointer$(i))-2)
4080 done=1
4090 WEND
4100 NEXT i
4110 RETURN
4120 :

5000 REM*********************************
5010 REM RE-ARRANGE FOR DELETION
5020 REM*********************************
5030 FOR i=0 TO 18
5040 done=0
5050 WHILE done=0 AND LEN(pointer$(i))<2
50 AND LEN(pointer$(i+1))>0
5060 pointer$(i)=pointer$(i)+LEFT$(point
er$(i+1),2)
5070 pointer$(i+1)=MID$(pointer$(i+1),3)
5080 done=1
5090 WEND
5100 NEXT i
5110 RETURN
```

```
5120 :

6000 REM*********************************
6010 REM DELETE
6020 REM*********************************
6030 PRINT
6040 PRINT "Deletion."
6050 GOSUB 3000
6060 pointer$(11)=LEFT$(pointer$(11),2*l
p-2)+MID$(pointer$(11),2*lp+1)
6070 GOSUB 5000
6080 RETURN
```

Repare que as linhas 1060 a 1110 são temporárias e serão usadas para testar a rotina.

Execute a rotina e introduza os seguintes valores para números e posição quando solicitado:

16705
251

16962
252

17219
253

16962
1

16705
1

16962
126

16705
126

Pare o programa com RUN/RESTORE e poderá verificar então que a rotina está a funcionar, introduzindo directamente, sem número de linha:

```
?PTR$(0)
```

que deverá resultar numa cadeia de «01» precedida de «AABB» que são a tradução dos números 16705 e 16962 quando expressos como pares de caracteres de dois *bytes*. Introduzir:

```
?PTR$(1)
```

deverá conduzir ao mesmo resultado.

O verdadeiro teste consiste na introdução de

```
?PTR$(2)
```

que deverá dar uma cadeia «01010101AABBCC». A contagem do número de caracteres nas duas primeiras cadeias deverá revelar que existem seis linhas cheias no visor, mais dez caracteres. Se tudo isto for verificado, demonstra que a rotina pode aceitar números de dois *bytes* ao longo de todo o comprimento dos dados, e desviará automaticamente os dados remanescentes, de tal modo que nenhuma cadeia única exceda 250 *bytes* de comprimento.

Podemos agora testar as duas últimas rotinas, que devem permitir apagar assuntos. Para executar o teste introduz-se:

```
16705
1

16705
126

16705
251
```

Isto colocará «AA» no início dos primeiros três elementos da matriz de cadeias.

Para apagar um assunto necessita primeiro de introduzir um número negativo à medida que o assunto é introduzido. Depois introduza « – 1» como resposta à próxima solicitação de um número e ser-lhe-á solicitada a posição do assunto a apagar — resposta

«1». Pare o programa com ESC e deverá descobrir que POINTER$(0) e POINTER$(1) contêm uma matriz de «01» terminada por «AA», e POINTER$(2) contém «0101». Isto demonstra que podemos inserir e apagar, ajustando o comprimento de cada elemento da matriz à medida que se progride. Pode agora apagar as linhas temporárias, 1060-1110.

ARMAZENAMENTO DE DADOS COM UMA CADEIA DE PONTEIROS

Sabemos o bastante para utilizarmos uma cadeia de ponteiros, agora só falta dar alguma coisa para que se aponte. Na rotina seguinte adicionam-se cadeias a uma matriz na posição mais conveniente, sendo utilizada a cadeia de ponteiros para registar a ordem correcta, aproximadamente da mesma maneira do que os ponteiros numéricos do fim do capítulo anterior.

```
1000 REM*********************************
1010 REM INITIALISE
1020 REM*********************************
1030 DIM pointer$(19),a$(499)
1040 items=2
1050 holes=0
1060 a$(0)=CHR$(0)
1070 a$(1)=CHR$(255)
1080 pointer$(0)=CHR$(0)+CHR$(0)+CHR$(0)
+CHR$(1)
1090 :

2000 REM*********************************
2010 REM INPUT ITEM
2020 REM*********************************
2030 done=0
2040 WHILE done=0
2050 CLS
2060 INPUT "Item to be inserted: ";in$
2070 small$=LOWER$(in$)
```

```
2080 IF small$="stop" OR small$="delete"
 OR small$="print" THEN done=1
2090 IF small$="delete" THEN GOSUB 6000
2100 IF small$="print" THEN GOSUB 7000
2110 IF small$="stop" OR small$="delete"
 OR small$="print" THEN done=1 ELSE done=0
2120 WHILE done=0
2130 GOSUB 3000
2140 nn$=CHR$(items/256)+CHR$(items-256*
INT(items/256))
2150 pointer$(ll)=LEFT$(pointer$(ll),2*l
p-2)+nn$+MID$(pointer$(ll),2*lp-1)
2160 a$(items)=in$
2170 items=items+1
2180 GOSUB 4000
2190 done=1
2200 WEND
2210 PRINT : INPUT "More (y/n): ";more$
2220 IF LOWER$(more$)="n" THEN done=1 EL
SE done=0
2230 WEND
2240 STOP
2250 :
3000 REM*********************************
3010 REM OBTAIN POSITION
3020 REM*********************************
3030 FOR pp=1 TO items-holes
3040 ll=INT((pp-1)/125)
3050 lp=pp-125*ll
3060 pa=256*ASC(MID$(pointer$(ll),2*lp-1
))+ASC(MID$(pointer$(ll),2*lp))
3070 IF a$(pa)>=in$ THEN pp=items-holes
3080 NEXT pp
3090 RETURN
3100 :
4000 REM*********************************
4010 REM RE-ARRANGE FOR INSERTION
4020 REM*********************************
4030 FOR i=0 TO 18
```

```
4040 done=0
4050 WHILE done=0 AND LEN(pointer$(i))>2
50
4060 pointer$(i+1)=RIGHT$(pointer$(i),2)
+pointer$(i+1)
4070 pointer$(i)=LEFT$(pointer$(i),LEN(p
ointer$(i))-2)
4080 done=1
4090 WEND
4100 NEXT i
4110 RETURN
4120 :
5000 REM*********************************
5010 REM RE-ARRANGE FOR DELETION
5020 REM*********************************
5030 FOR i=0 TO 18
5040 done2=0
5050 WHILE done2=0 AND LEN(pointer$(i))<
250 AND LEN(pointer$(i+1))>0
5060 pointer$(i)=pointer$(i)+LEFT$(point
er$(i+1),2)
5070 pointer$(i+1)=MID$(pointer$(i+1),3)
5080 done2=1
5090 WEND
5100 NEXT i
5110 RETURN
5120 :
6000 REM*********************************
6010 REM DELETE
6020 REM*********************************
6030 PRINT
6040 INPUT "Item to be deleted: ";in$
6050 GOSUB 3000
6060 IF in$<>a$(pa) THEN PRINT : PRINT "
Not present." : RETURN
6070 pointer$(11)=LEFT$(pointer$(11),2*1
p-2)+MID$(pointer$(11),2*lp+1)
6080 GOSUB 5000
6090 holes=holes+1
```

```
6000 REM*********************************
6010 REM DELETE
6020 REM*********************************
6030 PRINT
6040 INPUT "Item to be deleted: ";in$
6050 GOSUB 3000
6060 IF in$<>a$(pa) THEN PRINT : PRINT "
Not present." : RETURN
6070 pointer$(ll)=LEFT$(pointer$(ll),2*lp-2)+MID$(pointer$(ll),2*lp+1)
6080 GOSUB 5000
6090 holes=holes+1
6100 a$(pa)=""
6110 empty$=CHR$(pa/256)+CHR$(pa-256*INT(pa/256))+empty$
6120 RETURN
6130 :
7000 REM*********************************
7010 REM PRINT LIST
7020 REM*********************************
7030 CLS
7040 IF items-holes=2 THEN RETURN
7050 FOR pp=2 TO items-holes-1
7060 ll=INT((pp-1)/125)
7070 lp=pp-125*ll
7080 pa=256*ASC(MID$(pointer$(ll),2*lp-1))+ASC(MID$(pointer$(ll),2*lp))
7090 PRINT a$(pa)
7100 NEXT pp
7110 RETURN
7120 :
```

Praticamente a única inovação consiste na sub-rotina iniciada em 3000, que faz a passagem da matriz, na ordem indicada, pela matriz de ponteiros, com o endereço correcto de cada assunto na matriz A$, obtido através da variável PA. Quando se descobrir a posição correcta já está armazenada na variável PP, LL e LP, e estas podem fazer-se regressar à sub-rotina 2000, que já está preparada para aceitar a posição desejada, definida por aquelas três. A

única alteração importante à secção começando em 2000 é que NN$, os dois caracteres que guardam o novo ponteiro até que seja inserido, é uma tradução de ITEMS, que é ao mesmo tempo o número de assuntos até então armazenados *e* o número do primeiro elemento livre na matriz A$.

Quando se quer apagar, segue-se quase o mesmo método que quando se introduz um assunto — faz-se a passagem da lista na ordem ditada pela cadeia de ponteiros para o primeiro assunto que é superior, alfabeticamente, ao assunto a ser introduzido. Depois da rotina de busca, o assunto encontrado na lista é comparado com o assunto a ser apagado, e o apagamento só ocorre se houver coincidência. Se o assunto especificado *for* encontrado, a sua posição já está na variável PA, estando a posição do seu ponteiro armazenada em LL e LP.

O funcionamento conjunto das duas rotinas pode ser testado, introduzindo «PRINT» quando for pedida uma introdução *(input)*. Isto chamará a rotina 7000 para imprimir a lista toda, utilizando valores sucessivos contidos em POINTER$.

O PROBLEMA DO BURACO NEGRO

Resta para ser resolvido um problema crucial que tem a ver com os espaços originados na matriz quando se apagam assuntos. No momento que um assunto é apagado deixa um buraco na matriz, que não será utilizado outra vez. Se não fizermos nada para resolver este problema, a matriz ficará rapidamente cheia destes espaços, até que eles a ocupem completamente e não haja mais espaço para os dados, mesmo que a matriz esteja quase vazia.

Felizmente o problema pode ser facilmente ultrapassado através de um método que já esboçámos, que é a utilização de uma «pilha». O que fazemos é registar cada buraco, à medida que forem criados, numa pilha chamada EMPTY$. Quando existe um novo assunto a ser inserido, o programa verificará primeiro se EMPTY$ regista a posição de algum buraco no qual o assunto possa ser colocado. Somente se não existir um buraco é que o assunto é colocado no fim da lista.

Para adicionar esta nova função, substitua as duas sub-rotinas correspondentes da última versão do programa por estas que se seguem:

```
1000 REM*********************************
1010 REM INITIALISE
1020 REM*********************************
1030 DIM pointer$(19),a$(499)
1040 empty$=""
1050 items=2
1060 holes=0
1070 a$(0)=CHR$(0)
1080 a$(1)=CHR$(255)
1090 pointer$(0)=CHR$(0)+CHR$(0)+CHR$(0)
+CHR$(1)
1100 :

2000 REM*********************************
2010 REM INPUT ITEM
2020 REM*********************************
2030 done=0
2040 WHILE done=0
2050 CLS
2060 INPUT "Item to be inserted: ";in$
2070 small$=LOWER$(in$)
2080 IF small$="stop" OR small$="delete"
 OR small$="print" THEN done=1
2090 IF small$="delete" THEN GOSUB 6000
2100 IF small$="print" THEN GOSUB 7000
2110 IF small$="stop" OR small$="delete"
 OR small$="print" THEN done=1 ELSE done
=0
2120 WHILE done=0
2130 GOSUB 3000
2140 temp=items
2150 IF LEN(empty$)=0 THEN 2200
2160 temp=256*ASC(empty$)+ASC(MID$(empty
$,2))
2170 empty$=MID$(empty$,3)
2180 items=items-1
2190 holes=holes-1
2200 nn$=CHR$(temp/256)+CHR$(temp-256*IN
T(temp/256))
```

```
2210 pointer$(ll)=LEFT$(pointer$(ll),2*l
p-2)+nn$+MID$(pointer$(ll),2*lp-1)
2220 a$(temp)=in$
2230 items=items+1
2240 GOSUB 4000
2250 done=1
2260 WEND
2270 PRINT : INPUT "More (y/n): ";more$
2280 IF LOWER$(more$)="n" THEN done=1 EL
SE done=0
2290 WEND
2300 STOP
2310 :
3000 REM*********************************
3010 REM POSITION
3020 REM*********************************
3030 FOR pp=1 TO items-holes
3040 ll=INT((pp-1)/125)
3050 lp=pp-125*ll
3060 pa=256*ASC(MID$(pointer$(ll),2*lp-1
))+ASC(MID$(pointer$(ll),2*lp))
3070 IF a$(pa)>=in$ THEN pp=items-holes
3080 NEXT pp
3090 RETURN
3100 :
4000 REM*********************************
4010 REM RE-ARRANGE FOR INSERTION
4020 REM*********************************
4030 FOR i=0 TO 18
4040 done=0
4050 WHILE done=0 AND LEN(pointer$(i))>2
50
4060 pointer$(i+1)=RIGHT$(pointer$(i),2)
+pointer$(i+1)
4070 pointer$(i)=LEFT$(pointer$(i),LEN(p
ointer$(i))-2)
4080 done=1
4090 WEND
4100 NEXT i
```

```
4110 RETURN
4120 :
5000 REM*********************************
5010 REM RE-ARRANGE FOR DELETION
5020 REM*********************************
5030 FOR i=0 TO 18
5040 done2=0
5050 WHILE done2=0 AND LEN(pointer$(i))<
250 AND LEN(pointer$(i+1))>0
5060 pointer$(i)=pointer$(i)+LEFT$(point
er$(i+1),2)
5070 pointer$(i+1)=MID$(pointer$(i+1),3)
5080 done2=1
5090 WEND
5100 NEXT i
5110 RETURN
5120 :
6000 REM*********************************
6010 REM DELETE
6020 REM*********************************
6030 PRINT
6040 INPUT "Item to be deleted: ";in$
6050 GOSUB 3000
6060 IF in$<>a$(pa) THEN PRINT : PRINT "
Not present." : RETURN
6070 pointer$(11)=LEFT$(pointer$(11),2*1
p-2)+MID$(pointer$(11),2*lp+1)
6080 GOSUB 5000
6090 holes=holes+1
6100 a$(pa)=""
6110 empty$=CHR$(pa/256)+CHR$(pa-256*INT
(pa/256))+empty$
6120 RETURN
6130 :
7000 REM*********************************
7010 REM PRINT LIST
7020 REM*********************************
7030 CLS
```

```
7040 IF items-holes=2 THEN RETURN
7050 FOR pp=2 TO items-holes-1
7060 ll=INT((pp-1)/125)
7070 lp=pp-125*ll
7080 pa=256*ASC(MID$(pointer$(ll),2*lp-1
))+ASC(MID$(pointer$(ll),2*lp))
7090 PRINT a$(pa)
7100 NEXT pp
7110 RETURN
7120 :
```

Deverá agora ser capaz de inserir e apagar todas as vezes que quiser — desde que não arranje mais de 125 buracos de uma só vez, o que causaria a necessidade de tornar EMPTY$ numa matriz com mais de uma linha. Isto, não sendo difícil, introduz um pouco mais de complexidade ao método.

UTILIZAÇÃO DE FUNÇÕES DEFINIDAS PELO UTILIZADOR

As pessoas que têm experiência de programação já repararam que em muitos casos as rotinas deste capítulo poderiam ter sido escritas mais condensadamente. A razão por que o não foram é que este não é um livro sobre programação em geral, mas uma tentativa de explicar os princípios que estão por trás da criação de programas de aplicações. Existe contudo uma técnica que necessita de ser mencionada, porque é inevitável para o tipo de programas que podem ser escritos a partir das rotinas que demos neste capítulo.

As estruturas de dados complexas implicam a utilização de linhas complexas de programação para extrair a informação — como, por exemplo, é o caso das cadeias de ponteiros. Na subrotina que obtém a posição de um assunto, no último programa deste capítulo, aparece a linha seguinte:

```
pa =
  = 256*ASC(MID$(pointer$(ll),2*lp-1)) + ASC(MID$(pointer$(ll),
    2*lp))
```

No próximo capítulo descobriremos a repetição desta linha várias vezes no decurso de uma sub-rotina, para acelerar a busca de uma posição determinada na matriz. Nestes casos existe uma alternativa satisfatória, que consiste na definição de uma função como se indica a seguir:

```
DEF FN A$ = 256*ASC(MID$(pointer$(11).2*1p-1)) + ASC(MID$
(pointer$ (11), 2*1p))
```

e depois fazer uso de:

```
pa = FNa$
```

no corpo principal de programa.

CONCLUSÃO

Ao debruçar-se sobre o conteúdo deste capítulo e do anterior, não duvidará de que o que se pretende não é que aplique de imediato todas as técnicas e métodos neles contidas. As estruturas de dados aqui descritas são uma lembrança de que existem muitas maneiras de utilizar a memória do Amstrad, todas elas com diferentes potencialidades mas podendo todas dar uma contribuição valiosa numa circunstância determinada. Se começar a programar com seriedade, chegará a altura em que um determinado conjunto de dados, talvez só dez ou vinte assuntos usados num dado ponto do programa, lhe despertem a atenção. Será a altura de regressar a este capítulo e constatar, felizmente, que um dos muitos métodos aqui apresentados será a resposta necessária. Quando fizer isto, tenha a certeza de que quase todas as vezes que trabalha com um programa de manuseamento de dados comercial eles fazem uso de estruturas de dados que têm uma semelhança notável com o conteúdo deste capítulo — e correm tão bem para si em BASIC como para as casas fornecedoras de *software* nas mais exotéricas linguagens de programação.

CAPÍTULO 10

Técnicas de busca

No capítulo anterior tivemos necessidade de, por momentos, ultrapassar os nossos conhecimentos. A razão para isto foi que, ao olhar para uma estrutura de dados e para a maneira como um assunto foi inserido ou apagado, tem de se assumir uma maneira de encontrar o lugar correcto para essa inserção ou apagamento. Quando a matriz estiver ordenada de uma maneira directa, ou quando se dispõe de ponteiros de qualquer tipo, uma das maiores tarefas que o programa executa é nada mais nada menos que procurar alguma coisa — muito pouco se pode fazer pelo manuseamento de dados se não se puder armazenar ou retirar os dados na ordem desejada.

No último capítulo ignorámos esta importante área, empregando sem comentários o método mais simples e, sem surpresa, o menos eficiente dos métodos de detectar o lugar correcto para um assunto numa lista ordenada, que foi começar por um dos extremos e examinar, um a um, os assuntos até encontrar a posição desejada. Neste capítulo vamos dar um passo atrás e olhar outra vez para o problema da busca e para o modo como está relacionado com a inserção de novos assuntos começando com buscas e estruturas muito simples e vendo finalmente como um método de busca avançado pode ser aplicado ao tipo de métodos de matrizes de ponteiros que examinámos no último capítulo.

BUSCA SIMPLES E DESVIO

De longe a maneira mais simples, em termos de programação, de inserir um assunto numa grande matriz ordenada é procurar assunto por assunto e deslocar um lugar para baixo todos os assuntos

que ficam a seguir. Diz-se muitas vezes que a maneira mais eficaz de conseguir isto é procurar a partir do fim da matriz, comparar com o assunto a inserir e depois desviar cada assunto que a comparação revele ser necessário deslocar no fim. A verdade ou falsidade desta afirmação depende de ser ou não possível combinar as linhas necessárias, para a busca do lugar correcto de um assunto, com as que deslocam os dados para que fique lugar para a inserção do novo assunto.

A listagem seguinte é a de uma simples rotina que, primeiro, posiciona uma matriz ordenada de números, depois procura, através da matriz, o lugar correcto de inserção de um assunto novo, e finalmente desvia os dados para criar espaço para o novo assunto:

```
1000 REM*********************************
1010 REM CONTROL ROUTINE
1020 REM*********************************
1030 EVERY 50 GOSUB 7000
1040 GOSUB 2000
1050 CLS
1060 INPUT "Input new value (-32768 to 3
2767): ";ni
1070 seconds=0 : minutes=0 : hours=0
1080 GOSUB 3000
1090 GOSUB 4000
1100 PRINT hours"/"minutes"/"seconds
1999 STOP
2000 REM*********************************
2010 REM INITIALISE
2020 REM*********************************
2030 DIM a%(9999)
2040 items=9950
2050 FOR i=0 TO items-1
2060 a%(i)=i
2070 NEXT i
2080 RETURN
2090 :
3000 REM*********************************
3010 REM SEARCH
3020 REM*********************************
```

```
3030 sp=0
3040 PRINT "Started search."
3050 FOR i=0 TO items-1
3060 IF a%(i)>=ni THEN sp=i : i=items-1
3070 NEXT i
3080 RETURN
3090 :
4000 REM*********************************
4010 REM INSERT
4020 REM*********************************
4030 PRINT "Started insert."
4040 FOR i=items TO sp+1 STEP -1
4050 a%(i)=a%(i-1)
4060 NEXT i
4070 items=items+1
4080 a%(sp)=ni
4090 RETURN
4100 :
7000 REM*********************************
7010 REM TIME
7020 REM*********************************
7030 seconds=seconds+1
7040 IF seconds>59 THEN seconds=0 : minu
tes=minutes+1
7050 IF minutes>59 THEN minutes=0 : hour
s=hours+1
7060 RETURN
7070 :
```

É uma rotina que se pode aplicar generalizadamente a outros tipos de dados com pequeníssimas alterações. A rotina, tal como está, pode aplicar-se à inserção em diferentes lugares.

Para testar a rotina introduza 5000 como assunto novo. O registo de tempos deverá mostrar que a execução demora 48 segundos.

Podemos agora comparar este método com um segundo, que dispensa um dos ciclos, executando comparações e deslocamentos dentro do mesmo ciclo. Junte a rotina seguinte à de busca anteriormente listada:

```
1110 seconds=0 : minutes=0 : hours=0
1120 GOSUB 5000
1130 PRINT hours"/"minutes"/"seconds
5000 REM*******************************
5010 REM SEARCH AND INSERT COMBINED
5020 REM*******************************
5030 PRINT "Starting combined search and
 insert."
5040 sp=0
5050 FOR i=items-1 TO 0 STEP -1
5060 IF ni>a%(i) THEN sp=i : i=0 : ELSE
a%(i+1)=a%(i)
5070 NEXT i
5080 a%(sp+1)=ni
5090 items=items+1
5100 RETURN
5110 :
```

Esta demora apenas 42 segundos a executar, ou seja, uma economia de 10% em relação ao método anterior. Obterá resultados muito diferentes se os assuntos se inserirem no princípio ou fim da matriz, mas colocando um assunto no meio dar-nos-á o tempo médio que levará a introduzir uma série de assuntos que não estiverem próximo do princípio ou fim da matriz.

O inconveniente do segundo método é que executa conjuntamente duas funções de programas totalmente diferentes, a busca e a inserção. Não há inconveniente no presente caso, mas já não se passa o mesmo se tivermos uma maneira mais rápida de inserir ou de fazer a busca. Já vimos que existem maneiras mais rápidas de inserir dados, no capítulo referente às matrizes de ponteiros. Também existe uma maneira mais rápida de fazer a busca.

BUSCA BINÁRIA

Nas duas rotinas dadas acima, é claro, têm de se fazer aproximadamente 5000 comparações antes de a posição correcta ser encontrada. Também é verdade que nem sempre a matriz terá todos os seus elementos cheios de assuntos úteis. Contudo, o número de comparações será sempre, em média, igual a metade do número de

assuntos da matriz. O facto é que isto é totalmente desnecessário, já que, para conhecer o lugar correcto na matriz acima necessita apenas de um máximo de 13 comparações. Considere o exemplo seguinte:

1) Na matriz acima desejamos inserir um item novo, seja o número 6172.

2) Começamos a busca pelo lugar certo examinando a posição da matriz que representa a mais elevada potência de dois que é possível conseguir no total do número de itens da matriz. Neste caso temos 10 000 itens e a mais elevada potência de dois que cabe neste número é 8192 e chamar-lhe-emos «*step value*» (valor de degrau), ou SV. A nossa primeira comparação é feita entre o novo item e o que se encontra na matriz na posição SV. Chamaremos à posição de busca SP *(search position);* ela terá de começar em 8191 para se entrar em conta com o facto de a matriz começar em 0 e não em 1.

3) O valor em 8191 é superior ao novo item, 6172; então tomamos o valor original de SV, 8192, e dividimo-lo por 2, obtendo o novo SV de 4096. Este é subtraído de 8191 (SP), dando um novo SP de 4095.

4) A comparação é feita agora para 4095 (SP). O número na posição da matriz é inferior ao novo item; assim, dividimos SV por 2 (=2048) e desta vez adiciona-se ao SP, o que dá o novo SP de 6143.

5) Mais uma vez, o item em 6143 é inferior ao novo item, por isso SV é dividido ao meio (=1024). Isto é adicionado ao SP, e dá 7167.

6) O item em 7167 é superior a 6172; portanto, SV é dividido por 2 (=512) e subtraído de SP, o que dá 6655.

7) A busca prossegue para as seguintes localizações e com os saltos indicados:

6655 (−256)
6399 (−128)
6271 (−64)
6207 (−32)
6175 (−16)
6159 (+8)
6167 (+4)
6171 (+2)
6173 (−1)
6172 que é a posição correcta.

Note que, em todo este processo, o que fizemos foi adicionar ou subtrair potências decrescentes de 2, dependendo de quando a posição de referência é superior ou inferior na matriz. O sucesso não está dependente de termos uma matriz em que, como é o caso, os seus elementos sejam números espaçados de um. Tudo o que é preciso será dispor de uma matriz de cadeias, ou numérica, que tenha sido previamente ordenada, por ordem crescente ou decrescente.

O número de comparações que é necessário fazer para inserir um novo item dentro da matriz será, em geral:

INT(LOG(ITEMS)/LOG(2)) + 1

O que esta fórmula faz é dizer quantos dígitos existem, no número em ITEMS, quando for expresso em notação binária, mais um, e este será sempre o número máximo de comparações necessárias. Então:

INT(LOG(500)/LOG(2)) + 1 = 9

e 500 em binário é:

111110100 — nove dígitos de comprimento

Um método destes representa uma economia maciça quando comparado com o método anterior, pois economizam-se 4087 comparações. Se estivéssemos à procura de um item numa matriz, em vez da posição em que devemos colocar alguma coisa, então a economia podia ainda ser superior. A busca de um item pode envolver, no caso de o item não estar presente, procurar entre 10 000 itens; em vez disso, com a busca binária só usaríamos 13 comparações e se, na 13.ª comparação, não tivéssemos encontrado o item desejado, saberíamos que ele não estaria na matriz. Nas linhas seguintes, que são para juntar às anteriores, a busca binária é utilizada para encontrar a posição correcta de inserção de um novo item, que mais uma vez pode ser 5000:

```
1140 seconds=0 : minutes=0 : hours=0
1150 GOSUB 6000
1160 GOSUB 4000
1170 PRINT hours"/"minutes"/"seconds
```

```
6000 REM*********************************
6010 REM BINARY SEARCH
6020 REM*********************************
6030 PRINT "Starting binary search."
6040 IF items=0 THEN sp=0 : RETURN
6050 power=INT(LOG(items)/LOG(2))
6060 sp=2^power-1
6070 FOR sv=power-1 TO 0 STEP -1
6080 sp=sp+2^sv*((ni<a%(sp))-(ni>a%(sp))
)
6090 IF sp>items-1 THEN sp=items-1
6100 NEXT sv
6110 IF a%(sp)<ni THEN sp=sp+1
6120 RETURN
6130 :
```

Existem duas diferenças ligeiras na descrição do método que seguimos mais acima. Primeiro é possível ao ponteiro de busca (SP) saltar por cima do número de assuntos da matriz, que acabam em ITEMS – 1. Se isto acontecer, SP é colocada no extremo dos dados, e esta alteração de valor não tem influência nenhuma na operação de busca. Em segundo lugar, se o item introduzido não estiver na matriz, a posição encontrada será uma antes ou depois da que ele teria se lá estivesse. Se o valor encontrado estivesse acima dele, então todos os dados a partir desse podiam ser desviados para criar lugar. Se for encontrado o valor abaixo, então SP deverá ser deslocado um lugar para cima na matriz.

Executando a rotina outra vez, com o valor 5000, teremos uma duração de somente 26 segundos, que é basicamente o tempo necessário para desviar os elementos. O atraso causado pelo elevado número de comparações foi quase inteiramente eliminado.

BUSCA PURA

Nos exemplos dados acima, supôs-se que estávamos a fazer a busca para inserir um item ou assunto novo. Isto não necessita de ser verdade. Os primeiro e terceiro métodos podem ser usados sim-

plesmente para encontrar a posição de um assunto, se realmente existir na matriz. Para ver como isto pode ser feito, junte as linhas seguintes ao programa:

```
1180 seconds=0 : minutes=0 : hours=0
1190 GOSUB 3000
1200 PRINT hours"/"minutes"/"seconds
1210 seconds=0 : minutes=0 : hours=0
1220 GOSUB 6000
1230 PRINT hours"/"minutes"/"seconds
```

Introduzir agora um valor implicará que sejam impressos tanto o valor como a sua posição e o tempo de uma busca normal e outra binária. Introduzindo 5000, o valor que está no meio da matriz deverá levar 23 segundos a encontrar com a busca normal e menos um segundo com a busca binária. Experimente introduzir um número não inteiro — obteremos o inteiro abaixo e acima do valor introduzido. Introduzindo uma comparação como a seguinte:

```
1225 IF A%(SP) <> NI THEN PRINT «NÃO PRESENTE» :
STOP
```

fornece-se um teste automático de presença de itens.

BUSCA BINÁRIA EM MATRIZES DE PONTEIROS

Até agora demonstrámos que a busca binária é claramente um grande passo em frente, na busca, em relação a uma lista de dados. Noutro aspecto, no entanto, demos um passo atrás: no que respeita a termos ficado restritos ao tipo de estruturas de dados, que envolvem o desvio de grandes quantidades de dados no interior da matriz — algo que já tínhamos ultrapassado no último capítulo.

Nesta última secção veremos como o método da busca binária pode ser aplicado à estrutura de matrizes de ponteiros, tal como aprendemos no fim do último capítulo. Note que o método não pode ser aplicado a todos os tipos de estruturas de dados: uma lista ligada não pode ser sujeita a uma busca deste tipo porque, nessa lista, só se pode aceder a um item de cada vez.

A maneira como podemos aplicar a busca binária ao programa do fim do último capítulo — o que, vem a propósito salientar, demonstra mais uma vez as vantagens da programação modular — consiste em substituir uma única sub-rotina para se ter uma alteração significativa no método:

```
3000 REM*********************************
3010 REM POSITION
3020 REM*********************************
3030 temp=items-holes
3040 power=INT(LOG(temp)/LOG(2))
3050 pp=2^power
3060 FOR sv=power-1 TO 0 STEP -1
3070 ll=INT((pp-1)/125) : lp=pp-125*ll
3080 pa=256*ASC(MID$(pointer$(ll),2*lp-1
))+ASC(MID$(pointer$(ll),2*lp))
3090 pp=pp+2^sv*((in$<a$(pa))-(in$>a$(pa
)))
3095 PRINT pp
3100 IF pp>temp-1 THEN pp=temp-1
3105 PRINT pp
3110 NEXT sv
3120 ll=INT((pp-1)/125)
3130 lp=pp-125*ll
3140 pa=256*ASC(MID$(pointer$(ll),2*lp-1
))+ASC(MID$(pointer$(ll),2*lp))
3150 IF a$(pa)<in$ THEN pp=pp+1
3160 ll=INT((pp-1)/125)
3170 lp=pp-125*ll
3180 pa=256*ASC(MID$(pointer$(ll),2*lp-1
))+ASC(MID$(pointer$(ll),2*lp))
3190 RETURN
3200 :
```

Não é uma rotina que se deseje introduzir num pequeno programa que armazene pequenas quantidades de dados. Com grandes quantidades, no entanto, utilizando uma matriz de ponteiros, para cortar os desvios das matrizes, e uma busca binária, para reduzir o tempo de busca, pode originar-se uma espantosa diferença de velocidade.

Posfácio

Se trabalhou este livro, procurando dominar todas as técnicas à medida que avançava, então tem mais paciência do que eu. É claro que as domino a todas, mas paguei-o bem caro.

Muito possivelmente passou por grandes partes do livro, parando para experimentar rotinas ou técnicas que pareciam particularmente atraentes. Se procedeu desse modo, não atire agora o livro para a prateleira, considerando-o como outro qualquer livro de texto a ir buscar de vez em quando. A melhor maneira de utilizar este livro, se não espera mergulhar num grande projecto já a seguir, é ter presentes alguns dos seus programas anteriores e ver em que medida pode ser aplicado neles o que aprendeu agora. Pode parecer estranho aplicar algumas das mais complexas técnicas a programas que são simples e até satisfatórios, mas não é isso que se pretende. A maneira de tornar sua qualquer técnica de programação é avançar com a sua utilização, uma vez ou duas, até mesmo quando a utilização não for adequada (não sei se me faço entender).

Outra ideia pode ser agarrar as rotinas deste livro e começar um dicionário de técnicas úteis em disco ou fita. As técnicas mais complexas estão, como descobriu, apresentadas já de maneira que podem ser introduzidas e testadas. As técnicas de uma linha podem introduzir-se conjuntamente com uma instrução de *input* e *print* para se ver o efeito. Armazenar as rotinas deste modo permitirá que numa data posterior as examine, para ver se constituem a resposta para um dado problema e, nesse caso, fazer *merge* com ela, no meio do programa, como foi sugerido no primeiro capítulo. À medida que ler revistas e outros livros, pode adicionar rotinas mais jeitosas, mesmo que não lhe interessem os programas de que elas fazem parte. Tenho também a minha própria colecção de rotinas que não fazem nada particularmente útil numa dada altura (mas

que fazem bem qualquer coisa) e fico sempre satisfeito por ter começado isto, quando surge um novo desafio. Por isso, não se esqueça das técnicas deste livro, para as quais não vê utilidade de momento. Na próxima semana podem constituir a diferença entre o fracasso e o sucesso.

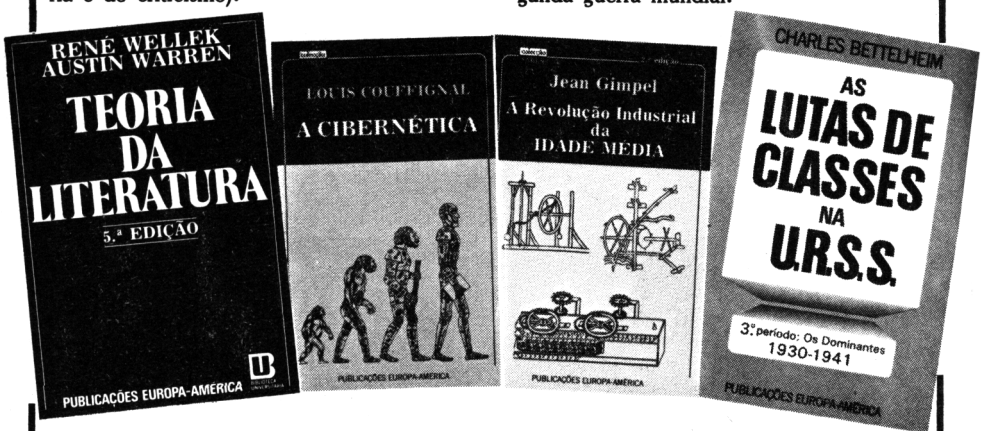
«A cultura é aquilo que fica depois de se esquecer o resto»

TEORIA DA LITERATURA
5.ª EDIÇÃO
RENÉ WELLEK
AUSTIN WARREN

Um livro que não tem paralelo próximo com outro. Procura unir a «poética» (ou teoria literária) e o «criticismo» (valoração da literatura) à «erudição» ('investigação') e à «história literária» ('a dinâmica' da literatura em contraste com a 'estatica' da teoria e do criticismo).

AS LUTAS DE CLASSES NA U.R.S.S.
3.º período: Os Dominantes
1930-1941
CHARLES BETTELHEIM

Com esse último volume, consagrado ao período 1930-1941, o autor conclui o vasto fresco da luta de classes da União Soviética, da Revolução de Outubro à segunda guerra mundial.



A CIBERNÉTICA
LOUIS COUFFIGNAL

Em termos gerais, pode dizer-se que a Cibernética é o campo da ciência que compara os sistemas de comunicação e de controlo, transformados em mecanismos, com aqueles que se apresentem nos organismos biológicos. Esta obra, para além de aspectos teóricos, analisa os resultados da Cibernética.

A Revolução Industrial da IDADE MÉDIA
2.ª edição
Jean Gimpel

Um livro que nos motra uma Idade Média diferente que, longe de ter sido um tempo de trevas, aparece aos nossos olhos como um dos períodos mais fecundos da história dos homens.

EUROPA-AMERICA ...a memória no futuro

Este livro pretende demonstrar como se desenvolvem programas de aplicação sérios para utilização no seu **Amstrad CPC 464** ou **664**. David Lawrence destaca a importância de os programas obedecerem a uma concepção e planeamento cuidados e ilustra os pontos principais, com grande profusão de exemplos.

Começa com as vantagens da programação modular, que torna muito mais fácil a subsequente verificação e a caça ao erro em relação às rotinas que tem. Seguem-se interessantes capítulos sobre métodos adequados de introdução da informação, manuseamento de cadeias, como evitar erros, armazenamento e recuperação da informação, estruturas de dados, ordenação e busca.

Em todos os pontos principais há valiosas sugestões e novas ideias, para outras maneiras de encarar os assuntos mais delicados: como gravar grandes quantidades de informação na memória de **Amstrad** ou como ordenar eficientemente longas listas de assuntos. Tudo isto torna o livro um valioso instrumento a ter à mão sempre que quiser teclar programas novos e potencialmente difíceis.

David Lawrence é autor, com muito sucesso, de grande número de livros sobre computadores. É dele, de parceria com Simon Lane, ***O Amstrad Funcional.*** Ao mesmo tempo é colaborador de programas de rádio e da revista ***Popular Computing Weekly.***

ARTE DE VIVER®

13311286 2