OSBORNE/ McGRAW-HILL

C

AVANÇADO

GUIA DO USUÁRIO

McGraw-Hill

Herbert Schildt

C
AVANÇADO
Guia do Usuário

C AVANÇADO

Guia do Usuário

Herbert Schildt

Tradução:

Cláudio Gaiger Silveira
Engenheiro Eletrônico
Analista de Software

Revisão Técnica:

Mônica Soares Rufino
Engenheira Eletrônica
Analista de Software

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

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

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

Do original
Advanced C

Editor: Milton Mira de Assumpção Filho
Coordenadora de Revisão: Daisy Pereira Daniel
Supervisor de Produção: Neder Roberto O. Campos

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

Schildt, Herbert.
S36c C avançado: guia do usuário / Herbert Schildt; tradução Cláudio Gaiger Silveira; revisão técnica Monica Soares Rufino. – São Paulo: McGraw-Hill, 1987.

1. C (Linguagem de programação para computadores) I. Título.

87-1130 CDD-001.6424

Índices para catálogo sistemático:
1. C: Linguagem de programação: Computadores: Processamento de dados 001.6424
2. Linguagem C: Programação: Computadores: Processamento de dados 001.6424

Quero agradecer aos amigos da SID e em particular a Bruna Curi que colaborou muito com este trabalho.

Cláudio Gaiger Silveira

Agradeço aos meus pais, aos amigos da SID e ao Chico a grande paciência e ajuda prestada.

Mônica Soares Rufino

A meus filhos
Sasha, Josselyn, Jonathan e Rachel

Herbert Schildt

NOTA DA REVISORA TÉCNICA

Este livro contém, em sua maioria, programas reais completos que foram testados diretamente a partir do texto.

Dentro das limitações apresentadas pelo autor, os exemplos foram analisados e, muitos, compilados e corrigidos pelo revisor, procurando deixar o programa mais útil e corrreto para o leitor.

O compilador utilizado foi o mesmo do autor, Aztec C86, que contém uma biblioteca gráfica e cujas funções como LINE( ) e MODE( ) poderão conter em outros compiladores, porém com diferentes nomes.

Apesar das alterações feitas, há, ainda, vários pontos em que o leitor poderá adaptar os programas ao compilador e ao formato mais conveniente para a sua aplicação.

SUMÁRIO

INTRODUÇÃO

Considero-me um felizardo por ter sido capaz de escrever o tipo de livro de programação que sempre quis possuir. Algum tempo atrás, quando comecei a programar, procurei encontrar um livro que possuísse algoritmos para algumas tarefas como ordenações, listas encadeadas, simulações e compilação de expressões apresentadas de uma forma direta. Queria um livro que me mostrasse a programação, mas que eu também pudesse tirar da prateleira para encontrar o que precisava quando necessário. Infelizmente, nunca encontrei o livro que procurava – decidi, então, escrevê-lo.

Este livro explora um "range" largo de assuntos e contém muitos algoritmos úteis, funções e métodos escritos em linguagem C. C é, de fato, uma linguagem de programação de sistemas, assim como uma das mais populares linguagens de programação profissional *general purpose* disponível. Uma grande variedade de compiladores C está disponível para praticamente todos computadores, e muitos não muito caros. Usei o compilador Aztec C86 para PC IBM; porém com algumas poucas exceções, qualquer versão 7, de compilador C compatível com UNIX, irá compilar e "rodar" todo o código deste livro.

O Capítulo 1 começa com uma breve história do C e uma pequena revisão da linguagem. A ordenação, tanto de vetores como de arquivos em disco, é explicada no Capítulo 2. O Capítulo 3 trata de pilhas, filas, listas encadeadas e árvores binárias. (Você pode pensar que é muita coisa para um capítulo; porém, os assuntos se desenvolvem perfeitamente juntos e fornecem uma unidade sólida.) Métodos de alocação dinâmica são discutidos no Capítulo 4. O Capítulo 5 apresenta uma visão geral das interfaces com o sistema operacional e o encadeamento da linguagem assembler. O Capítulo 6 cobre a parte de estatística e inclui um programa completo de estatística. Códigos, cifragens e compressão de dados são os tópicos abordados no Capítulo 7, que inclui também uma

pequena história da criptografia. O Capítulo 8 detalha vários geradores de números randômicos e discute como utilizá-los em duas simulações. A primeira simulação é uma fila de caixa de supermercado; a segunda é um programa de gerenciamento de acesso aleatório de uma carteira de investimentos.

O Capítulo 9, particularmente, é meu favorito porque contém o código completo de um compilador recursivo descendente. Anos atrás, eu teria dado qualquer coisa para ter este código! Se você precisa avaliar expressões, o Capítulo 9 é feito para você. Os Capítulos 10 e 11 discutem conversões para outras linguagens, eficiência, portabilidade e depuração.

H.S.

Capítulo 1

UMA REVISÃO DA LINGUAGEM C

Este livro utiliza um método de solução de problemas para ilustrar conceitos avançados de programação em linguagem C: examina tarefas usuais de programação e desenvolve soluções com ênfase quanto ao estilo e estrutura. Através deste método, muitos tópicos avançados e variações de C são descritos, bem como a teoria geral de programação que é mencionada por trás de cada solução. É preciso ter algum conhecimento prático de C; porém, sua experiência não precisa ser grande. Uma revisão da linguagem C é apresentada mais adiante, neste capítulo.

Convencionou-se duas notações neste livro. Primeiro, todos os nomes de variáveis e palavras-chaves de C são impressos em **negrito**. Segundo, todas as funções são em **negrito** e seguidas de parênteses. Estas convenções eliminam possíveis confusões entre nomes de variáveis e nomes de funções. Por exemplo, uma variável chamada "test" é apresentada como **test**; por outro lado, uma função com o mesmo nome é apresentada como **test**().

Todos exemplos e programas neste livro foram compilados e rodam usando-se o compilador Aztec C para IBM PC. Geralmente, qualquer compilador versão 7 compatível com UNIX, tais como os compiladores Lattice ou Microsoft, compilará e rodará o código deste livro. Existem muitos compiladores disponíveis para a maioria dos computadores do mercado, e você deve facilmente encontrar algum que sirva as suas necessidades. Lembre-se, no entanto, de que todos compiladores diferem levemente – especialmente em suas bibliotecas –, assim sendo, leia atentamente o manual do compilador que você está usando.

AS ORIGENS DO C

C foi criado e primeiramente implantado por Dennis Ritchie num Dec PDP-11, utilizando o sistema operacional UNIX. C é o resultado do processo de desenvolvimento que começou com uma antiga linguagem chamada BCPL, que é ainda usada principalmente na Europa. BCPL, desenvolvida por Martin Richards, influenciou na linguagem chamada B, que foi criada por Ken Thompson e levou ao desenvolvimento de C.

Embora C tenha sete diferentes tipos de dados internos, não é uma linguagem com digitação pesada como Pascal ou Ada. C permite conversões de quase todos os tipos, e tipos caractere e inteiro podem ser livremente misturados na maioria das expressões. Não possui nenhuma checagem de erro em tempo de processamento – tais como checagem de vetor limite ou checagem de compatibilidade tipo argumento. Isto é de responsabilidade do programador.

C é especial porque permite a direta manipulação de bits, bytes, palavras e ponteiros. Isto o torna bem adequado para programação a nível de sistema, onde essas operações são comuns. Outra vantagem de C é ter somente 28 *palavras-chaves*, que são os comandos que compõem a linguagem. Para uma comparação, considere o Basic do IBM PC que tem 159 palavras-chaves.

Inicialmente desenvolvido para rodar no sistema operacional UNIX, C tornou-se tão popular que existem compiladores disponíveis para praticamente todos os computadores e sistemas operacionais. Isto significa que o código C é de grande portabilidade entre computadores e sistemas operacionais, possibilitando escrever um único código e utilizá-lo em qualquer máquina.

C COMO UMA LINGUAGEM ESTRUTURADA

A linguagem C é considerada normalmente uma linguagem estruturada com alguma semelhança com Algol e Pascal. Embora o termo *linguagem em bloco estruturado* não se aplique a C no sentido estritamente acadêmico, a linguagem C faz parte informal deste grupo de linguagens. A característica que distingue uma linguagem em bloco estruturado é a *compartimentalização de códigos e dados*. Isto é, a linguagem pode cortar ou esconder do resto do programa todas as informações e instruções para realizar uma tarefa específica. Geralmente, compartimentalização é atingida por sub-rotinas com *variáveis locais*, que são temporárias. Desta forma, é possível escrever sub-rotinas onde os eventos dentro delas não provoquem efeitos colaterais em outras partes do programa. O uso excessivo de *variáveis globais* (variáveis existentes em todo programa) pode provocar o

aparecimento de *bugs*, causando efeitos indesejáveis. Em C, todas as sub-rotinas são discretas.

Funções são blocos de código C onde ocorre toda a atividade do programa. Possibilitam a realização de tarefas específicas no programa, porém codificadas e definidas separadamente. Após depurar uma função que usa somente variáveis locais, você pode dispor dela para trabalhar em várias situações, sem criar efeitos colaterais em outras partes do programa. Todas as variáveis declaradas em uma determinada função serão reconhecidas somente por esta.

Em C, usar blocos de código cria também estrutura de programa. Um *bloco de código* é um grupo de linhas de programação logicamente interligadas que pode ser tratado como uma unidade. É criado pela colocação das linhas de código entre chaves, como em:

```
if (x<10)  {
           printf("too low, try again");
           reset_counter(-1);
}
```

Neste exemplo, as duas declarações entre chaves após o **if** são ambas executadas se **x** for menor do que 10. Essas duas declarações juntas com as chaves representam um bloco de código. Elas são enlaçadas (como o link) juntas: uma das declarações não pode ser executada sem que a outra também o seja. Em C, toda linha de programa pode ser ou uma simples linha ou um bloco de linhas. O uso de blocos de código cria programas de fácil leitura com uma lógica fácil de ser seguida.

C é a linguagem do programador. Diferente da maioria das linguagens de alto nível, impõe certas restrições. Utilizando C o programador pode evitar o uso de código assembler em quase todas as situações. De fato, um dos motivos da criação de C foi o de ter uma alternativa para programação em código assembler.

A linguagem assembler utiliza uma representação simbólica do código binário real que o computador executa diretamente. Cada operação da linguagem assembler é mapeada em uma tarefa simples para o computador executar. A linguagem assembler fornece ao programador o potencial para realizar tarefas com a máxima flexibilidade e eficiência, é notadamente difícil de trabalhar quando desenvolvendo ou depurando programas. Além do mais, já que a linguagem assembler é desestruturada por natureza, o programa final tende a ser um "espagueti de códigos", um emaranhado de saltos (jumps), chamadas (calls) e indexações. Isto faz dos programas em linguagem assembler programas difíceis de entender, alterar e manter.

Inicialmente, C foi usado para programação de sistemas. Um *programa de sistema* é parte de uma larga classe de programas que formam a porção do sistema operacional do computador ou seus utilitários de suporte. Por exemplo, os seguintes programas são normalmente conhecidos com *programas de sistema*

- Sistemas operacionais
- Interpretadores
- Editores
- Assemblers
- Compiladores
- Gerenciadores de banco de dados

Com o aumento de popularidade de C, muitos programadores começaram a usar o C para programar todo tipo de tarefa por causa da portabilidade e eficiência. Assim sendo, existem compiladores para C para quase todos os computadores, é fácil transportar o código escrito para uma máquina e compilar e rodar com poucas ou nenhuma alteração em outra máquina. Essa portabilidade economiza tempo e dinheiro. Compiladores C tendem a gerar códigos, reduzidos e rápidos – menores e mais rápidos que a maioria dos compiladores BASIC, por exemplo.

Talvez a razão real de C ser usado em todos os tipos de tarefa de programação seja porque os programadores gostam de C. C tem a velocidade do assembler e a extensão do FORTH, embora tendo algumas das restrições do PASCAL. Um programador de C pode criar e manter uma única biblioteca de funções de acordo com sua personalidade. Por que C permite – e certamente encoraja – compilações separadas, grandes projetos são facilmente gerenciados.

Muitas funções neste livro usam a função chamada **getnum( )**. C não possui nenhum método intrínseco para entrada de números decimais através do teclado e, contrário ao que se possa pensar, a função de biblioteca-padrão **scanf( )** é geralmente inadequada para o usuário. Porém, a função especial **getnum** é usada sempre que um número decimal precisa ser lido do teclado. O código-fonte para **getnum( )** é mostrado a seguir.

```
getnum()        /* lê um número decimal
                do teclado */
{
                char s[80];

                gets(s);
                return(atoi(s));

}
```

A função **atoi(s)** é da biblioteca-padrão, usada para converter uma seqüência de caracteres de dígitos em um número inteiro. Se o seu compilador possuir uma função similar a **getnum()**, sinta-se à vontade para substituí-la.

UMA BREVE REVISÃO

Antes de você começar a explorar os vários problemas e soluções de programação, leia o restante deste capítulo para rever a linguagem C. Se você é um experimentado programador de C, salte para o Capítulo 2.

Um sumário de declarações da maioria das palavras-chaves em C, uma revisão das diretivas do pré-processador e a descrição de algumas das funções da biblioteca-padrão usadas neste livro são apresentados em *Linguagem C – Guia do Operador*, de Cláudio Gaiger Silveira, Ed. McGraw-Hill Ltda.

As 28 palavras-chaves seguintes, combinadas com a sintaxe característica de C, compõem a linguagem C de programação.

auto	**double**	**if**	**static**
break	**else**	**int**	**struct**
case	**entry**	**long**	**switch**
char	**extern**	**register**	**typedef**
continue	**float**	**return**	**union**
default	**for**	**short**	**unsigned**
do	**goto**	**sizeof**	**while**

As palavras-chaves de C são sempre escritas em letras minúsculas. Em C, letras maiúsculas fazem diferença, isto é, **else** é uma palavra-chave, mas ELSE não.

VARIÁVEIS – TIPOS E DECLARAÇÕES

C é composto de sete tipos de dados inerentes, como mostramos aqui:

Tipo de Dado	Palavra-Chave Equivalente em C
caractere	**char**
inteiro curto	**short int**
inteiro	**int**
inteiro sem sinal	**unsigned int**
inteiro longo	**long int**
ponto flutuante	**float**
ponto flutuante dupla precisão	**double**

Algumas implementações de C também suportam **inteiro longo s/ sinal.**

Os nomes das variáveis em C podem ter um ou vários caracteres de comprimento; o máximo comprimento depende do seu compilador. Para melhor compreensão, o traço pode também ser usado como parte do nome de uma variável (por exemplo **first name**). Não esqueça que em C, letras maiúsculas e letras minúsculas são diferentes – **test** e **TEST** são duas variáveis diferentes.

Todas as variáveis precisam ser declaradas antes de serem usadas. O formato geral de declaração é:

type **variable name;**

Por exemplo, para declarar **x** como um ponto flutuante, **y** variável inteira e **ch** um caractere, você faria:

```
float x;
int y;
char ch;
```

Além dos vários tipos de dados, você pode criar combinações de tipos de dados utilizando-se de **struct** e **union**. Você também pode criar novos nomes para as variáveis usando **typedef.**

Uma *estrutura* é uma lista de variáveis agrupadas e referenciadas por um único nome. O formato geral de declaração de uma estrutura é:

struct struct_name {
elemento 1;
elemento 2;
•
•
•
} **struct_variable;**

Como exemplo, a estrutura seguinte tem 2 elementos **name**, um vetor de caracteres e **balance**, um número de ponto flutuante.

```
struct client {
        char name[80];
        float balance;
};
```

Para referenciar um elemento individual de uma estrutura, o operador ponto é usado se a estrutura for global ou declarada na função a ele referenciada. O operador vetor é usado em todos os outros casos.

Quando duas ou mais variáveis compartilham da mesma área de memória, uma **union** é definida. O formato geral de uma **union** é:

union union_name {
elemento 1;
elemento 2;
•
•
•
} **union_variable;**

Os elementos de uma **união** se sobrepõem uns aos outros. Por exemplo, a seguir apresentamos a declaração da **união t**, que em memória pode ser vista como na Figura 1.1.

```
union  tom {
          char ch;
          int x;
} t;
```

As variáveis individuais que compreendem a **união** são referenciadas usando-se o operador ponto se a **união** for global ou for declarada na mesma função que a variável referenciada.

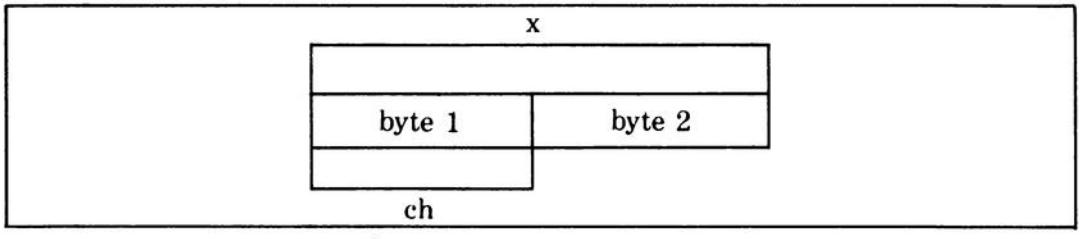


Figura 1.1. União t em memória.

Os três tipos de modificadores em C são: **extern, register** e **static** – podem ser usados para alterar a maneira como C trata as variáveis que se seguem a esses modificadores. Se o modificador **extern** for colocado antes do nome de uma variável, o compilador reconhecerá que a variável foi declarada em algum outro lugar.

O modificador **extern** é normalmente usado quando existem dois ou mais arquivos dividindo as mesmas variáveis globais.

O modificador **register** só pode ser usado em variáveis locais tipo caractere ou inteiro. Faz o compilador tentar manter o valor no registrador da CPU em vez de colocá-lo na memória. Isto faz com que todas as referências a esta variável sejam extremamente rápidas.

Neste livro, variáveis **register** são usadas para controle de laços. Por exemplo, a seguinte função usa a variável **register** para um controle de laço:

```
f1( )
{
    register int t;
    for(t=0;t<10000;++t) {
        .
        .
        .
    }
}
```

O modificador **static** instrui o compilador C a manter a variável local existente durante o tempo de vida do programa, em vez de criá-la e destruí-la. Lembre-se de que o valor das variáveis locais são descartados quando a função termina e retorna. Usar **static** mantém o valor da variável entre chamadas de funções.

Vetores Você pode declarar vetores com quaisquer tipos de dados descritos anteriormente. Por exemplo, para declarar um vetor de variáveis inteiras de 100 elementos, você escreveria

```
int x[100];
```

Isto cria uma matriz com 100 elementos de extensão; o primeiro elemento é o 0 e o último o 99. Por exemplo, o laço abaixo armazena números de 0 a 99 no vetor x:

```
for(t=0;t<100; t++) x[t]=t;
```

Vetores com várias dimensões são declarados colocando-se a dimensão adicional dentro de um outro par de colchetes. Por exemplo, para declarar uma matriz inteira 10 por 20, você escreveria

```
int x[10][20];
```

OPERADORES

C tem um rico conjunto de operadores que podem ser divididos em classes: *aritmética, relacional* ou *lógico, bit-a-bit, ponteiros, declarações* e *miscelânia.*

Operadores Aritméticos C possui sete operadores aritméticos

Operador Aritmético	Ação
–	subtração, menos unitário
+	adição
*	multiplicação
/	divisão
%	módulo divisão
– –	decremento
+ +	incremento

As prioridades destes operadores são:

maior prioridade ++ -- - (menos unitário)
* / %
menor prioridade + -

Operadores com o mesmo grau de prioridade são avaliados da esquerda para a direita.

Operadores Lógicos e Relacionais Operadores lógicos e relacionais são usados, freqüentemente juntos, para produzir resultados VERDADEIRO/FALSO. Qualquer número diferente de zero indica VERDADE em C. No entanto, um operador lógico ou relacional em C produz um número 1 para VERDADE e 0 para FALSO. Aqui estão os operadores lógicos e relacionais:

Operador Relacional	Significado
>	maior do que
>=	maior ou igual a
<	menor do que
<=	menor ou igual a
==	igual
!=	diferente

Operador Lógico	Significado
&&	e
||	ou
!	inversor

As prioridades destes operadores são:

maior prioridade !
> >= < <=
== !=
&&
menor prioridade ||

Por exemplo, esta expressão é avaliada como VERDADE:

```
(100<200) && 10
```

Operadores bit-a-bit Ao contrário de outras linguagens, C tem operadores bit-a-bit, que permitem alterar cada bit de uma variável. Os operadores bit-a-bit, apresentados aqui, só podem ser usados em inteiros ou caracteres.

Operador bit-a-bit	Significado
&	e
¦	ou
^	ou exclusivo
~	complemento de 1
>>	deslocamento para a direita
<<	deslocamento para a esquerda

As tabelas-verdade para E, OU e OU EXCLUSIVO são como segue:

&	0	1
0	0	0
1	0	1

¦	0	1
0	0	1
1	1	1

^	0	1
0	0	1
1	1	0

Essas regras são aplicadas para cada bit em um byte quando uma das operações bit-a-bit E, OU ou OU EXCLUSIVO é executada; por exemplo,

```
  0100    1101
& 0011    1011
-------------
  0000    1001

  0100    1101
¦ 0011    1011
-------------
  0111    1111
```

```
  0100   1101
^ 0011   1011
-------------
  0111   0110
```

Em um programa, você usa **&**, **|** ou **^** como qualquer outro operador, como mostrado aqui:

```
main()
{
        char x,y,z;
        x=1,y=2,z=3;
        x=x & y; /* x agora igual a zero */
        y=x | z; /* y agora igual a 4 */
}
```

O operador complemento de um (~) inverte todos os bits de um byte. Por exemplo, se uma variável **ch** tem o padrão binário

```
0011   1001
```

então

```
ch = ~ch;
```

coloca o seguinte padrão binário

```
1100   0110
```

em **ch.**

Os operadores de deslocamento à direita e à esquerda movimentam todos os bits de um byte ou uma palavra para a direita ou para a esquerda de um número especificado de bits. Com o deslocamento dos bits, zeros vão sendo inseridos. O número do lado direito do operador de deslocamento especifica o número de posições a deslocar. As formas gerais dos operadores de deslocamento são:

variável >> *número de deslocamentos*
variável << *número de deslocamentos*

Por exemplo, dada a configuração binária

0 0 1 1 1 1 0 1

um deslocamento simples à direita fornece

0 0 0 1 1 1 1 0

enquanto um deslocamento simples à esquerda produz

0 1 1 1 1 0 1 0

O deslocamento à direita é efetivamente uma divisão por 2, e o deslocamento à esquerda é uma multiplicação por 2. Por exemplo, a seqüência abaixo primeiro multiplica e depois divide o valor de **x**.

```
int x;
x=10;
x=x<<1;
x=x>>1;
```

Devido à maneira como os números negativos são representados no interior da máquina, você deve tomar cuidado quando tentar usar o deslocamento para multiplicação ou divisão. Mover 1 para o bit mais significativo faz com que o computador interprete que o número é negativo.

Lembre-se: os operadores bit-a-bit são usados para modificar o valor da variável – eles diferem dos outros operadores lógicos e relacionais, que produzem resultado VERDADEIRO/FALSO.

A prioridade dos operadores bit-a-bit é como segue:

maior prioridade ~
>> <<
&
^
menor prioridade |

Operadores de Ponteiros Operadores de ponteiro são importantes em C: eles permitem não só que seqüências de caracteres, vetores e estruturas sejam passados para funções, como também permitem que funções de C modifiquem seus argumentos chamados. Os dois operadores de ponteiros são **&** e *. (Infelizmente, esses operadores usam os mesmos símbolos da multiplicação e do E bit-a-bit, que são completamente desrelacionados dos operadores de ponteiros.)

O operador & retorna o endereço da variável a que precede. Por exemplo, se o inteiro **x** está localizado no endereço de memória 2000, então

```
y=&x;
```

coloca o valor 2000 em **y**. O & pode ser lido como "o endereço de". Por exemplo, a declaração anterior poderia ser lida como "Coloque o endereço de **x** em **y**".

O operador * leva o valor da variável à que precede e usa este valor como endereço da informação na memória. Por exemplo,

```
y=&x;
*y=100;
```

coloca o valor 100 em **x**. O * pode ser lido como "no endereço". Este exemplo poderia ser lido como "Coloque o valor 100 no endereço y". O operador * pode também ser usado do lado direito da declaração. Por exemplo,

```
y=&x;
*y=100;
z=*y/10;
```

coloca o valor 10 em **z**.

Estes operadores são chamados operadores de ponteiro por serem designados para trabalhar em *variáveis ponteiro*.

A variável ponteiro armazena o endereço de outra variável; isto é, aponta esta variável, como mostra a Figura 1.2.

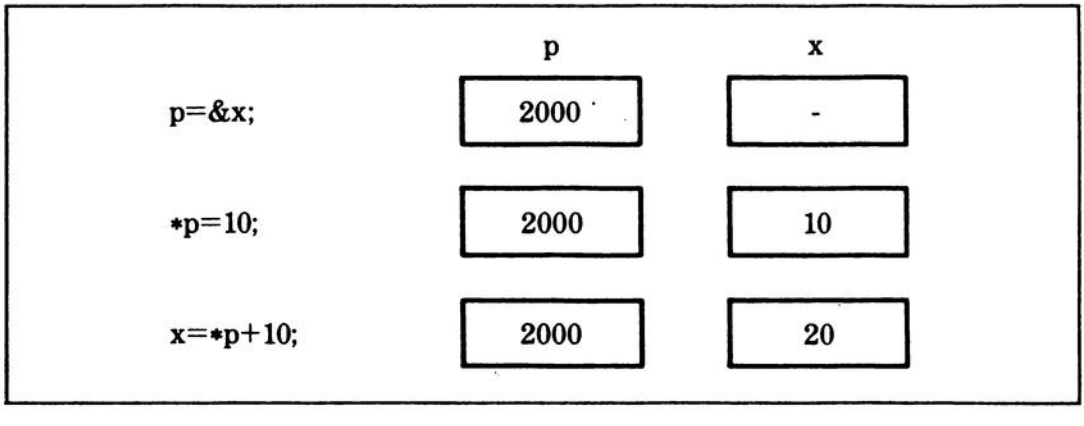


Figura 1.2. Operações com ponteiros para o ponteiro caractere **p** e inteiro **x**, com **x** na localização de memória 2000.

Operador Declaração Em C, o operador declaração é um simples sinal de igual. Porém, C permite uma conveniente "abreviação" para declarações do tipo igual.

variável 1 = variável 1 expressão operador;

Aqui estão dois exemplos:

```
x = x + 10
y = y / z
```

Declarações deste tipo podem ser abreviadas para forma geral

variável 1 operador = expressão;

Nestes dois exemplos, eles podem ser abreviados para

```
x + = 10;
y / = z;
```

Esta notação abreviada é usada freqüentemente em programas escritos em C por programadores experientes de C; assim, você deve familiarizar-se com elas.

O Operador ? O operador ? é um ternário que é usado para trocar declarações **if** do tipo geral

if *expressão1* **then x** = *expressão2*
else x = *expressão2*

A forma geral do operador ? é

variável = expressão1 ? *expressão2* : *expressão3;*

Se *expressão1* é VERDADE, então o valor de *expressão2* é assumido para a *variável;* do contrário, a *variável* assume o valor da *expressão3*. Por exemplo,

```
x=(y<10) ? 20 : 40;
```

atribui a x ou o valor 20 se y for menor do que 10 ou 40 se y não for menor do que 10.

O operador ? existe porque o compilador C pode produzir muitos códigos

eficientes para este tipo de declaração – muito mais rápido do que a declaração equivalente **if/else.**

Operador Miscelânea O operador . (ponto) e o operador → (vetor) são usados para referenciar elementos individuais de estruturas e uniões.

O operador ponto é usado quando a estrutura ou união é global ou quando o código referenciado está dentro da mesma função como uma declaração de estrutura ou união. O operador vetor é usado quando uma estrutura ou união é passada para uma função ou quando só um ponteiro para uma estrutura ou união está disponível. Por exemplo, dada a estrutura global

```
struct tom {
            char ch;
            float w;
} clyde;
```

você escreveria

```
clyde.w=123.23;
```

para atribuir o valor 123.23 ao elemento **w** da estrutura **clyde.**

O operador , (vírgula) é normalmente usado numa declaração tipo **for.** Faz com que uma seqüência de operações seja executada. Quando é usado do lado direito de uma dada declaração o valor de toda expressão é o valor da última expressão separada por vírgula na lista. Por exemplo, depois da execução de

```
y=10;
x=(y=y-5,25/y);
```

x tem o valor 5 porque o valor original de **y** (10) é subtraído de 5, e então este valor é divisor de 25, fornecendo o resultado de 5.

Embora **sizeof** seja também considerado uma palavra-chave, é um operador de tempo de compilação usado para determinar o tamanho do tipo de dado em bytes, incluindo definições do usuário de estruturas e uniões. Por exemplo,

```
int x;

printf("%d", sizeof(x));
```

imprime o número 2 para muitos microcomputadores.

Parênteses são considerados operadores que aumentam a prioridade das operações contidas neles.

Colchetes executam indexação de vetores.

Um *cast* é um operador especial que força a conversão de um tipo de dado em outro. A forma geral é

(type) variável

Por exemplo, com o objetivo de fazer o inteiro *count* ser usado numa chamada para **sqrt(),** que é uma rotina de raiz quadrada da biblioteca padrão de C e requer um parâmetro ponto flutuante, "cast" força **count** a ser tratado como um tipo **float:**

```
float y;
int count;

count= 10;
y= sqrt ((float) count);
```

A Tabela 1.1 lista a prioridade de todos os operadores C. Observe que todos os operadores – exceto o operador unitário e ? – associam-se da esquerda para a direita. Os operadores unitários (*, & e –) e o operador ? associam-se da direita para a esquerda.

FUNÇÕES

Um programa em C é uma seqüência de uma ou mais funções. Uma das funções deve ser **main( )** porque a execução inicia-se com esta função. Historicamente **main( )** é a primeira função no programa; no entanto, ela poderia estar em qualquer lugar. A forma geral de uma função em C é:

function_name *(lista de parâmetros)*
declaração de parâmetros
{
corpo da função
}

Se a função não tem declaração de parâmetros, nenhuma declaração de parâmetros é necessária. Todas as funções terminam e retornam para o procedimento de chamada, automaticamente, quando a última chave é encontrada. Você pode forçar esse retorno antes, com o uso da declaração **return.**

Tabela 1.1. Prioridade dos operadores C

Maior	( ) [ ] → .
	! ~ ++ −− − (type) * & sizeof
	* / %
	+ −
	<< >>
	< <= > >=
	== !=
	&
	^
	|
	&&
	||
	?:
	= += −= *= /= %= >>= <<= &= ^= |=
Menor	,

Todas as funções retornam um valor. Se uma declaração **return** for parte da função, então o valor da função será o valor na declaração **return**. Se nenhum **return** estiver presente, a função retornará zero. Por exemplo

```
f1()
{
        int x;

        x= 100;
        return (x/10);
}
```

retorna o valor 10, enquanto

```
f2()
{
        int x;

        x=100;
        x= x/10;
}
```

retorna o valor zero porque nenhuma explícita declaração **return** foi encontrada.

Porque todas as funções têm valores, elas podem ser usadas em qualquer declaração aritmética. Por exemplo, programadores iniciantes em C tendem a escrever código como este:

```
x=sqrt(y);
z=sin(x);
```

Um programador mais experiente escreveria o seguinte:

```
z=sin(sqrt(y));
```

Lembre-se de que para um programa determinar o valor da função, ela precisa ser executada. Isto significa que o seguinte código lerá teclas do teclado até que um **u** seja digitado.

```
while((ch=getchar())!='u') ;
```

Este código trabalha porque **getchar( )** deve ser executado para determinar seu valor, que é o caractere digitado no teclado.

A Relação entre Funções e Classes de Variáveis C tem duas classes gerais de variáveis: *global* e *local*. A variável global está disponível para todas as funções do programa, enquanto a variável local é reconhecida e usada somente pela função na qual foi declarada.

Em alguma literatura de C, variáveis globais são referenciadas como variáveis externas e variáveis locais são chamadas de *variáveis dinâmicas ou automáticas*. No entanto, este livro usa os termos global e local porque eles são os termos geralmente aceitos.

Uma variável global deve ser declarada fora de todas as funções, incluindo a função **main( )**. Variáveis globais são normalmente colocadas no topo do arquivo antes de **main( )** porque isto torna o programa mais fácil de ser lido e porque a variável deve ser declarada antes de ser usada. Uma variável local é declarada dentro da função depois da chave de abertura da função. Por exemplo, o seguinte programa declara uma variável global **x**, e duas variáveis locais **x** e **y**.

```
int  x;
main()
{
      int y;
      y= get_value();
      x= 100;
      print ("%d", x, x*y);
}
f1()
{
      int x;

      x= getnum();

      return x;
}
```

Este programa multiplica o número inserido pelo teclado por 100. Note que a variável local x em **f1( )** não tem nenhuma relação com a variável global **x**, porque variáveis locais que têm o mesmo nome de variáveis globais sempre têm prioridade sobre as globais.

Variáveis globais existem durante todo o programa. Variáveis locais são criadas quando a função é introduzida e destruídas quando a função é terminada. Quer dizer que variáveis locais não conservam seu valor entre chamadas de funções. Entretanto, para preservar valores entre funções você pode usar o modificador **static.** (Você verá exemplos do **static** mais adiante neste livro.)

Os parâmetros mais específicos para a função são também variáveis locais, e, exceto para receber o valor dos atributos de chamada, elas comportam-se e podem ser como qualquer outra variável local.

A Função main( ) Como foi dito anteriormente, todos os programas em C precisam ter uma função **main( ).** Quando o processamento é iniciado, **main( )** é a primeira função chamada. Você não pode ter mais do que uma função com o nome **main( ).** Quando **main( )** termina, o programa acaba, e o controle retorna ao sistema operacional.

Os únicos parâmetros permitidos para a função **main( )** são **argc** e **argv.** A variável **argc** armazena o número de *argumentos comando de linha* e a variável **argv** armazena o ponteiro caractere para eles. Argumentos comando de linha são as informações que

você fornece após o nome do programa quando ele é executado. Por exemplo, quando você compila um programa em C, digita.

CC MYPROG.C

onde **MYPROG.C** é o nome do programa que você deseja compilar. Já que a maioria dos compiladores são eles mesmos escritos em C, **MYPROG.C** é passado ao compilador, usando o mecanismo **argc** e **argv**.

O valor de **argc** é sempre pelo menos 1, porque C considera o nome do programa como sendo o primeiro argumento. Se um programa em C utiliza **argc** e **argv, argv** deve ser declarada como um vetor de ponteiros caractere e **argc** como um inteiro. Isto é mostrado no pequeno programa seguinte, que imprime seu nome na tela.

```
main(argc,argv)
int argc;
char *argv[];
{
      if(argc<2)
            printf("entr co se nom n linh d comando);
      else
            printf("hello %s\n",argv[1]);
}
```

Observe que **argv** é declarada como um vetor ponteiro caractere de extensão desconhecida. O compilador C determina automaticamente a extensão do vetor que é necessário para conter todos os argumentos da linha de comando.

O uso de argumentos de comando de linha dão aos programadores uma visão e sentido mais profissional, bem como permitem aos programas serem submetidos a um arquivo batch para uso automático.

Agora que você leu esta revisão, está pronto para explorar problemas de programação C e suas soluções. Lembre-se, o Apêndice A contém um sumário da maior parte das palavras-chaves em C, as diretivas do pré-processador, e algumas das funções da biblioteca-padrão usadas neste livro.

Capítulo 2

ORDEM E BUSCA

No mundo da computação, talvez nenhuma outra tarefa seja mais fundamental ou tão extensivamente analisada como estas de ordenação e busca. Estas rotinas são utilizadas em praticamente todos os programas de banco de dados, bem como em compiladores, interpretadores e sistemas operacionais. Neste capítulo são apresentados os conceitos básicos de ordenação e busca.

Pelo fato da ordenação de dados geralmente fazer uma busca mais fácil e rápida, a ordenação é discutida primeiro.

ORDEM

Ordenação é o processo de arranjar um conjunto de peças parecidas de informação numa ordem crescente e decrescente. Dada uma lista ordenada **i** de **n** elementos,

$$i_1 <= i_2 <= \ldots <= i_n$$

Existem duas categorias de algoritmos de classificação: a ordenação de vetores, ambos na memória ou em acessos randômicos a arquivos em disco, e a ordenação de arquivos em fita ou em disco. Este capítulo focalizará a primeira categoria porque é de maior interesse dos usuários de microcomputadores. No entanto, o método geral de ordenação seqüencial de arquivos também será comentado.

A principal diferença entre ordenação de vetores e ordenação seqüencial de arquivos é que cada elemento de um vetor está sempre disponível. Isto é, qualquer elemento pode ser comparado ou trocado com qualquer outro elemento a qualquer ins-

tante. Em um arquivo seqüencial, porém, somente um elemento está disponível num determinado momento.

Devido a esta diferença, técnicas de ordenação diferem enormemente entre as duas categorias.

Geralmente quando a informação é ordenada, uma pequena porção desta informação é usada como *chave de ordenação* com a qual as comparações são baseadas. Quando uma troca deve ser feita, toda a estrutura de dados é transferida. Em listagem de endereços, por exemplo, o campo de código de área (CEP) pode ser usado como chave, e o nome completo e endereço acompanham o CEP quando uma troca é feita. Com o objetivo de simplificar, exemplos de vários métodos de ordenação apresentados aqui são enfocados em ordenação de vetores caracteres. Mais tarde, você aprenderá como adaptar qualquer um destes métodos a qualquer tipo de estrutura de dados.

CLASSES DE ALGORITMOS DE ORDENAÇÃO

Existem três métodos gerais que podem ser usados para ordenar vetores:

- por troca
- por seleção
- por inserção

Imagine as cartas de um baralho. Para classificá-las *por troca,* você as espalhará, voltadas para cima, numa mesa, e então procederá a troca das cartas fora de ordem até que todo baralho esteja em ordem.

Para ordenação por *seleção,* você espalhará as cartas na mesa e escolherá a de menor valor, e a retirará do baralho. Então, das cartas restantes na mesa, você selecionará a menor, colocando-a atrás da já existente na sua mão. Você sempre escolhe a de menor valor restante na mesa; quando o processo for completado, as cartas em sua mão estarão ordenadas.

Na ordenação por *inserção,* você seguraria as cartas em sua mão, tirando uma por vez. Conforme você tirar as cartas do maço, as colocará em um novo maço sobre a mesa, inserindo-as sempre na posição correta. O maço estará ordenado quando não tiver mais cartas na mão.

UMA AVALIAÇÃO DOS ALGORITMOS DE ORDENAÇÃO

Existem muitos algoritmos para cada um dos três métodos de ordenação. Cada um deles tem seus méritos, mas, de uma forma geral, a avaliação de um algoritmo de ordenação está baseada nas respostas às seguintes perguntas:

- O quanto pode ser rápido, em média, um algoritmo de ordenação?
- Qual a velocidade de seu melhor e pior casos?
- Este algoritmo apresenta um comportamento *natural* ou *não-natural*?
- Ele rearranja elementos com chaves iguais?

Com que velocidade um algoritmo ordena para ter um ótimo desempenho? A velocidade na qual a ordenação de um vetor seja diretamente relacionada ao número de comparações e ao número de trocas exigidas, com as trocas exigindo mais tempo. Mais adiante neste capítulo você verá que algumas ordenações variam o tempo de ordenação de um elemento de forma exponencial, e algumas de forma logarítmica.

Os tempos de processamento no pior e melhor caso são importantes se você desejar saber regularmente quais as situações de melhor e pior caso. Freqüentemente uma ordenação terá um bom caso médio mas um terrível pior caso, ou vice-versa.

Diz-se que uma ordenação tem um comportamento *natural* se ela trabalha o menos possível quando a lista já está ordenada, e quanto mais desordenada estiver a lista mais trabalho terá o algoritmo, e trabalhará o maior tempo quando a lista estiver em ordem inversa. O maior trabalho de uma ordenação é o número de comparações e movimentos que ele deve executar.

Para entender a importância de rearranjar elementos com chaves iguais, imagine um banco de dados que é ordenado de acordo com uma chave principal e uma subchave – por exemplo, uma lista postal com o código CEP como chave principal e o último nome com o mesmo código CEP como subchave. Quando um novo endereço for acrescentado à lista e a lista for novamente ordenada, você não desejará que as subchaves sejam novamente rearranjadas. Para garantir isto, a ordenação não poderá trocar chaves principais de mesmo valor.

Nas seções seguintes, são analisados e avaliados conforme sua eficiência cada um dos algoritmos de ordenação.

A Ordenação Bolha A mais conhecida (e mais difamada) ordenação é a *ordenação bolha*. Sua popularidade vem de seu nome fácil e de sua simplicidade. Porém, é uma das piores ordenações concebidas.

A *ordenação bolha* usa o método de ordenação por trocas. Faz repetidas comparações e, se necessário, troca elementos adjacentes. Seu nome provém da semelhança com a maneira como bolhas de água movimentam-se num tanque, onde cada bolha procura seu nível próprio. Neste mais simples exemplo de *ordenação bolha*

```
bubble(item,count)        /* ordenacao bolha */
char *item;
int count;
{
   register int a,b;
   register char t;

   for (a=1; a < count; ++a)
        for (b=count-1; b>=a; --b)
             {
                  if (item [b-1] > item [b])
                     {
                          /* trocando os elementos */
                          t = item [b-1];
                          item [b-1] = item [b];
                          item [b] = t;
                     }
              }
}
```

item é um ponteiro a um vetor de caracteres a ser ordenado e **count** é o número de elementos no vetor.

A *ordenação bolha* é feita através de dois laços (loops). Uma vez que existem **count** elementos no vetor, o laço mais externo faz o vetor ser varrido **count-1** vezes. Isto assegura que, no pior caso, todos os elementos estarão na sua posição correta quando a função estiver terminada. O loop mais interno faz as comparações e trocas.

Esta versão de *ordenação bolha* pode ser usada para ordenar um vetor de caracteres em ordem ascendente. Por exemplo, este programa ordena uma seqüência de caracteres digitada:

```
main()      /* ordena caracteres digitados atraves do teclado */
{
            char s[80];
            int count;

            printf("entre um caracter:");

            gets(s);

            count=strlen(s);

            bubble(s,count);

            printf("cadeia de caracteres ordenados: %s",s);

}
```

Para ilustrar o funcionamento da *ordenação bolha,* aqui estão os passos para se ordenar o vetor **dcab**

início	d c a b
passo 1	a d c b
passo 2	a b d c
passo 3	a b c d

Na avaliação de qualquer ordenação, você deve determinar quantas comparações serão feitas, para o melhor, médio e pior caso. Para a *ordenação bolha,* o número de comparações é sempre o mesmo porque os dois laços **(for)** serão repetidos um número determinado de vezes, estando ou não a lista inicialmente ordenada. Isto significa que a *ordenação bolha* executará sempre 1/2 (n^2-n) comparações, onde n é igual ao número de elementos a ser ordenado. A fórmula é derivada do fato de o laço mais externo da *ordenação bolha* ser executado $n-1$ vezes e o laço mais interno $n-2$ vezes. Multiplicando-se um pelo outro obtemos a fórmula.

O número de trocas é 0 para o melhor caso – para uma lista já ordenada. Os números são: para o caso médio 3/4 (n^2-n) e para o pior caso 3/2 (n^2-n). Está fora do objetivo deste livro explicar a origem destes casos; você pode observar, porém, que à medida que a lista torna-se menos ordenada, o número de elementos fora de ordem aproxima-se do número de comparações. (Existem três trocas para cada elemento fora de ordem na *ordenação bolha.)* Ela é um *algoritmo n–quadrado* pois seu tempo de execução é um múltiplo do quadrado do número de elementos. A *ordenação bolha* é muito ruim para um número grande de elementos porque o tempo de execução é diretamente proporcional ao número de comparações e trocas.

Por exemplo, se você ignorar o tempo que leva para trocar um elemento qualquer fora de ordem, verá que cada comparação leva 0.001 segundos, então para ordenar 10 elementos levará 0.05 segundos, ordenar 100 elementos levará 5 segundos, e ordenar 1000 elementos levará 500 segundos. Uma ordenação de 100.000 elementos, o tamanho de uma pequena lista telefônica, levaria em torno de 5.000.000 de segundos, ou 1:400 horas, por volta de dois meses de ordenação contínua! O gráfico da Figura 2.1 mostra como o tempo de execução aumenta em relação ao tamanho do vetor.

Você pode fazer umas melhorias para que a *ordenação bolha* fique mais rápida – e melhore um pouco a sua imagem. Por exemplo, a *ordenação bolha* tem uma peculiaridade: um elemento fora-de-ordem no "final longo", tal como o **a** no vetor **dcab** dado como exemplo, irá para sua posição correta em um passo, mas um elemento desordenado no "final curto", como o **d**, subirá vagarosamente para seu lugar apropriado. Em vez de sempre ler o vetor numa mesma direção, podemos alternar a direção entre passos consecutivos. A maior parte dos elementos fora-de-ordem irá mais rapidamente para suas posições corretas. É mostrada aqui uma versão de *ordenação bolha* chamada *ordenação chacoalhadeira*, por causa de seu movimento chacoalhador sobre o vetor:

```
shaker(item,count) /* ordenacao chacoalhadeira */
char *item;
int count;
{
        register int a,b,c,d;
        char t;

        c=1;
        b=count-1; d=count-1;

        do {
                for(a=d; a>=c; --a) {
                        if(item[a-1]>item[a]) {
                                t=item[a-1];
                                item[a-1]=item[a];
                                item[a]=t;
                                b=a;
                        }
                }
                c=b+1;
                for(a=c;a<d+1;++a) {
                        if(item[a-1]>item[a]) {
```

```
                                        t=item[a-1];
                                        item[a-1]=item[a];
                                        item[a]=t;
                                      b=a;
                                }
                        }
                        d=b-1;
                } while (c<=d);
}
```

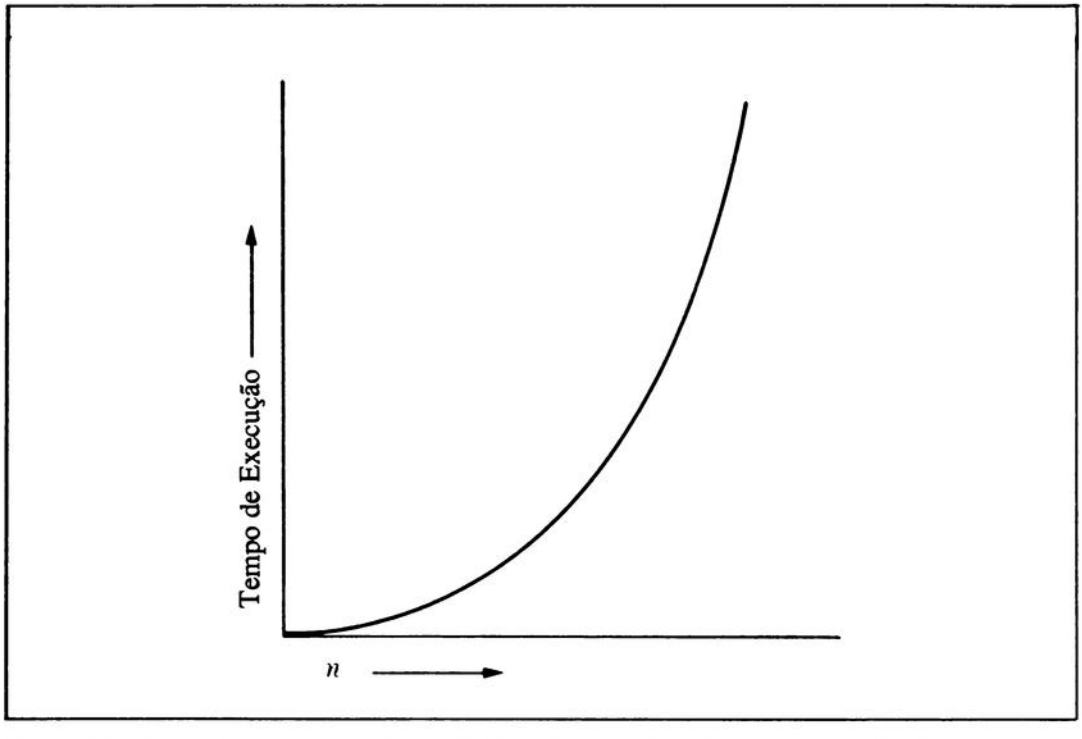


Figura 2.1. Tempo de processamento em relação ao tamanho do vetor numa ordenação n^2.

Embora a ordenação "chacoalhadeira" melhore a *ordenação bolha*, ela ainda é executada na ordem de n^2, porque o número de comparações não é alterado e porque o número de trocas foi reduzido de uma pequena constante.

Ordenação por Seleção A *ordenação por seleção* seleciona o elemento de menor valor e troca-o como primeiro elemento. Então, para os n–1 elementos restantes, o

elemento com o menor valor é encontrado e trocado com o segundo elemento, e assim por diante até os dois últimos. Por exemplo, se o método de seleção tivesse sido usado no vetor **bdac,** cada passo se apresentaria como:

início	b d a c
passo 1	a d b c
passo 2	a b d c
passo 3	a b c d

Uma ordenação por seleção simples é vista aqui:

```
select(item,count)  /* ordenacao por selecao */
char *item;
int count;
{
      register int a,b,c;
      char t;

      for(a=0;a<count-1;++a)  {
              c=a;
              t=item[a];
              for(b=a+1; b<count;++b)  {
                      if(item[b]<t)  {
                              c=b;
                              t=item[b];
                      }
              }
              item[c]=item[a];
              item[a]=t;
      }
}
```

Infelizmente, como na *ordenação bolha,* o laço mais externo é executado n–1 vezes e o laço mais interno $1/2(n)$ vezes. Isto significa que a *ordenação por seleção* requer $1/2(n^2-n)$ comparações, que a tornam muito lenta para um número grande de itens. O número de trocas para o melhor caso é $3(n-1)$ e para o pior caso $n^2/4+3(n-1)$.

Para o melhor caso (a lista está ordenada) somente $n-1$ elementos precisam ser movimentados, e cada movimento requer três trocas. O pior caso aproxima-se do número de comparações. Não obstante o desenvolvimento do cálculo do caso médio esteja fora do objetivo deste livro, é igual a $n(ln\ n+y)$, onde y é a constante de Euler, aproximadamente 0,577216. Isto significa que embora o número de comparações para a *ordenação bolha* e para a *ordenação por seleção* seja o mesmo, o número de trocas no caso médio é muito menor para a *ordenação por seleção.*

Ordenação por Inserção A *ordenação por inserção* é o último dos algoritmos simples. Inicialmente ela ordena os dois primeiros membros no vetor. A seguir, o algoritmo insere o terceiro membro na sua posição ordenada em relação aos dois primeiros membros. Então, o quarto elemento é inserido na lista de três elementos. O processo continua até que todos elementos tenham sido ordenados. Por exemplo, no vetor **dcab**, cada passo da *ordenação por inserção* seria como:

início	d c a b
passo 1	c d a b
passo 2	a c d b
passo 3	a b c d

A versão da *ordenação por inserção* é vista aqui:

```
insert(item,count) /* ordenacao por insercao */
char *item;
int count;
{
      register int a,b;
      char t;

      for(a=1; a<count; ++a) {
            t=item[a];
            b=a-1;
            while(b>=0 && t<item[b] )  {
                  item[b+1]=item[b];
                  b--;
            }
            item[b+1]=t;
      }
}
```

Ao contrário da *ordenação bolha* e da *ordenação por seleção,* o número de comparações que ocorrem enquanto a *ordenação por inserção* é usada depende a rigor de como a lista está inicialmente ordenada. Se a lista estiver em ordem, então o número de comparações é $n-1$. Se a lista estiver fora de ordem, então o número de comparações é $1/2(n^2+n)-1$, enquanto sua média é $1/4(n^2+n-2)$.

O número de trocas para cada caso é o seguinte:

melhor	$2(n-1)$
médio	$1/4(n^2+9n-10)$
pior	$1/2(n^2+3n-4)$

Como se vê, o número para o pior caso é tão ruim quanto aqueles para as ordenações *bolha* e *por seleção*, e este número para o caso médio é somente um pouco melhor.

A *ordenação por inserção* tem certamente duas vantagens. Em primeiro lugar comporta-se com *naturalidade:* trabalha menos quando o vetor já está ordenado e o máximo quando o vetor está na ordem inversa. Isto faz com que a *ordenação por inserção* seja útil para listas que estão quase ordenadas. Em segundo, não rearranja elementos de mesma chave: se uma lista é ordenada com duas chaves, mantém-se ordenada para ambas as chaves após uma *ordenação por inserção.*

ORDENAÇÕES MELHORES

Os algoritmos vistos até agora tiveram a grave falha de serem processados numa grandeza de tempo n^2. Para um grande número de dados, a ordenação será lenta – em certos casos, muito lenta para sua utilização. Todo programador já ouviu ou contou a estória da "ordenação que levou três dias". Infelizmente, estas estórias freqüentemente são verdadeiras.

Quando uma ordenação demora tanto tempo, deve ser uma falha do algoritmo básico. Porém, o pior é que a primeira idéia freqüentemente é "vamos escrevê-lo em assembler". Embora o assembler aumente quase sempre a velocidade de uma constante, se o algoritmo básico for ruim, a ordenação ainda será lenta, não importando o quão bom seja o código. Lembre-se: quando o tempo de uma rotina é n^2, aumentar a velocidade do código ou do computador apenas causa uma melhoria marginal, pois a razão na qual o tempo de execução aumenta é alterada exponencialmente (o gráfico da Figura 2.1 é deslocado para a direita, levemente, mas a curva continua a mesma). Tenha em mente que se alguma coisa não for rápida o bastante em C, não o será em assembler. A solução é usar um algoritmo de ordenação melhor.

Ordenação Shell A *ordenação Shell* é assim chamada por causa do seu inventor, D.L.Shell. Porém, o nome parece ter sido apropriado porque seu método de operação lembra as conchas do mar empilhadas umas sobre as outras.

O método geral, derivado da ordenação por inserção, está baseado em *diminishing increments*. A Figura 2.2 mostra o diagrama da *ordenação Shell* no vetor **fdacbe**. Primeiro, todos os elementos que estão três posições afastados são ordenados. Então todos os elementos que estão duas posições afastados são ordenados. Finalmente, todos aqueles adjacentes um ao outro são ordenados.

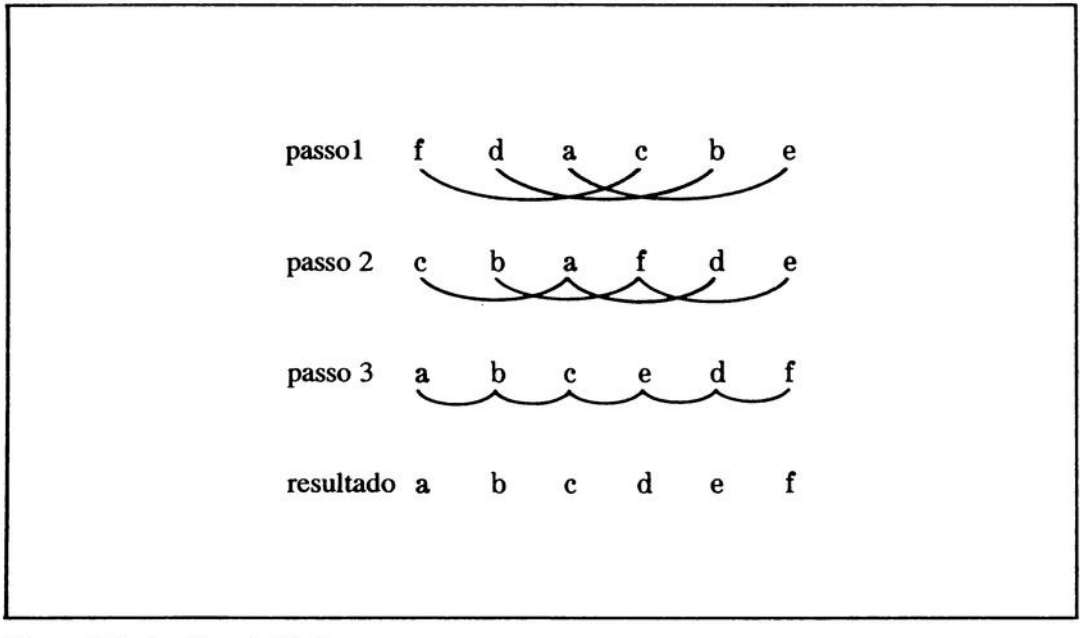


Figura 2.2. A ordenação Shell.

Não parece óbvio que esse método leve a bons resultados, ou mesmo que irá ordenar um vetor, mas ele executa ambas as funções. Este algoritmo é eficiente porque cada passo da ordenação envolve, ou relativamente poucos elementos, ou elementos que já estão em razoável ordem; contudo, cada passo aumenta a ordenação dos dados.

A seqüência exata para os incrementos pode ser alterada. A única regra é que o último incremento deve ser um. Por exemplo, a seqüência 9, 5, 3, 1 funciona e é usada na ordenação Shell mostrada aqui. Evite seqüências de potências de 2 porque, por razões matemáticas complexas, elas reduzem a eficiência do algoritmo de ordenação. (Contudo, mesmo se você usá-las, a ordenação ainda funcionará.)

```
shell(item,count)
char *item;
int count;
{

        register int i,j,k,s,w;
        char x, a[5];

       a[0]=9; a[1]=5; a[2]=3; a[3]=3; a[4]=1;

        for(w=0; w<5; w++) {
                k=a[w]; s=-k;
                for(i=k; i<count; ++i) {
                        x=item[i];
                        j=i-k;
                        if(s==0) { s=-k;
                                s++;
                                item[s]=x;
                        }
                        while(x<item[j] && j>=0 && j<=count)  {
                                item[j+k]=item[j];
                                j=j-k;
                        }
                        item[j+k]=x;
                }
        }
}
```

Você deve ter observado que o laço interno **while** contém três condições de teste. O **x<item[j]** é a comparação necessária para o processo de ordenação. Os testes **j>=0** e **j<=count** são usados para evitar que os limites do array **item** sejam ultrapassados. Estas checagens extras degradarão até certo ponto a performance da ordenação Shell. Versões um pouco diferentes de ordenação Shell empregam elementos especiais, vetores chamados *sentinelas*, que não fazem parte do vetor a ser ordenado. Sentinelas carregam uma terminação particular que indica os elementos, menor e maior possível. Desta maneira, as checagens dos limites são desnecessárias. No entanto, usar sentinelas requer um conhecimento dos dados, o que limita a generalização da função de ordenação. A análise da ordenação Shell apresenta problemas matemáticos difíceis que estão fora dos objetivos deste livro. O tempo de processamento é proporcional a $n^{1.2}$ para uma ordenação de n elementos. Esta é uma redução significante sobre as ordenações n^2 da seção anterior; observe a Figura 2.3 que mostra o gráfico da curva n^2 junto com a curva $n^{1.2}$. Porém, antes que você se decida a usar a *ordenação Shell*, deverá saber que a *ordenação Quicksort* é ainda melhor.

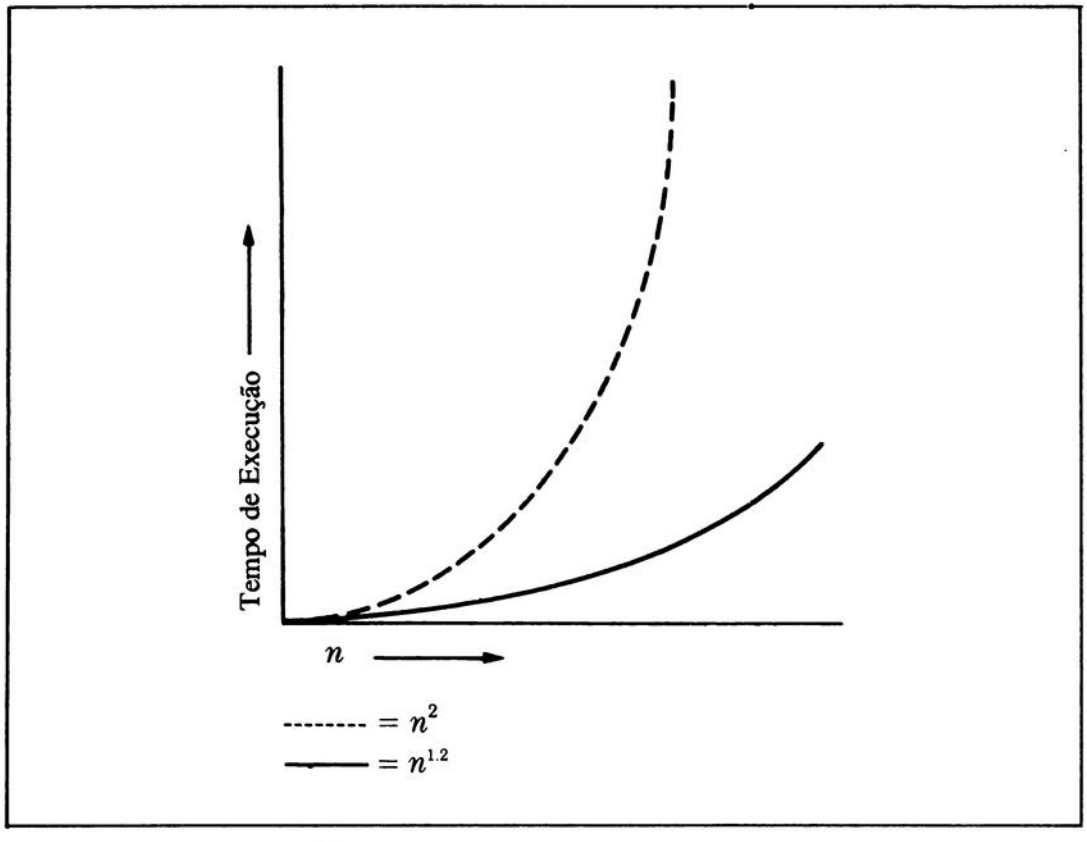


Figura 2.3. As curvas n^2 e $n^{1.2}$.

Ordenação Rápida – Quicksort A *Quicksort*, criada e denominada por C.A.R.Hoare, é geralmente considerada o melhor algoritmo de ordenação, normalmente avaliado. Está baseada no método de ordenação por trocas. Isto é surpreendente se você considerar a péssima performance da *ordenação bolha*, que também está baseada no método de trocas.

A *Quicksort* é construída sobre a idéia das partições. O procedimento mais usual seleciona um valor chamado de *comparador* e então divide o vetor em duas partes, com todos os elementos maiores e iguais ao valor de partição do outro. Esse processo é repetido para cada parte restante até que o vetor é ordenado. Por exemplo, dado o vetor **fedacb** e o valor **d** para partição, o primeiro passo da *Quicksort* rearranjará o vetor como segue:

início	f e d a c b
passo 1	b c a d e f

Este processo é repetido para cada parte **(bca** e **def).** O processo é essencialmente recursivo; certamente, as mais claras implementações da *Quicksort* são algoritmos recursivos.

A escolha do valor do comparador mediano pode ser feita de duas maneiras. O valor pode ou ser escolhido randomicamente ou por média dos valores de um pequeno conjunto retirado do vetor. Para uma ordenação ideal é melhor escolher o valor que é precisamente o meio do escopo de valores. Porém, isto não é fácil para a maioria dos conjuntos de dados. Mesmo no pior caso – o valor escolhido é uma das extremidades – a *Quicksort* ainda assim tem um rendimento razoavelmente bom.

A versão de *Quicksort* a seguir seleciona o elemento médio do vetor. Embora esta forma nem sempre resulte numa boa escolha, é uma técnica simples e rápida, e funciona corretamente.

```
quick(item,count)  /* quick sort */
char *item;
int count;
{

      qs(item,0,count-1);

}

qs(item,left,right)
char *item;
int left,right;
{

      register int i,j;
      char x,y;

      i=left; j=right;
      x=item[(left+right)/2];

      do  {
            while(item[i]<x && i<right) i++;
            while(x<item[j] && j>left) j--;

            if(i<=j)  {
                  y=item[i];
                  item[i]=item[j];
                  item[j]=y;
                  i++; j--;
```

```
        }
    } while(i<=j);

    if(left<j)  qs(item,left,j);
    if(i<right)  qs(item,i,right);
}
```

Aqui **quick( )** executa a chamada para a função principal de ordenação, chamada **gs( )**. Enquanto isto mantém a mesma interface comum a **item** e **count**, não é essencial, porque **qs( )** poderia ter sido chamada diretamente, usando-se três argumentos.

A demonstração de como o número de comparações e o número de trocas que a *Quicksort* executa requer um ambiente matemático fora do escopo deste livro. Porém, você pode assumir que o número de comparações é *n*log *n*, e que o número de trocas é aproximadamente *n*/6 log *n*. Eles são significativamente melhores do que qualquer uma das ordenações vistas até agora.

A equação

$$N = a^x$$

pode ser reescrita como

$$x = \log_a N$$

Isto quer dizer, por exemplo, que se 100 elementos fossem ordenados, *Quicksort* iria precisar em média de 100*2, ou 200, comparações porque log 100 é 2. Comparando com a média de 990 comparações da *ordenação bolha*, este número é muito bom.

No entanto, existe um estranho aspecto em *Quicksort* sobre o qual você deve ser advertido. Se acontecer de o valor do comparador para cada partição ser o valor máximo, então a *Quicksort* degenera-se em uma ordenação lenta com tempo n^2. Geralmente, porém, isto não acontece.

Você deve tomar cuidado ao escolher um método para determinar o valor do comparador. Freqüentemente o valor é determinado pelo valor de um dado real que está sendo ordenado. Em grandes listas postais onde a ordenação é freqüentemente feita através dos códigos CEP, a seleção é simples, porque os códigos são distribuídos ordenadamente e uma função algébrica pode determinar o melhor comparador. Porém, em determinados bancos de dados, as chaves de ordenação possuem valores tão próximos, com muitos tendo o mesmo valor, que a seleção randômica é a melhor disponível. Um método co-

mum e satisfatoriamente eficaz é amostrar três elementos da partição e retirar o de valor médio.

ORDENANDO OUTRAS ESTRUTURAS DE DADOS

Até agora, as ordenações só têm sido aplicadas para vetores de caracteres, tornando mais fácil a apresentação de cada rotina de ordenação. Obviamente, vetores de qualquer dos tipos de dados intrínsecos podem ser ordenados simplesmente trocando-se os tipos de dados dos parâmetros e variáveis na função de ordenação. No entanto, são geralmente tipos de dados complexos como *strings* ou estruturas que precisam ser odenados. (A maior parte das ordenações envolvem uma chave e informação enlaçadas em uma chave.) Para adaptar os algoritmos para ordenar outras estruturas de dados, você precisa trocar ou a parte da comparação ou a parte das trocas, ou ambas. O algoritmo permanecerá basicamente inalterado.

A *ordenação Quicksort* será usada em nossos exemplos por ser considerada uma das melhores rotinas de propósito geral disponíveis até o momento. As mesmas técnicas, no entanto, se aplicarão para qualquer uma das ordenações descritas anteriormente.

Ordenação de Seqüência de Caracteres A maneira mais fácil de ordenar seqüência de caracteres é criar um vetor de ponteiros a caractere para essas seqüências, possibilitando uma indexação fácil, e mantendo a base do algoritmo Quicksort inalterada. A versão para seqüências de caracteres da Quicksort mostrada aqui contém um vetor de ponteiros a caractere que apontam para as seqüências a serem ordenadas. A ordenação rearranja os ponteiros às seqüências, não as seqüências reais na memória. Esta versão ordena seqüências de caracteres em ordem alfabética.

O passo da comparação foi alterado para o uso da função **strcmp( )**, que retorna um número negativo se a primeira seqüência de caracteres for lexograficamente menor que a segunda, 0 se as seqüências forem iguais, ou um número positivo se a primeira seqüência for lexograficamente maior que a segunda. A parte da troca da rotina foi deixada inalterada porque só os ponteiros são trocados – não as seqüências reais. Para trocar as seqüências reais, você teria de usar a função **strcpy( )**.

```
quick_string(item,count)
char *item[];
int count;
{
```

```
        qs(item,0,count-1);

}

qs(item,left,right)
char *item[];
int left,right;
{
        register int i, j;
        char *x, *y;

        i=left; j=right;
        x=item[(left+right)/2];

        do {
                while(strcmp(item[i],x)<0 && i<right) i++;
                while(strcmp(item[j],x)>0 && i>left) j--;

                if(i<=j)  {
                        y=item[i];
                        item[i]=item[j];
                        item[j]=y;
                        i++; j--;
                }
        }  while(i<=j);

        if(left<j)  qs(item,left,j);
        if(i<right)  qs(item,i,right);
}
```

O uso da **strcmp( )** retarda a ordenação por duas razões. Primeiro ela envolve uma função de chamada, que sempre leva tempo; segundo, a **strcmp** executa várias (e algumas vezes muitas) comparações para determinar a relação entre duas seqüências. Se a velocidade é realmente crítica, o código para **strcmp( )** pode ser reescrito através de uma rotina particular mais rápida. No entanto, não existe nenhuma maneira de evitar a comparação entre seqüências de caracteres, desde que esta seja a definição que envolva a tarefa.

Ordenação de Estruturas A maioria dos programas de aplicação que requer ordenação precisará ter um grupo de dados ordenados. Uma lista postal é um excelente exemplo porque o nome, rua, cidade, Estado e código CEP estão todos ligados. Quando esta unidade conglomerada de dados é ordenada, uma chave de ordenação é utilizada, mas a estrutura toda é trocada. Para entender esse processo, você primeiro precisa criar uma estrutura. Para uma lista postal, por exemplo, uma estrutura conveniente é

```
struct    address   {
          char name[40];
          char street[40];
          char city[20];
          char state[3];
          char zip[[10];
};
```

state tem extensão de três caracteres e o código CEP tem extensão de 10 caracteres porque um vetor de caracteres precisa ter extensão de um caractere a mais do que o comprimento máximo de qualquer seqüência de caracteres no sentido de armazenar uma terminação nula.

Assim é razoável arranjar uma mala postal como um vetor de estrutura; assuma para este exemplo que a rotina ordena um vetor de estruturas do tipo **address** pelo código ZIP, como mostrado aqui:

```
quick_structure(item,count)      /* ordenacao de estruturas */

struct address item[];
int count;
{

     qs(item,0,count-1);

}
qs(item,left,right)
struct address item[];
int left,right;
{

     register int i,j;
     char *x,*y;

     i=left; j=right;
     x=item[(left+right)/2].zip;

     do {
          while(strcmp(item[i].zip,x)<0 && i<right)  i++;
          while(strcmp(item[j].zip,x)>0 && j>left)  j--;

          if(i<=j)  {
               swap_all_fields(item,i,j);
               i++; j--;
          }
```

```
    }while(i<=j);

    if(left<j)  qs(item,left,j);
    if(i<right) qs(item,i,right);

}
```

Observe que tanto o código de comparação como o código de troca precisaram ser alterados. Em razão de muitos campos precisarem ser alterados, uma função separada chamada **swap-all-fields( )**, mostrada mais adiante, foi criada.

ORDENAÇÃO DE ARQUIVOS EM DISCOS

Existem dois tipos de arquivos em disco: *seqüenciais* e *de acesso randômico*. Se um arquivo em disco é pequeno o bastante, ele deve ser lido na memória, desta forma as rotinas de ordenação de vetores apresentadas anteriormente podem ordenar esses arquivos mais eficientemente. No entanto, muitos arquivos em disco são muito grandes para serem ordenados facilmente na memória e requerem técnicas especiais.

Ordenação de Arquivos em Disco por Acesso Randômico Utilizado pela maioria das aplicações de banco de dados em microcomputadores, arquivos em disco de acesso randômico têm duas vantagens sobre arquivos em disco seqüenciais. Primeiro, eles são de fácil manutenção, você pode atualizar informações sem ter que copiar toda a lista. Segundo, arquivos em disco de acesso randômico podem ser tratados como um grande vetor no disco, que simplifica bem a ordenação. Aplicar esse método significa que você pode usar a base da Quicksort com modificações para procurar diferentes registros no disco, em vez de ter que indexar um vetor. Diferente de ordenar arquivos em disco seqüenciais, ordenar um arquivo randomicamente significa que um disco cheio não precisa ter espaço para arquivos ordenados e desordenados.

Cada situação de ordenação difere na estrutura de dados exata que é ordenada e na chave que é usada. Todavia, o conceito geral de ordenação de arquivos em disco de acesso randômico pode ser compreendido pelo desenvolvimento de programas de ordenação que ordenam uma estrutura de lista postal, chamada **address,** que foi definida anteriormente. Este programa exemplo assume que o número de elementos é fixado em 100. Mas em aplicações reais, o contador de registros teria de ser dinamicamente mantido. Novamente, a chave de ordenação é o campo de código CEP.

```
#include "stdio.h"
#define NUM_ELEMENTS 100          /* este numero e arbitrario,
                                     o operador podera alterar de
                                     acordo com a sua conveniencia */
struct address  {
        char name[30];
        char street[40];
        char city[20];
        char state[3];
        char zip[10];
}ainfo;

main()
{
        FILE *fp;
        int t;

        if((fp=fopen("mlist","r+"))==0)  {
                printf("arquivo inacessivel para escrita e leitura\n"
                exit(0);
        }

        quick_disk(fp,NUM_ELEMENTS);

        fclose(fp);

        printf("lista ordenada\n");

}

quick_disk(fp,count)     /* quick sort para arquivos randomicos */
FILE *fp;                /* chamada */
long int count;
{
        qs_disk(fp,0l,(long)count-1);
}


qs_disk(fp,left,right)  /* quick sort para arquivos randomicos */
FILE *fp;
long int left,right;
{
        long int i,j;
        char x[100],*y,*get_zip();

        i=left; j=right;

        strcpy(x,get_zip(fp,(long) (i+j)/2));    /* media */.

        do  {
                while(strcmp(get_zip(fp,i),x)<0 && i<right) i++;
                while(strcmp(get_zip(fp,j),x)>0 && j>left) j--;
```

```
            if(i<=j)  {
                    swap_all_fields(fp,i,j);
                    i++; j--;
            }
    } while(i<=j);

    if(left<j)  qs_disk(fp,left,j);
    if(i<right)  qs_disk(fp,i,right);
}



swap_all_friends(fp,i,j)
FILE *fp;
long int i,j;
{

        char a[sizeof(ainfo)],b[sizeof(ainfo)];
        register int t;

        /* primeira leitura */
        fseek(fp,sizeof(ainfo)*i,0);
        for(t=0;t<sizeof(ainfo);++t) a[t]=getc(fp);

        fseek(fp,sizeof(ainfo)*j,0);
        for(t=0;t<sizeof(ainfo);++t) b[t]=getc(fp);

        /* escrita */
        fseek(fp,sizeof(ainfo)*j,0);
        for(t=0;t<sizeof(ainfo);++t) putc(a[t],fp);

        fseek(fp,sizeof(ainfo)*i,0);
        for(t=0;t<sizeof(ainfo);++t) putc(b[t],fp);
}

char *get_zip(fp,rec)
FILE *fp;
long int rec;
{
        char *p;
        register int t;

        p=&ainfo;

        fseek(fp,rec*sizeof(ainfo),0);
        for(t=0;t<sizeof(ainfo);++t) *p++=getc(fp);

        return ainfo.zip;

}
```

Muitas funções de suporte têm sido escritas para ordenar os registros de endereços. Na parte da comparação da ordenação, **get–zip** retorna um ponteiro para o código ZIP do comparador e o registro é conferido. A função **swap-all-fields( )** executa a troca real de dados. Para a maioria dos sistemas operacionais, os pedidos de leitura e escrita causam um grande impacto sobre a velocidade de ordenação. O código, como ele é escrito, força uma busca ao registro **i** e então ao **j**. Enquanto o controlador do acionador de disco está ainda posicionado em **j**, o dado de **i** é escrito. Isto significa que não é necessário para o controlador movimentar-se em grandes distâncias. O dado de **i** terá sido escrito primeiro, e, então, uma nova busca se fará necessária.

Ordenação de Arquivos Seqüenciais Ao contrário dos arquivos de acesso randômico, arquivos seqüenciais geralmente não têm tamanho fixo para registros e podem ser organizados em periféricos de armazenagem que não permitem acesso randômico fácil. Portanto, arquivos em disco seqüenciais são comuns porque em uma aplicação específica adaptam-se melhor a comprimento de registros variáveis ou porque o periférico de armazenamento é seqüencial por natureza. Por exemplo, a maioria dos arquivos de texto é seqüencial.

Embora ordenar arquivos em disco como se eles fossem vetores tenha muitas vantagens, este método não pode ser utilizado com arquivos seqüenciais – não existe maneira de se ter um acesso rápido a um elemento arbitrário qualquer. Por exemplo, não existe nenhuma maneira rápida de buscar registros arbitrários de um arquivo seqüencial gravado em uma fita. Portanto, seria difícil apresentar qualquer dos algoritmos de ordenação de vetores anteriormente descritos para os arquivos seqüenciais.

Existem dois métodos de ordenação de arquivos seqüenciais. O primeiro lê a informação na memória e ordena-a com um dos algoritmos usuais de ordenação de vetores. Embora este seja um método rápido, a memória limita o tamanho do arquivo que pode ser ordenado.

O segundo método, chamado de *ordenação agrupada*, divide o arquivo a ser ordenado em dois arquivos de mesmo comprimento. Usando esses arquivos, a ordenação lê um elemento de cada arquivo, ordena esse par e escreve os elementos em terceiro arquivo em disco. Esse novo arquivo é então dividido, e as duplas ordenadas são unidas em quádruplos ordenados. O novo arquivo é dividido novamente, e o mesmo procedimento prossegue até que a lista esteja ordenada. Por razões históricas, esta ordenação conjunta é chamada de *three-tape merge* porque requer três arquivos (acionadores de fita) cada vez que é utilizado.

Para entender como a *ordenação agrupada* trabalha, considere a seguinte seqüência:

1 4 3 8 6 7 2 5

A primeira divisão produz

1 4 3 8
6 7 2 5

O primeiro agrupamento fornece

1 6 – 4 7 – 2 3 – 5 8

Dividindo-se novamente torna-se

1 6 – 4 7
2 3 – 5 8

O próximo agrupamento dá

1 2 3 6 – 4 5 7 8

A divisão final é

1 2 3 6
4 5 7 8

que resulta

1 2 3 4 5 6 7 8

Como você já deve ter imaginado, o "agrupamento de três fitas" *(tree-tape merge)* requer que cada arquivo seja acessado $\log_2 n$ vezes, onde n é o número total de elementos a serem ordenados.

Temos aqui uma versão simples de ordenação por agrupamento. Esta ordenação assume que o arquivo de entrada é um vetor de caracteres, tal qual um arquivo texto, e que o arquivo possui comprimento de potência par. Você pode facilmente alterar esta versão para ordenar qualquer tipo de arquivo de dados

```
#include "stdio.h"

#define LENGTH 16    /* arbitrario */

main(argc,argv)
int argc;
char *argv[];
{

        FILE *fp1,*fp2,*fp3;

        if((fp1=fopen(argv[1],"rw"))==0) {
                printf("arquivo 1 inacessivel %s\n",argv[1]);
                exit(0);
        }

        if((fp2=fopen("sort1","rw"))==0) {
                printf("arquivo 2 inacessivel \n");
                exit(0);
        }

        if((fp3=fopen("sort2","rw"))==0) {
                printf("arquivo 3 inacessivel \n");
                exit(0);
        }

        merge(fp1,fp2,fp3,LENGTH);

        fclose(fp1);  fclose(fp2);  fclose(fp3);

}

merge(fp1,fp2,fp3,count)
FILE *fp1,*fp2,*fp3;
int count;
{
        register int t,n,j,k,q;
        char x,y;

        for(n=1;n<count;n=n*2) {

                for(t=0;t<count/2;++t) putc(getc(fp1),fp2);
                for(;t<count;++t) putc(getc(fp1),fp3);

                rewind(fp1,fp2,fp3);

                for(q=0;q<count/2;q+=n) {
                        x=getc(fp2);
                        y=getc(fp3);
                        for(j=k=0;;) {
                                if(x<y) {
```

```
                                        putc(x,fp1);
                                        j++;
                                        if(j<n) x=getc(fp2);
                                        else break;

                                }
                                else {
                                        putc(y,fp1);
                                        k++;
                                        if(k<n) y=getc(fp3);
                                        else break;

                                }
                        }
                        if(j<n) {
                                putc(x,fp1);
                                j++;
                        }
                        if(k<n) {
                                putc(y,fp1);
                                k++;
                        }
                        for(;j<n;++j) putc(getc(fp2),fp1);
                        for(;k<n;++k) putc(getc(fp3),fp1);
                }
                rewind(fp1,fp2,fp3);
        }
}

rewind(fp1,fp2,fp3)
FILE *fp1,*fp2,*fp3;
{
        fseek(fp1,(long)0,0);
        fseek(fp2,(long)0,0);
        fseek(fp3,(long)0,0);
}
```

Os três arquivos foram abertos para modo **leitura/escrita**, e **rewind**( ) foi criada para desmontar os arquivos a cada vez. A atribuição **long** é usada com **fseek** porque esta função geralmente requer um inteiro longo para o deslocamento do arquivo.

BUSCA

Bancos de dados funcionam normalmente da seguinte forma: de tempos em tempos, o usuário pode localizar e utilizar o dado de um registro qualquer, tão logo a chave do registro seja conhecida. Existe somente um método de encontrar informação em um arquivo ou vetor desordenado e em um arquivo ou vetor ordenado.

MÉTODOS DE BUSCA

Encontrar informações em um vetor desordenado requer uma busca seqüencial, começando pelo primeiro elemento e parando ou quando o elemento procurado é encontrado ou quando o fim do vetor é alcançado.

Este método deve ser usado em dados desordenados mas também pode ser aplicado para dados ordenados. Se o dado tiver sido ordenado, então a busca binária pode ser usada, o que irá aumentar a velocidade de qualquer busca.

A Busca Seqüencial A busca seqüencial é fácil de ser codificada. A função a seguir faz uma busca em um vetor de caracteres de comprimento conhecido até encontrar o elemento procurado a partir de uma chave específica:

```
sequential_search(item,cout,key)
char *item;
int count ;
char key ;
{

      register int t ;

      for(t=0;t<count;++t)
            if(key==item[t]) return t ;

      return -1;

}
```

Esta função retorna ou o número de indexação da entrada encontrada se existir um, ou –1 se não houver.

Uma ordenação seqüencial clássica testará em média $1/2n$ elementos. No melhor caso, testará somente um elemento e, no pior caso, n elementos. Se a informação está armazenada em disco, o tempo de busca pode ser muito longo. Mas se os dados estiverem desordenados, a busca seqüencial é o único método disponível.

Busca Binária Se o dado a ser encontrado estiver colocado de forma ordenada, então um método superior, chamado *busca binária,* poderá ser usado para encontrar o elemento procurado. O método utiliza "divisão e conferência". Ele primeiro testa o elemento médio; se o elemento é maior do que a chave, ele testa o elemento mediano da primeira metade; caso contrário, ele testa o elemento médio da segunda metade. Esse processo é repetido até que o elemento procurado seja encontrado ou até que não haja mais elementos para testar. Por exemplo, para encontrar o número 4 no vetor **1 2 3 4 5 6 7 8 9,** a busca binária testaria primeiro o elemento mediano, que é o **5**. Como este elemento é maior do que 4, a busca continuaria com a primeira metade ou

1 2 3 4 5

Neste exemplo, o elemento médio é o **3,** e é menor do que 4, então a primeira metade é descartada, e a busca continua com

4 5

Neste momento, o elemento procurado é encontrado.

Na busca binária, o número de comparações no pior caso é $\log_2 n$. Para o caso médio, o número é um pouco melhor; no melhor caso o número é 1.

Você pode usar a seguinte função de busca binária para vetores caracteres para encontrar qualquer estrutura de dados arbitrária, trocando a parte de comparação da rotina:

```
bsearch(item,count,key)  /* busca binaria */
char *item;
int count;
char key;
{

      int low,high,mid;

      low=0; high=count-1;

      while(low<=high)  {
            mid=(low+high)/2;
            if(key<item[mid]) high=mid-1 ;
            else if(key>item[mid]) low=mid+1
            else return mid;  /* achou */

      }
      return -1 ;

}
```

O próximo capítulo explora diferentes métodos de armazenamento e recuperação de dados, que, em alguns casos, podem tornar as tarefas de busca e ordenação muito mais fáceis.

Capítulo 3

FILAS, PILHAS, LISTAS ENCADEADAS E ÁRVORES BINÁRIAS

Programas consistem em *algoritmos* e *estruturas* de dados. Um bom programa é uma combinação de ambos. Escolher e implementar uma estrutura de dados são tarefas tão importantes como as rotinas que manipulam os dados. A maneira como a informação é organizada e acessada é normalmente determinada pela natureza do problema de programação. Desta forma, como programador, você precisa ter em sua "cartola de mágicas" o método correto de armazenar e recuperar dados para cada situação.

A representação real dos dados em um computador é construída "de baixo para cima", começando com os tipos básicos de dados como **char, int** e **float.** Num próximo nível estão os vetores, que são coleções organizadas de tipos básicos de dados. Depois estão as estruturas, que são conglomerados de tipos de dados acessados por um nome. Transcendendo esses aspectos físicos dos dados, o nível final concentra-se na seqüência na qual os dados serão *ordenados* e *acessados*. Em verdade, o dado físico é enlaçado a um "dado de máquina" que controla a maneira como seu programa acessa a informação. Existem quatro dessas máquinas:

- fila
- pilha
- lista enlaçada
- árvore binária

Cada método fornece a solução para uma classe de problemas; cada um é essencialmente um *periférico* que executa uma operação específica de armazenamento e recuperação da informação dada, de acordo com o que lhe foi requisitado. Os métodos utilizam duas operações, *armazenar um item* e *recuperar um item,* onde o item é uma unidade de

informação. Este capítulo mostra-lhe como desenvolver esses métodos para utilizá-los em seus próprios programas.

FILAS

Uma *fila* é uma lista linear de informação que é acessada na ordem *primeiro que entra, primeiro que sai* (algumas vezes chamada FIFO). O primeiro item colocado em uma fila é o primeiro que é recuperado, e assim por diante. Esta ordem é a única forma de armazenar e recuperar; uma fila não permite acesso randômico de algum item específico.

Ação	Conteúdo da Fila
qstore(A)	A
qstore(B)	A B
qstore(C)	A B C
qretrieve( ) returns A	B C
qstore(D)	B C D
qretrieve( ) returns B	C D
qretrieve( ) returns C	D

Figura 3.1. Uma fila em ação.

Filas são comuns no nosso dia a dia. Por exemplo, uma linha em um banco ou em uma lanchonete são filas (exceto quando maus patrões permitem que se fure a fila). Para visualizar como uma fila trabalha, considere duas funções: **qstore( )** e **qretrieve( )** **qstore( )** coloca um item no final da fila, e **qretrieve( )** remove o primeiro item da fila e retorna seu valor. A Figura 3.1 mostra o efeito de uma série dessas operações.

Tenha em mente que uma operação de recuperação remove um item da fila e, se o item não estiver armazenado em algum outro lugar, o destruirá. Portanto, mesmo com o programa em atividade, uma fila pode estar vazia por um certo tempo porque todos os itens foram recuperados.

Filas são usadas em muitas situações de programação, tal qual simulações (discutidas mais adiante num capítulo próprio), distribuição de eventos (como nas cartas PERT ou carta Gant), e armazenamento de E/S.

Por exemplo, considere um programa simples de distribuição de eventos que permite a você inserir um certo número de eventos. Assim que cada evento é executado, é retirado da lista, e o próximo evento é visualizado. Você pode utilizar um programa como este para organizar eventos tais como compromissos diários. Para simplificar o exemplo, o programa usa um vetor de ponteiros para cada evento. A descrição de cada evento está limitada a 256 caracteres e o número de eventos a 100. Primeiro, aqui estão as funções **qstore( )** e **qretrieve( )** que serão usadas no programa de distribuição de eventos:

```
#define MAX_EVENT 100

char *p[MAX_EVENT];
int spos;
int rpos;

qstore(q)
char *q;
{
        if(spos==MAX_EVENT) {
                printf("Lista cheia\n");
                return;
        }
        p[spos]=q;
        spos++;
}

qretrieve()
{
        if(rpos==spos) {
                printf("nao ha eventos para executar.\n");
        }
        rpos++;
        return p[rpos-1];
}
```

Essas funções requerem duas variáveis globais: **spos,** que contém a posição ou índice da próxima posição livre de armazenamento, e **rpos,** que contém o índice do próximo item a recuperar.

Neste programa, a função **qstore()** coloca um ponteiro para um novo evento no fim da lista e checa se a lista está cheia. A função **qretrieve()** retira eventos da fila quando existem eventos a serem executados. Quando um novo evento é escalado, **spos** é incrementado, e quando um evento é completado, **rpos** é incrementado. Na realidade, **rpos** "persegue" **spos** através da fila. A Figura 3.2 mostra a maneira como esse processo aparece na memória enquanto o programa é executado. Se **rpos** for igual a **spos,** não existirão

mais eventos a executar. Lembre-se de que embora a informação da fila não seja realmente destruída pela função **qretrieve( )**, ela nunca mais poderá ser acessada novamente e tudo se passa como se ela tivesse sido destruída.

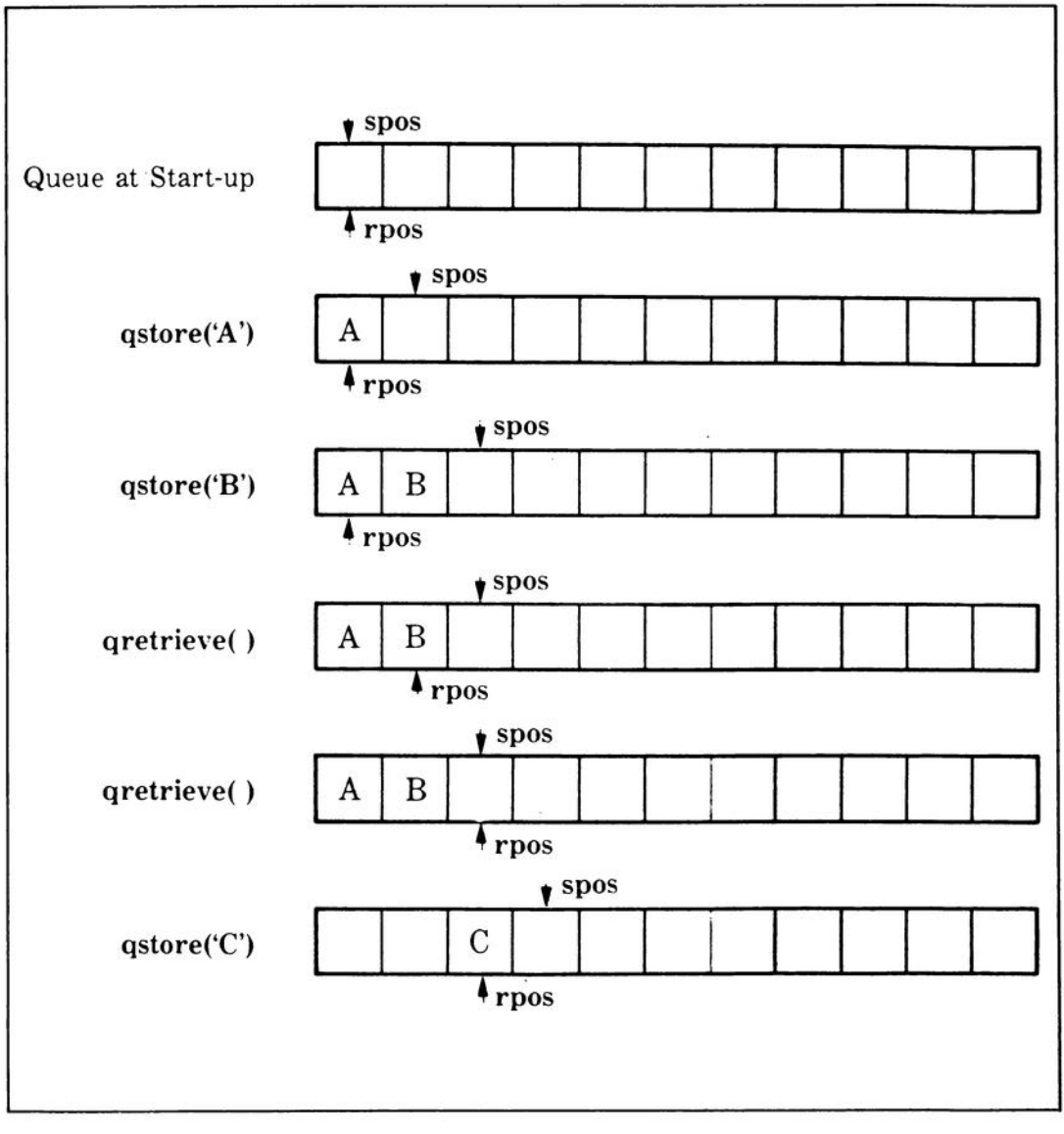


Figura 3.2. O indexador de recuperação persegue o indexador de armazenamento.

Aqui está um programa completo de distribuição de eventos. Você poderá adaptar esse programa para seu uso próprio.

```
#define MAX_EVENT 100

char *p[MAX_EVENT];
int spos;
int rpos;

main()
{
        char s[80];
        register int t;

        for(t=0;t<MAX_EVENT;++t) p[t]=0;
        /* inicializa c/ zeros */
        spos=0; rpos=0;

        for(;;) {
                printf("entrar, rever, fazer, sair: ");
                gets(s);
                *s=toupper(*s);

                switch(*s) {
                        case 'E':
                                enter();
                                break;
                        case 'R':
                                review();
                                break;
                        case 'F':
                                perform();
                                break;
                        case 'S':
                                exit(0);
                }
        }
}

enter()
{
        char s[256], *p, *malloc;

        do {
                printf("entre com um evento %d:",spos+1);
                gets(s);
                if(*s==0) break;          /* nao entra */
                p=malloc(strlen(s));
                if(p==0) {
                        printf("estouro de memoria.\n");
                        return;
                }
                strcpy(p,s);
                if(*s) qstore(p);
```

```
        } while(*s);
}

review()
{

        register int t;

        for(t=rpos;t<spos;++t)
                printf("%d. %s\n",t+1,p[t]);
}

perform()
{
        char *p;

        if((p=qretrieve())==0) return;

        printf("%s\n",p);
}

qstore(q)
char *q;
{
        if(spos==MAX_EVENT) {
                printf("Lista cheia\n");
                return;
        }
        p[spos]=q;
        spos++;
}

qretrieve()
{
        if(rpos==spos) {
                printf("nao ha eventos para fazer.\n");
                return 0;
        }
        rpos++;
        return p[rpos-1];
}
```

A FILA CIRCULAR

Você deve ter pensado em um melhoramento para o programa Mini-Scheduler da seção anterior. Em vez de ter uma parada do programa quando ele encontra o limite do vetor utilizado para armazenar a fila, você poderia ter tanto o laço indexador de armazenamento **spos** como o laço indexador de recuperação **rpos** retornando ao início do vetor. Isto permitiria armazenar qualquer número de itens na fila, contanto que itens fossem

também retirados. Chamada de *fila circular*, essa implementação de fila utiliza seu vetor de armazenamento como se fosse um círculo em vez de uma lista linear.

Para criar uma fila circular para o programa Mini-Scheduler, as funções **qstore()** e **qretrieve()** precisam ser alteradas.

```
qstore(q)
char *q;
{
        if(spos+1==rpos) {
                printf("Lista cheia\n");
                return;
        }
        p[spos]=q;
        spos++;
        if(spos==MAX_EVENT) spos=0;      /* voltar p/ inicio */
}

qretrieve()
{
       if(rpos==MAX_EVENT) rpos=0;      /* voltar p/ inicio */
        if(rpos==spos) {
                printf("nao ha eventos para fazer.\n");
                return 0;
        }
        rpos++;
        return p[rpos-1];
}
```

Na verdade, a fila só enche quando o indexador de armazenamento e o indexador de recuperação são iguais; de outra forma a fila ainda possui espaço para outro evento. Assim, isso significa dizer que quando o programa se inicia, o indexador **rpos** não pode ser 0, mas preferivelmente igual a **MAX–EVENT**, de forma que a primeira chamada a **qstore( )** não forneça uma messagem de fila cheia. É importante notar que a fila armazenará somente **MAX–EVENT**–1 elementos porque **rpos** e **spos** devem ser defasados de pelo menos um elemento; de outra forma, seria impossível saber se a fila estava cheia ou vazia. A Figura 3.3 é a representação conceitual do vetor usado para a versão circular do programa Mini-Scheduler. O uso mais comum para a versão de fila circular pode ser em sistemas operacionais que armazenam informações lidas ou escritas em arquivos em disco ou no console. Outro uso comum é em programas de aplicação em tempo real nos quais, por exemplo, o programa executa outra tarefa enquanto o usuário continua a fazer entradas pelo teclado.

Muitos processadores de palavras fazem isto quando reformatam um parágrafo ou ajustam um alinhamento. Existe um pequeno período no qual o que está sendo digitado não é visualizado até que o outro procedimento ativo não tenha sido completado; o programa de aplicação deve continuar a checar as entradas por teclado durante a execução do outro procedimento. Se uma tecla é digitada, é rapidamente colocada na fila e o procedimento continua. Após ser completado o procedimento, os caracteres são retirados da fila e manipulados de maneira normal.

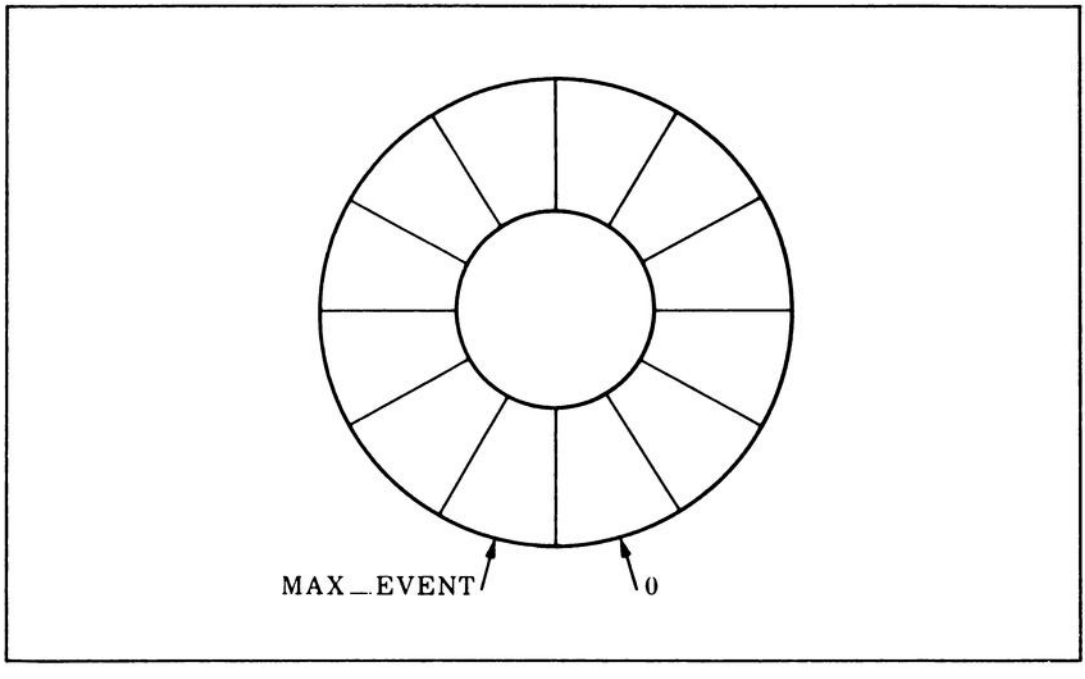


Figura 3.3. O vetor circular para o programa Mini-Scheduler.

Para ver como isto é feito, você pode estudar o programa apresentado a seguir, que contém dois processos. O primeiro imprime números de 1 a 32000 na tela. O segundo coloca caracteres em uma fila circular assim que são digitados, sem ecoá-los na tela até que um ponto-e-vírgula seja teclado. Os caracteres digitados não serão visualizados, porque neste instante o primeiro procedimento tem prioridade sobre a tela. Após o ponto-e-vírgula ser teclado, os caracteres na fila são restaurados e impressos. Note que cada sistema operacional e compilador C diferem-se na maneira de checar o status do teclado. Não existe função de biblioteca para fazer isto.

O pequeno programa aqui exemplificado trabalha com o IBM PC e utiliza a função **bdos( )** do Aztec C. Esta função especial permite ao programa usar rotinas disponíveis no sistema operacional do computador – neste caso, PC-DOS. Uma dessas chamadas ao sistema operacional determina e retorna o status do teclado, assim como qualquer caractere que tenha sido digitado. O programa usa a chamada para ler os caracteres sem ecoá-los na tela. Procure no manual do seu compilador C para saber como são feitas essas chamadas de função.

```
#include "stdio.h"
#define MAX 80
char buf[MAX+1];
int pos=0;

main()  /* exemplo de lista circular - buffer do teclado */
{
        register char ch;
        int dx,dy,t;

        buf[80]=0;

        for(t=0;t<32000 && ch!=';';++t) {
                if(kbhit()) {
                        ch=bdos(7,dx,dy);
                        qstore(ch);
                }
                printf("%d ",t);
        }
        while((ch=qretrieve())!=0) putchar(ch);
}

qstore(ch)
register char ch;
{

        if(pos==MAX) pos=0;      /* volta p/ o inicio */
        buf[pos]=ch;
        pos++;
}

qretrieve()
{

        static int dq_pos=MAX;

        dq_pos++;
        if(dq_pos>MAX) dq_pos=0;          /*volta p/ o inicio */
```

```
        if(dq_pos==pos) {
                dq_pos--;
                return 0;
        }
        return buf[dq_pos];
}

kbhit()
{

        register int dx,dy;
        return(bdos(11,dx,dy));
}
```

kbhit( ) utiliza uma função de chamada ao sistema operacional, que retorne "VERDADE" quando uma tecla for digitada; caso contrário, retorna "FALSO". A chamada **bdos(7,dx,dy)** dentro de **main( )** lê a tecla sem ecoá-la na tela. Como mencionado anteriormente, essas chamadas funcionam somente para o IBM PC. (No Capítulo 5, você aprenderá a usar essa e outras chamadas ao sistema operacional mais detalhadamente.)

PILHAS

Uma *pilha* é o contrário de uma fila porque usa o acesso *último que entra, primeiro que sai* (também chamado LIFO). Imagine uma pilha de pratos. O prato da base da pilha é o último a ser usado, e o prato do topo (o último prato a ser colocado na pilha) é o primeiro a ser usado. Pilhas são usadas em grande quantidade em software de sistemas, incluindo compiladores e interpretadores. A maioria dos compiladores C usa uma pilha quando passa comandos para as funções.

Por razões históricas, as duas operações primárias de pilha – *armazenamento* e *recuperação* – são normalmente chamadas *push* e *pop*, respectivamente. Desta forma, para implementar uma pilha, você precisa de duas funções: **push( )** que coloca um valor na pilha, e **pop( )**, que recupera o valor do topo da pilha. Você também precisa de uma região de memória para usar como pilha: você poderia ou usar um vetor, ou alocar uma região de memória, utilizando-se das funções de alocação de memória dinâmica do C. Como nas filas, as funções de recuperação retiram um valor da lista e, se o valor não estiver armazenado em algum outro lugar, será destruído. Aqui estão os formatos gerais de **push( )** e **pop( )** que usam um vetor inteiro:

```
int stack[MAX];
int tos=0;        /* topo da pilha */

push(i) /* colocar um elemento na pilha */
int i;
{
        if(tos>=MAX)  {
                printf("estouro de pilha\n");
                return;
        }
        stack[tos]=i;
        tos++;
}

pop()   /* retira o primeiro elemento do topo da pilha */
{
        tos--;
        if(tos<0)  {
                printf("underflow da pilha\n");
               return 0;
        }
        return stack[tos];
}
```

Ação	Conteúdo da Pilha
push(A)	A
push(B)	B A
push(C)	C B A
pop( ) retrieves C	B A
push(F)	F B A
pop( ) retrieves F	B A
pop( ) retrieves B	A
pop( ) retrieves A	*cheia*

Figura 3.4. Funcionamento da pilha.

A variável **tos** é o índice da próxima posição livre da pilha. Quando implementar estas funções, lembre-se *sempre* de prevenir o armazenamento na pilha, ultrapassando os seus limites *(overflow)* e a tentativa de recuperar dados depois de ter esvaziado a pilha *(underflow)*. Nesta rotina, se **tos** for igual a 0, a pilha estará vazia; se **tos** for maior do que a última posição de memória, a pilha estará cheia. A Figura 3.4 mostra como funciona uma pilha.

Um excelente exemplo do uso de pilha são as quatro funções de calculadora. A maioria das calculadoras hoje em dia aceita um formato-padrão de expressão chamado de *notação "infix"*, que leva a forma geral *operando-operador-operando*. Por exemplo, para somar 100 a 200, você inseriria **100,** pressionaria a tecla +, digitaria **200** e pressionaria o sinal de igual. Porém, algumas calculadoras utilizam uma expressão de avaliação chamada *notação "postfix" (posfixada)* na qual ambos os operandos entram antes do operador. Por exemplo, para somar 100 a 200, usando posfixado, você digitaria primeiro **100**, então **200**, e aí pressionaria a tecla +. Conforme os operandos vão sendo inseridos, vão sendo armazenados na pilha; quando um operador é inserido, dois operandos são restaurados da pilha e o resultado é armazenado novamente na pilha. A vantagem do formato posfixado é que expressões complexas podem ser calculadas facilmente pela calculadora sem utilizar muito código.

Antes de mostrar uma calculadora completa com quatro funções para expressões posfixadas, você precisa modificar as funções **push( )** e **pop( )**. As rotinas de alocação de memória dinâmica do C podem atribuir uma área de memória para a pilha. Estas funções, como serão usadas mais adiante no exemplo da calculadora, são mostradas aqui.

```
int *p;         /* aponta para regiao de memoria livre */
int *tos; /* ponteiro para o topo da pilha */
int *bos; /* ponteiro para a base da pilha */

push(i) /* coloca na pilha */
int i;
{

        if(p>bos) {
                printf("pilha cheia\n");
                return;
        }
        *p=i;
        p++;
}

pop()   /* retira do topo da pilha */
{
        p--;
        if(p<tos) {
                printf("underflow da pilha\n");
                return 0;
        }
        return *p;
}
```

Antes de você poder usar essas funções, precisa usar **malloc( )** – algumas vezes chamada **alloc( )** por certos compiladores – para alocar uma região livre de memória; você também precisa assumir o endereço inicial desta região para **tos** e assumir o endereço final para **bos.**

O programa completo da calculadora é visto aqui:

```
int *p; /* aponta para regiao de memoria livre */
int *tos; /* ponteiro para o topo da pilha */
int *bos; /* ponteiro para a base da pilha */

main()
{
        int a,b;
        char s[80], *malloc;

        p=malloc(100);  /* aloca memoria para pilha */
        tos=p;
        bos=p+(100/sizeof(int))-sizeof(int);

        printf("quatro operacoes da calculadora\n");

        do {
                printf(": ");
                gets(s);
                switch(*s) {
                        case '+':
                                a=pop();
                                b=pop();
                                printf("%d\n",a+b);
                                push(a+b);
                                break;
                        case '-':
                                a=pop();
                                b=pop();
                                printf("%d\n",b-a);
                                push(b-a);
                                break;
                        case '*':
                                a=pop();
                                b=pop();
                                printf("%d\n",b*a);
                                push(b*a);
                                break;
                        case '/':
                                a=pop();
                                b=pop();
                                if(a==0) {
```

```
                                        printf("divisao por 0\n");
                                        break;
                                }
                                printf("%d\n",b/a);
                                push(b/a);
                                break;
                        default:
                                push(atoi(s));
                }
        } while(*s!='q');
}

push(i) /* coloca na pilha */
int i;
{

        if(p>bos) {
                printf("pilha cheia\n");
                return;
        }


        *p=i;
        p++;
}

pop()   /* retira do topo da pilha */
{
        p--;
        if(p<tos) {
                printf("underflow da pilha\n");
                return 0;
        }
        return *p;
}
```

LISTAS ENCADEADAS

Filas e pilhas dividem características comuns. Primeiro, ambos têm regras específicas para referenciar estruturas de dados. Segundo, operações de restauração são por natureza *destrutivas*, isto é, acessar um item em uma pilha ou fila requer sua retirada e, a menos que seja salvo em algum outro lugar, é destruído. Tanto pilhas como filas requerem, pelo menos conceitualmente, uma região contínua de memória para operar.

Diferente das pilhas e filas, uma *lista encadeada* pode acessar seu buffer de

modo randômico, porque cada informação carrega consigo *um enlace* ao próximo item da corrente. Uma lista encadeada requer uma estrutura de dados complexa, ao contrário das pilhas e filas que podem operar tanto com dados simples ou complexos. (Um enlace é algumas vezes chamado *ponteiro* mas, como a linguagem C usa a palavra ponteiro para descrever um elemento totalmente diferente, este livro utiliza a palavra *enlace.)* Uma operação de recuperação em uma lista encadeada não destrói um item da lista; para se fazer uma deleção é preciso *acrescentar* uma operação específica para isso.

Listas encadeadas são usadas para dois propósitos. O primeiro é para criar vetores de extensão de memória desconhecida. Se você sabe previamente a quantidade a armazenar, pode usar um vetor *simples;* mas se você não sabe o tamanho real da lista, então deve usar uma lista encadeada. O segundo uso de uma lista encadeada é para armazenamento de banco de dados em arquivos em disco. Uma lista encadeada permite-lhe inserir e deletar itens rápida e facilmente, sem rearranjar todo o arquivo em disco.

Por essa razão, listas encadeadas são usadas extensivamente em gerenciamento do software de banco de dados.

Listas encadeadas podem ser singularmente encadeadas ou duplamente encadeadas. Uma *lista singularmente encadeada* contém um elo ao próximo item de dados. Uma *lista duplamente encadeada* contém elos tanto para o próximo elemento como para o elemento anterior da lista. O tipo que você utilizar depende de sua aplicação.

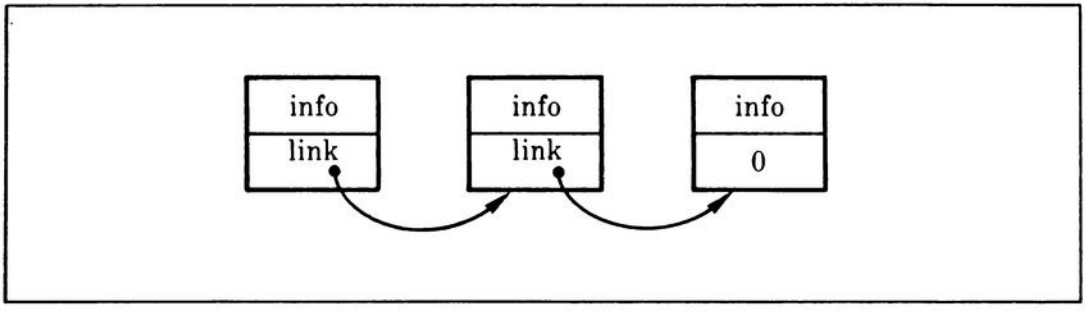


Figura 3.5. Lista singularmente encadeada na memória.

LISTAS SINGULARMENTE ENCADEADAS

Uma lista singularmente encadeada requer que cada item de informação contenha um elo ao próximo elemento da lista. Cada item de dado geralmente consiste em

uma estrutura que contém campos de informação e um ponteiro de enlace. O conceito de uma lista singularmente encadeada é visto na Figura 3.5.

Existem duas maneiras de construir uma lista singularmente encadeada. A primeira simplesmente acrescenta cada novo item ao início ou ao fim da lista. O outro acrescenta itens em lugares específicos da lista (por exemplo, em ordem crescente). A maneira como você construiu sua lista determina a maneira como a *função armazenamento* será codificada, como pode ser visto no caso simples de criação de uma lista encadeada onde são acrescentados itens ao final da lista.

Primeiro você precisa definir uma estrutura de dados que contenha a informação e os enlaces. Por ser comum o uso de listas postais, o exemplo a utiliza. A estrutura de dados para cada elemento em uma lista postal é a mesma do Capítulo 2, exceto que o enlace tem de ser acrescentado.

```
struct address  {
        char name[40];
        char street[40];
        char city[20];
        char state[3];
        char zip[10];
        struct address *next;
} info;
```

A função **slstore()** constrói uma lista singularmente encadeada, colocando cada elemento novo no fim. Um ponteiro para uma estrutura do tipo **address** deve ser passado para **slstore()** como é visto aqui:

```
slstore(i)
struct address *i;
{
        static struct address *last=0;  /* comeca com link nulo */

        if(last==0) last=i;      /* primeiro item da lista */
        else last->next=i;
        i->next=0;
        last=i;
}
```

Observe o uso da variável **static last:** pelo fato da inicialização de uma **static** ocorrer uma vez no início do programa, uma **static** pode ser usada para começar o processo de montagem da lista. Se o seu compilador C não aceita nem **static** nem inicializa-

dores, então você precisa fazer **last** como uma variável global e inicializar explicitamente no **main( )**.

Embora você possa ordenar a lista criada com **slstore( )** como uma operação separada, é mais fácil ordenar enquanto a lista é construída pela inserção de cada novo item na seqüência apropriada da corrente. Além disso, se a lista já estiver ordenada, é vantagem mantê-la ordenada pela inserção de novos itens em suas posições apropriadas. Para fazer isso, a lista é constantemente varrida até que a posição apropriada seja encontrada, o novo endereço seja inserido neste ponto e os enlaces sejam rearranjados como necessário.

Três possíveis situações podem ocorrer quando você insere um item em uma lista singularmente encadeada. Primeiro, o item pode tornar-se o novo primeiro item; segundo, um item pode ser inserido entre dois outros itens; terceiro, o item pode tornar-se o último elemento. A Figura 3.6 mostra como os enlaces são trocados para cada caso.

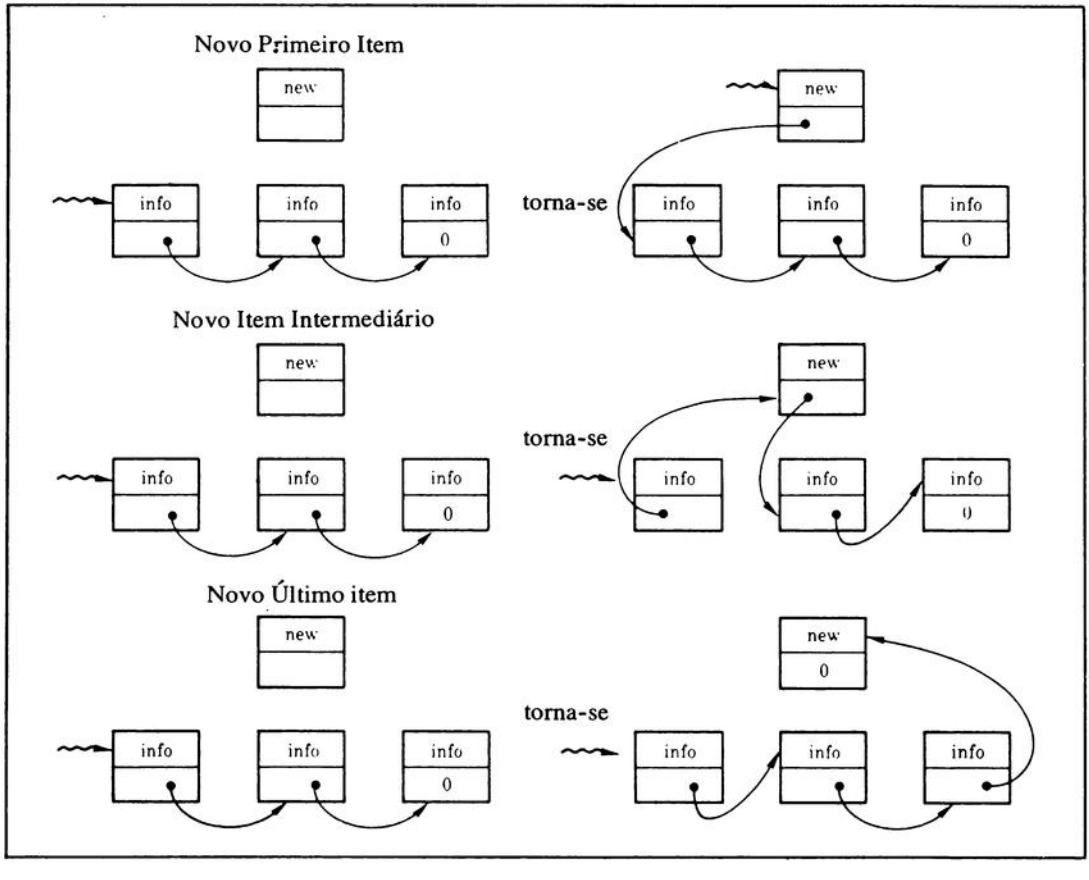


Figura 3.6. Inserção de um item em uma lista singularmente encadeada.

Se você trocar o primeiro item da lista, precisará atualizar o ponto de entrada para a lista em outro ponto de seu programa. Para evitar isto, você pode usar um *sentinela* como primeiro item. Um sentinela é um valor especial que sempre, sobre todas as circunstâncias, será o primeiro na lista. Usando este método, você pode manter o ponto de entrada da lista. Porém, esse método tem a desvantagem de usar uma posição extra de armazenamento que contém o sentinela.

A função **sls-store( )**, mostrada aqui, insere endereços numa lista postal em ordem crescente baseada no campo **nome.** Ela retorna um ponteiro para o primeiro elemento na lista e também requer que o ponteiro para o início da lista seja passado para ela. Quando o primeiro elemento é inserido, tanto **top** quanto **i** são os mesmos.

```
struct address *sls_store(i,top)        /* armazena ordenado */
struct address *i;      /* novo elemento para armazenar */
struct address *top;    /* comeco da lista */
{
        static struct address *last=0;  /* comeca com link nulo */
        struct address *old,*start;

        start=top;

        if(last==0)  {  /* primeiro elemento da lista */
                i->next=0;
                last=i;
                return i;
        }

        old=0;
        while(top)  {
                if(strcmp(top->name,i->name)<0)  {
                        old=top;
                        top=top->next;
                }
                else  {
                        if(old)  {
                                old->next=i;
                                i->next=top;
                                return start;
                        }
                        i->next=top;    /* novo primeiro elemento */
                        return i;
                }
        }
        last->next=i;   /* coloca no final */
        i->next=0;
        last=i;
        return start;
}
```

Em uma lista encadeada não é comum encontrar uma função especificamente dedicada ao *processo de recuperação,* que retorna item por item na ordem da lista. A codificação é normalmente tão pequena que é simplesmente colocada dentro de outra rotina, assim como uma busca, deleção, ou uma função de visualização no vídeo. Por exemplo, a rotina seguinte mostra todos os nomes de uma lista postal.

```
display(top)
struct address *top;
{
        while(top)  {
                printf(top->name);
                top=top->next;
        }
}
```

Aqui, **top** é um ponteiro para a primeira estrutura da lista, e **top** precisa ser inicializado com 0 em algum lugar do programa. A recuperação de itens de uma lista é tão simples como percorrer uma corrente. A rotina de busca baseada no campo **nome** poderia ser escrita assim:

```
struct address *search(top,n)
struct address *top;
char *n;
{
        while(top)  {
                if(!strcmp(n,top->name)) return top;
                top=top->next;
        }
        return 0;       /* nao encontrou */
}
```

Como **search( )** retorna um ponteiro para o item da lista que confere com o nome buscado, **search( )** precisa ser declarado como retornando um ponteiro de estrutura do tipo **address.** Se não houver o elemento procurado, a função devolve um nulo.

O processo de deleção de um item de uma lista singularmente encadeada é feito diretamente. Como com a inserção, existem 3 casos: deletar o primeiro item, deletar um item intermediário e deletar o último item. A Figura 3.7 mostra cada caso.

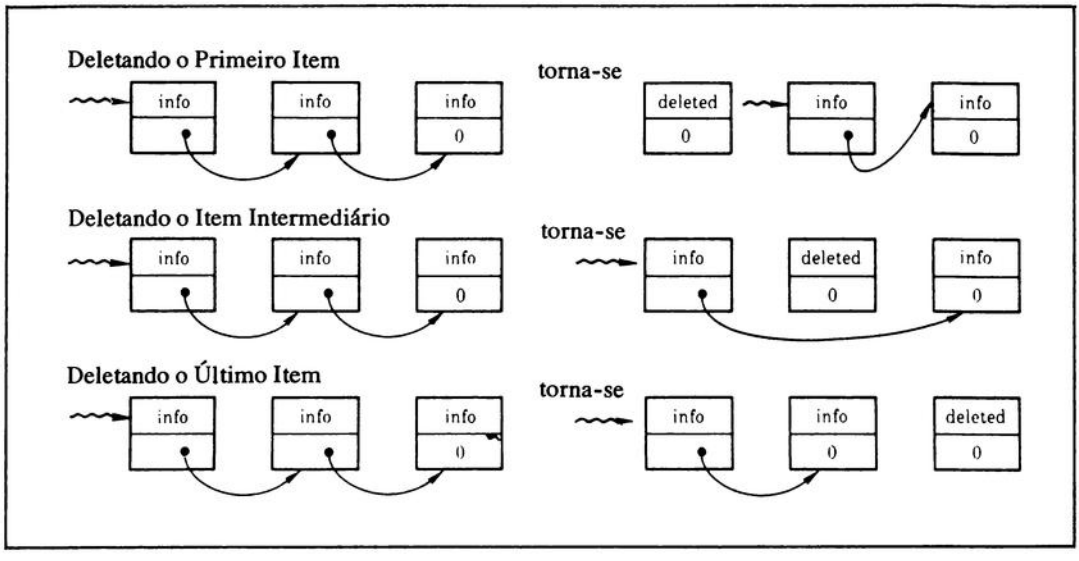


Figura 3.7. Os casos de deleção de itens de uma lista singularmente encadeada.

Esta função deleta um item dado de uma lista de estruturas do tipo **address:**

```
struct address *sldelete(p,l,top)
struct address *p;      /* item anterior */
struct address *l;      /* item a deletar */
struct address *top;    /* comeco da lista */
{
        if(p) p->next=l->next;
        else top=l->next;

        return top;
}
```

Ponteiros ao item deletado, ao item anterior a este na seqüência, e o início da lista devem ser enviados por **sldelete( ).** Se o primeiro item será removido, então o ponteiro ao item anterior deverá ser 0. A função deve retornar um ponteiro ao início da lista porque, no caso do primeiro item da lista ser deletado, o programa precisa saber onde o novo primeiro elemento da lista deve ser colocado.

Listas singularmente encadeadas têm um empecilho maior que previne seu uso

extensivo: a lista não pode ser seguida em ordem inversa. Por esta razão, listas duplamente encadeadas são geralmente utilizadas.

LISTAS DUPLAMENTE ENCADEADAS

Listas duplamente encadeadas consistem em dados e elos para o próximo item e para o item precedente. A Figura 3.8 mostra como esses enlaces são feitos. Uma lista que tem duas ligações em vez de só uma tem duas grandes vantagens. Primeiro, a lista pode ser lida em ambas as direções. Isto não só simplifica a ordenação de uma lista como também, no caso de banco de dados, permite ao usuário varrer a lista em ambas as direções. Segundo, pelo fato de tanto o elo de avanço como o elo de retrocesso poder ler toda a lista, se um dos elos tornar-se inválido, a lista poderá ser reconstruída usando o outro elo. Isto é significativo somente no caso de uma falha no equipamento.

Três operações primárias podem ser efetuadas em uma lista duplamente encadeada: inserir um novo primeiro elemento, inserir um novo elemento intermediário e inserir um novo último elemento. Estas operações são vistas a seguir na Figura 3.9.

Construir uma lista duplamente encadeada é semelhante à montagem de uma lista singularmente encadeada, exceto que a estrutura deve ter espaço para manter dois enlaces. Usando o exemplo da lista postal novamente, você pode modificar a estrutura **address** como mostrado aqui, que torna-se

```
struct address  {
        char name[40];
        char street[40];
        char city[20];
        char state[3];
        char zip[10];
        struct address *next;
        struct address *prior;
} info;
```

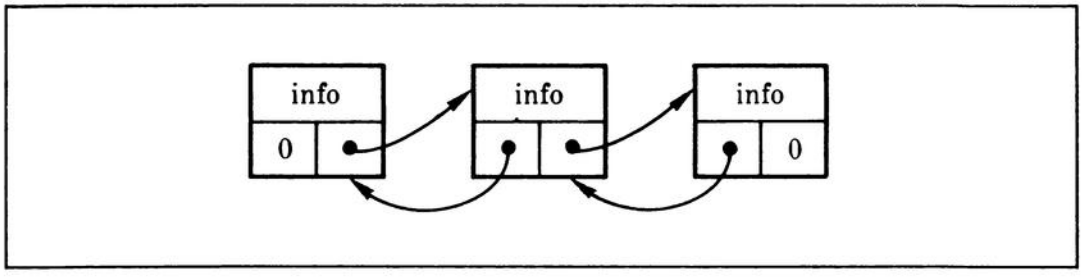


Figura 3.8. Lista duplamente encadeada.

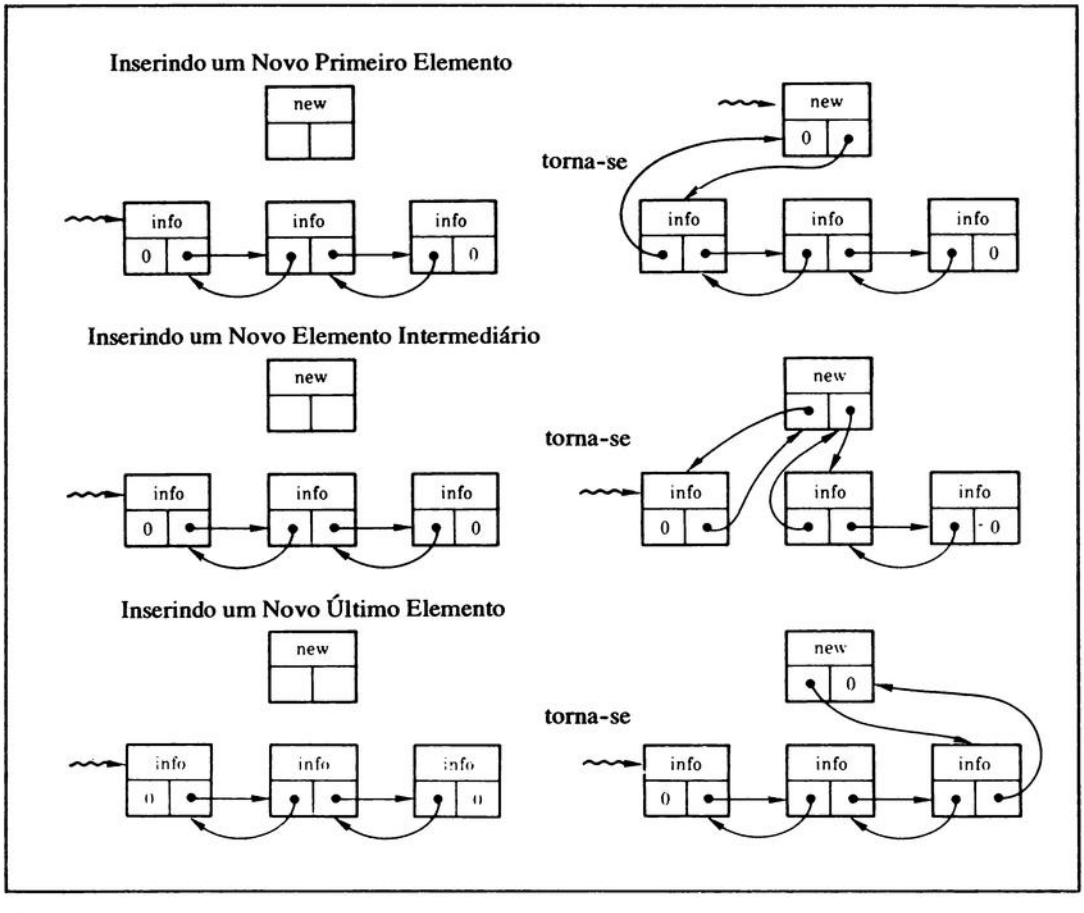


Figura 3.9. As três operações primárias que podem ser executadas em uma lista duplamente encadeada.

Usando a estrutura **address** como o item de dado básico, a função **dlstore( )** constrói uma lista duplamente encadeada:

```
dlstore(l)
struct address *l;
{
        static struct address *last=0;  /* inicia com link nulo */
        if(last==0) last=l;      /* primeiro item da lista */
        else last->next=l;
        l->next=0;
        l->prior=last;
        last=l;
}
```

Esta função coloca cada nova entrada no final da lista.

Como a lista singularmente encadeada, uma lista duplamente encadeada pode ter a função que armazena cada elemento na sua posição específica na lista enquanto a lista está sendo construída, em vez de sempre colocar cada novo item no fim. A função **dls-store( )** cria uma lista que é ordenada em ordem crescente.

```
struct address *dls_store(l,top)        /*armazena em ordem */
struct address *l;      /* novo elemento */
struct address *top;    /* primeiro elemento na lista */
{
        static struct address *last=0;  /* comeca com link nulo */
        struct address *old,*p;

        if(last==0) {   /*primeiro elemento na lista */
                l->next=0;
                l->prior=0;
                last=l;
                return l;
        }

        p=top;  /* comeca no topo da lista */

        old=0;
        while(p) {
                if(strcmp(p->name,l->name)<0)
                        { /* procura p/ onde vai */
                        old=p;
                        p=p->next;
                }
                else {
                        if(p->prior) {
                                p->prior->next=l;
                                l->next=p;
                                l->prior=p->prior;
                                p->prior=l;
                                return top;
                        }
```

```
                    i->next=p;
                    /* primeiro elemento novo */
                    i->prior=0;
                    p->prior=i;
                    return i;
            }
    }
    old->next=i;        /*coloca no fim */
    i->next=0;
    i->prior=old;
    last=i;
    return start;
}
```

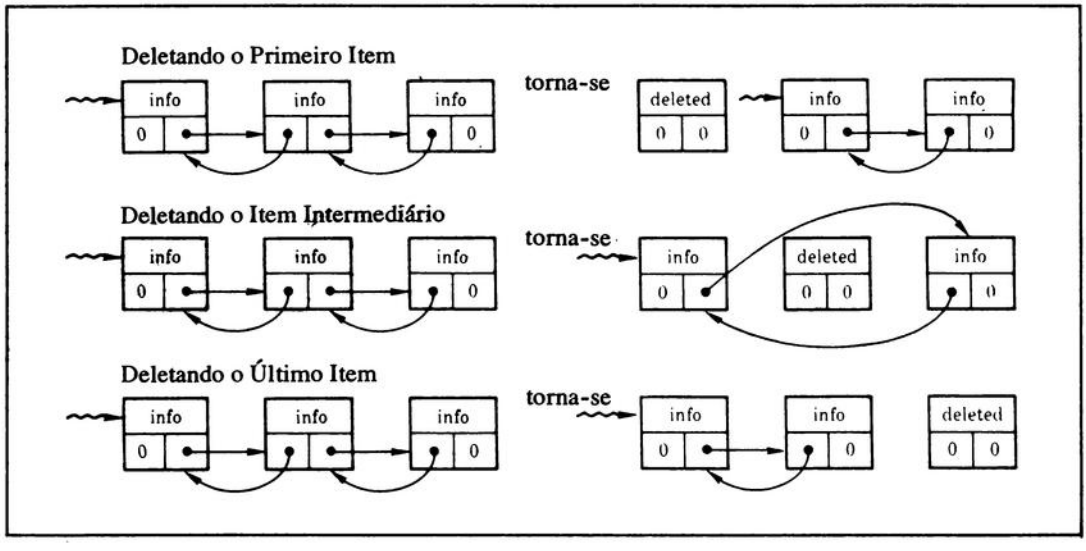


Figura 3.10. Deleção de três itens de uma lista duplamente encadeada.

Para que um item seja inserido no topo da lista, essa função precisa retornar um ponteiro para o primeiro item para que outras partes do programa saibam onde a lista começa. Como ocorre com a lista singularmente encadeada, para recuperar um item específico, o programa segue os elos até que o elemento procurado seja encontrado.

Existem três casos a serem considerados ao deletar um elemento de uma lista duplamente encadeada: deleção do primeiro item, deleção do item intermediário, e deleção do último item. A Figura 3.10 mostra como os elos são rearranjados.

A função seguinte deleta um item do tipo **address** de uma lista duplamente encadeada

```
struct address *dldelete(l,top)
struct address *l;      /* item a deletar */
struct address *top;    /* primeiro item da lista */
{
        if(l->prior)  l->prior->next = l->next;
        else {
                top=l->next;
                if(top) top->prior=0;
        }

        if(l->next) l->next->prior = l->prior;

        return top;
}
```

Essa função requer um ponteiro a menos do que a versão de lista singularmente encadeada, porque o item que está sendo deletado já incorpora nele um encadeamento até o elemento anterior e ao seguinte. Como o primeiro elemento na lista pode mudar, o ponteiro ao elemento do topo é devolvido à rotina chamadora.

UMA LISTA QUE USA LISTA DUPLAMENTE ENCADEADA

Eis um exemplo simples de um programa de lista postal que mostra o uso da lista duplamente encadeada. Toda a lista fica em memória enquanto é utilizada; no entanto, o programa pode salvar a lista postal em disco.

```
#include "stdio.h"
#include "malloc.h"

struct address {
        char name[30];
        char street[40];
        char city[20];
        char state[3];
        char zip[10];
        struct address *next;
        /* ponteiro para a proxima entrada */
        struct address *prior;
} list_entry;

struct address *start;
        /* ponteiro para a primeira entrada na lista */
struct address *last;
        /* ponteiro para a ultima entrada */
```

```
main()
{
        char s[80], choice;
        struct address *info;
        start=0;        /* tamanho da lista */
        for(;;) {
                switch(menu_select()) {
                        case 1: enter();
                                break;
                        case 2: delete();
                                break;
                        case 3: list();
                                break;
                        case 4: search();    /* procurar uma rua */
                                break;
                        case 5: save();
                                /* salvar uma lista num disco */
                                break;
                        case 6: load(); /* carregar do disco */
                                break;
                        case 7: exit(0);
                }
        }
}

menu_select()
{
        char s[80];
        int c;

        printf("1. Entrar com um nome\n");
        printf("2. Deletar um nome\n");
        printf("3. Listar um arquivo\n");
        printf("4. Procurar\n");
        printf("5. Salvar um arquivo\n");
        printf("6. Carregar um arquivo\n");
        printf("7. sair\n");
        do {
                printf("\nEntre com sua escolha: ")
                gets(s);
                c=atoi(s);
        } while(c<0 || c>7);
        return c;
}

enter()
{

        struct address *info,*dis_store();

        for(;;) {
```

```
                info=malloc(sizeof(list_entry));
                if(info==0) {
                        printf("\nestouro de memoria");
                        return;
                }
                inputs("entre com um nome: ",info->name,30);
                if(!info->name[0]) break;       /* parar entrada */
                inputs("entre com a rua: ",info->street,40);
                inputs("entre com a cidade: ",info->city,20);
                inputs("entre com o estado: ",info->state,3);
                inputs("entre com o cep: ",info->zip,10);

                start=dls_store(info,start);
        }       /* entrada do loop */
}
inputs(prompt,s,count)

char *prompt;
char *s;
int count;
{
        char p[256];

        do {
                printf(prompt);
                gets(p);
                if(strlen(p)>count) printf("\nmuito longo\n");
        } while(strlen(p)>count>count);
        strcpy(s,p);
}
struct address *dls_store(i,top)        /*armazena em ordem */
struct address *i;      /* novo elemento */
struct address *top;    /* primeiro elemento na lista */
{
        static struct address *last=0;  /* comeca com link nulo */
        struct address *old,*p;

       if(last==0) {   /*primeiro elemento na lista */
                i->next=0;
                i->prior=0;
                last=i;
                return i;
        }

        p=top;  /* comeca no topo da lista */

        old=0;
        while(p) {
                if(strcmp(p->name,i->name)<0) {
                        old=p;
                        p=p->next;
                }
```

```
                else {
                        if(p->prior) {
                                p->prior->next=i;
                                i->next=p;
                                p->prior=i;
                                return top;


                        }
                        i->next=p;      /* primeiro elemento novo */
                        i->prior=0;
                        p->prior=i;
                        return i;
                }
        }
        old->next=i;    /*coloca no fim */
        i->next=0;
        i->prior=old;
        last=i;
        return start;
}

delete()
{
        struct address *info, *find();
        char s[80];
        printf("entre com o nome: ");
        gets(s);
        info=find(s);
        if(info) {
                if(start==info) {
                        start=info->next;
                        if(start) start->prior=0;
                        else last=0;
                }
                else {
                        info->prior->next=info->next;
                        if(info!=last)
                                info->next->prior=info->prior;
                        else
                                last=info->prior;
                        }
                       free(info);
                        /* retorna memoria para sistema */

        }


truct address *find(name)
har *name;
```

```
        struct address *info;

        info=start;
        while(info) {
                if(!strcmp(name,info->name)) return info;
                info=info->next;        /* pega proximo endereco */
        }
        printf("nome nao foi encontrado\n");
        return 0;       /* nao encontrou */
}

list()
{
        register int t;
        struct address *info;

        info=start;
        while(info) {
                display(info);
                info=info->next;        /* pega proximo endereco */
        }
        printf("\n\n");

}

display(info)
struct address *info;
{
                printf("%s\n",info->name);
                printf("%s\n",info->street);
                printf("%s\n",info->city);
                printf("%s\n",info->state);
                printf("%s\n",info->zip);
                printf("\n\n");
}

search()
{
        char name[40];
        struct address *info,*find();

        printf("entre com o nome para procurar: ");
        gets(name);
        if(!(info=find(name))) printf("nao encontrou\n");
        else display(info);
}

save()
{
       register int t,size;
        struct address *info;
        char *p;
```

```
        FILE *fp;
        if((fp=fopen("mlist","w"))==0) {
                printf("arquivo nao aberto\n");
                exit(0);
        }
        printf("\nsalvando arquivo\n");
        size=sizeof(list_entry);
        info=start;
        while(info) {
                p=(char *) info;
                for(t=0;t<size;++t)
                        putc(*p++,fp);  /* salva byte por byte */
                info=info->next;        /* pega proximo endereco */
        }
        putc(EOF,fp);    /* envia EOF */
        fclose(fp);
}

load()
{
        register int t,size;
        struct address *info, *temp;
        char *p;
        FILE *fp;

        if((fp=fopen("mlist","r"))==0) {
                printf("arquivo nao aberto\n");
                exit(0);
        }

        printf("\ncarregando o arquivo\n");
        size=sizeof(list_entry);
        start=malloc(size);
        if(!start) {
                printf("estouro de memoria\n");
                return;
        }
        info=start;
        p=(char *) info;
        while((*p++=getc(fp))!=EOF) {
                for(t=0;t<size-1;++t)
                        *p++=getc(fp);
                info->next=malloc(size);
                if(!info->next) {

                        printf("estouro de memoria\n");
                        return;
                }
                info->prior=temp;
                temp=info;
                info=info->next;
```

```
        p=(char *) info;
    }
    temp->next=0;    /* ultima entrada */
    last=temp;

    start->prior=0;
    fclose(fp);
}
```

ÁRVORES BINÁRIAS

A quarta estrutura de dados é a *árvore binária*. Embora possa haver muitos tipos de árvores, árvores binárias são especiais porque, quando são ordenadas, elas se aplicam a buscas velozes, inserções e deleções. Cada item numa árvore binária consiste em informação com um elo no ramo esquerdo e um no ramo direito. A Figura 3.11 mostra uma pequena árvore.

A terminologia especial necessária para discutir árvores é um caso clássico de metáforas misturadas. A *raiz* é o primeiro item na árvore. Cada item (dado) é chamado de *nó* (ou algumas vezes *folha*) da árvore e qualquer parte da árvore de *subárvore*. Um nó sem subárvores ligadas a ele é chamado de *nó terminal*. A *altura* da árvore é igual ao número de camadas (de profundidade) que as suas raízes atingem. Durante toda essa discussão imagine as árvores binárias aparecendo na memória como elas aparecem no papel, mas lembre que a árvore é apenas uma maneira de estruturas de dados na memória, que tem formato linear.

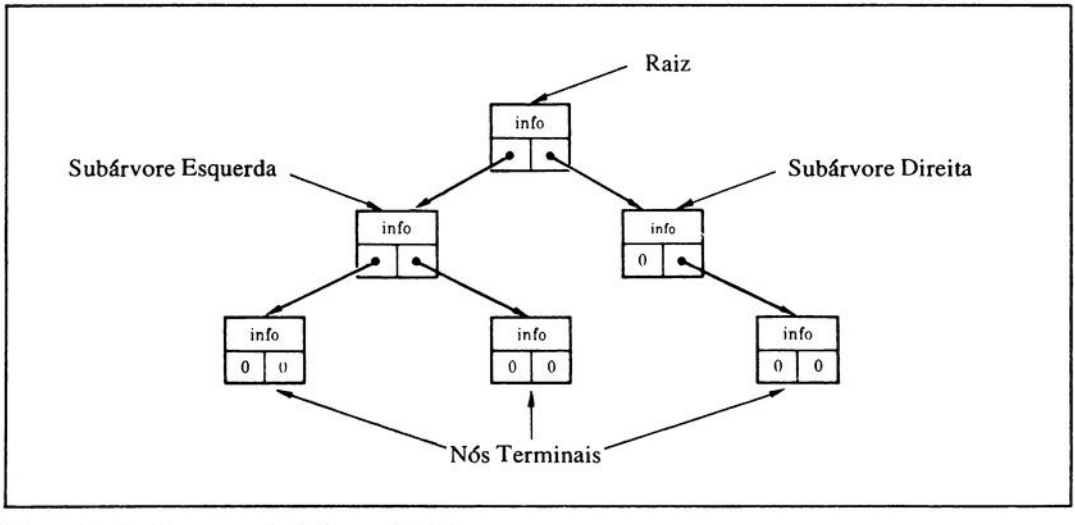
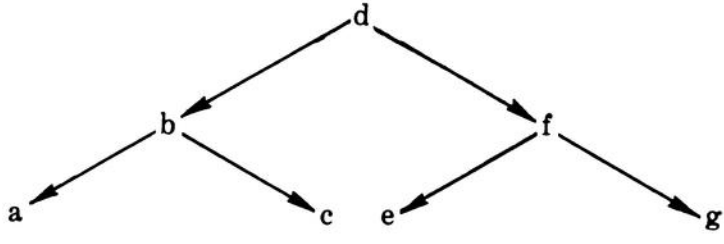


Figura 3.11. Um exemplo de árvore binária.

A árvore binária é uma forma especial de lista encadeada. Pode-se inserir, deletar e acessar itens em qualquer ordem. Além disto, a operação de recuperação é não-destrutiva. Embora fáceis de visualizar, as árvores apresentam difíceis problemas de programação que essa seção apenas introduzirá.

A maioria das funções que usam árvores é recursiva porque a própria árvore é uma estrutura de dados recursivos; isto é, cada subárvore é uma árvore. Assim, as rotinas que são aqui desenvolvidas também são recursivas. Versões não recursivas dessas funções existem, mas o seu código é muito mais difícil de ser compreendido.

A ordenação de uma árvore depende de como ela será referenciada. O procedimento de acesso a cada nó em uma árvore é chamado *árvore transversal* e pode ser visto neste diagrama:



Existem três maneiras de *transversalizar uma árvore: ordenada, pré-ordenada e pós-ordenada*. Usando a maneira ordenada você visita a subárvore da esquerda, a raiz e em seguida a subárvore da direita. Na maneira pré-ordenada você visita a raiz, depois a subárvore da esquerda e depois a subárvore da direita. Com a pós-ordenada, você visita a subárvore da esquerda, depois a subárvore da direita, e depois a raiz.

A ordem de acesso para a árvore acima, utilizando cada um dos métodos, é a seguinte:

ordenada	a b c d e f g
pré-ordenada	d b a c f e g
pós-ordenada	a c b e g f d

Embora uma árvore não precise ser ordenada, a maioria dos usos assim o requer. A elaboração de uma árvore ordenada depende de como você pode *transversalizar*. Os exemplos restantes, neste capítulo, acessam a árvore de maneira ordenada. Numa árvore binária ordenada, a subárvore da esquerda contém nós que são menores ou iguais à raiz, enquanto os da direita são maiores que a raiz. A função seguinte **stree( )** cria uma árvore binária ordenada.

```
struct tree  {
        char info;
        struct tree *left;
        struct tree *right;
} t;
struct tree *stree(root,r,info)
struct tree *root;
struct tree *r;
char info;
{

        if(r==0)  {     /* primeiro no da sub-arvore */
                r=malloc(sizeof(t));
                if(r==0)  {
                        printf("estouro de memoria\n");
                        exit(0);
                }
                r->left=0;
                r->right=0;
                r->info=info;
                if(root)  {
                        if(info<root->info)  root->left=r;
                        else  root->right=r
                }
                else  { /* primeiro no da arvore */
                        r->right=0;
                        r->left=0;
                }
                return r;
        }

        if(info<=r->info)  stree(r,r->left,info);
        if(info>r->info)  stree(r,r->right,info);
}
```

Esse algoritmo simplesmente segue os elos através da árvore, indo para a esquerda ou para a direita baseado no campo **info.** Para usar essa função, você necessita de uma variável global que contenha a raiz da árvore. Essa global deve ser inicializada com 0, e um ponteiro para raiz será designado na primeira chamada a **stree( ).**

Chamadas sucessivas não necessitam de redesignar a raiz. Se você assume para esta variável global o nome **rt,** então para chamar a função **stree( )** você usaria

```
/* call stree() */
if(!rt) rt=stree(rt,rt,info);
else stree(rt,rt,info);
```

desta forma, o primeiro elemento e seu sucessivo podem ser incluídos corretamente.

A função **stree( )** é um algoritmo recursivo, como a maioria das rotinas de árvore. A mesma rotina seria muito mais longa se métodos interativos clássicos fossem usados. A função deve ser chamada com um ponteiro na raiz, nó esquerdo ou direito e informação. Embora um único caractere seja usado aqui como a informação, para simplificar, você pode substituir por qualquer dado simples ou complexo que queira.

Para *transversalizar* a árvore composta pelo método ordenado **stree( )** e imprimir o campo **info** de cada nó, você pode usar a função **inorder.**

```
inorder(root)
struct tree *root;
{
        if(!root) return;

        inorder(root->left);
        printf("%c ",root->info);
        inorder(root->right);
}
```

Essa função recursiva retorna quando alcança o nó terminal (um ponteiro nulo). As funções que percorrem a árvore pré-ordenada e pós-ordenada são aqui mostradas.

```
preorder(root)
struct tree *root;
{
        if(!root) return;

        printf("%c ",root->info);
        preorder(root->left);
        preorder(root->right);
}

postorder(root)
struct tree *root;
{
        if(!root) return;

        postorder(root->left);
        postorder(root->right);
        printf("%c ",root->info);
}
```

É possível escrever um pequeno programa que crie uma árvore binária ordenada e que imprima a árvore lateralmente na tela do seu computador. Você necessita apenas de uma pequena modificação na função **inorder( )**. A nova função, que é chamada **print-tree( )**, imprime uma árvore de modo ordenado.

```
print_tree(r,l)
struct tree *r;
int l;
{
        int i;

        if(r==0) return;

        print_tree(r->left,l+1);
        for(i=0;i<l;++i) printf("    ");
        printf("%c\n",r->info);
        print_tree(r->right,l+1);
}
```

Ao testar o programa de impressão de árvores exemplificado aqui, procure utilizar várias árvores para ver como cada uma é criada.

```
struct tree {
        char info;
        struct tree *left;
        struct tree *right;
}t;

struct tree *root;      /* primeiro no na arvore */

main()  /* programa imprime arvore */
```

```
{
        char s[80];
        struct tree *stree();

        root=0; /* inicializa */

        do {
                printf("entre com uma letra: ");
                gets(s);
                if(!root)  root=stree(root,root,*s);
                else stree(root,root,*s);
        } while(*s);

        print_tree(root,0);

}


struct tree *stree(root,r,info)
struct tree *root;
struct tree *r;
char info;
{

        if(r==0) {
                r=malloc(sizeof(t));
                if(r==0) {
                        printf("estouro de memoria\n");
                        exit(0);
                }
                r->left=0;
                r->right=0;
                r->info=info;
                if(info<root->info) root->left=r;
                else root->right=r;
                return r;
        }

        if(info<r->info) stree(r,r->left,info);
        else
        if(info>r->info) stree(r,r->right,info);
}

print_tree(r,l)
struct tree *r;
int l;
{
        int i;

        if(r==0) return;

        print_tree(r->left,l+1);
```

```
                    p->left=root->left;
                    free(root);
                    return p2;
                }
            }
            if(root->info<key)  root->right=dtree(root->right,key);
            else root->left=dtree(root->left,key);
            return root;
}
```

Lembre-se de atualizar o ponteiro à raiz no resto do seu programa porque o nó deletado pode ser a raiz da árvore.

Quando usado em programas de gerenciamento de banco de dados, árvores binárias oferecem poder, flexibilidade e eficiência porque a informação para esses bancos de dados deve residir em disco e porque tempos de acesso são importantes. Porque uma árvore binária balanceada tem, como pior caso, $\log_2 n$ comparações em um processo de busca, ela comporta-se muito melhor que uma lista encadeada, a qual depende de uma busca seqüencial.

Capítulo **4**

ALOCAÇÃO DINÂMICA

Projetar um programa de computador pode ser comparado com projetar uma edificação, com numerosas considerações funcionais e estéticas que contribuem para o resultado final. Por exemplo, alguns programas são funcionalmente rígidos como uma casa, com um certo número de dormitórios, uma cozinha, dois banheiros etc. Outros programas devem ser flexíveis como centros de convenções com paredes divisórias e chão modular que possam ser adaptados a várias necessidades. Este capítulo apresenta mecanismos de armazenamento que lhe permitem escrever programas flexíveis.

Existem duas maneiras de um programa em C armazenar informação na memória principal do computador. A primeira usa variáveis globais e locais – incluindo vetores e estruturas – que são definidas pela linguagem C. Para variáveis globais, o armazenamento é fixo durante todo o tempo de processamento do programa. Para variáveis locais, o armazenamento é alocado no espaço de pilha do computador. Embora variáveis globais e locais sejam eficientemente implementadas em C, elas exigem que o programador saiba de antemão qual o espaço de armazenamento necessário para cada situação.

A segunda e mais eficiente maneira de armazenar informação é usando as funções **malloc( )** e **free( )** de alocação dinâmica do C. O espaço de armazenamento de informação é alocado da área livre de memória que se encontra entre a área permanente de armazenamento do seu programa e a pilha (a qual é usada pelo C para armazenar variáveis locais). A Figura 4.1 mostra como um programa em C apareceria na memória. A pilha *cresce* de cima para baixo na medida em que é usada; sendo assim, a quantidade de memória necessária é determinada pela maneira como seu programa foi projetado. Por exemplo, um programa com muitas funções recursivas utiliza mais a memória de pilha do que um programa sem funções recursivas, porque variáveis locais são armazenadas na pilha. A memória necessária para o programa e os dados globais é determinada durante a execução

do programa. A memória necessária para **malloc( )** é retirada da área livre de memória, começando acima das variáveis globais e crescendo no sentido da pilha. Em casos extremos, é possível que a pilha penetre na memória alocada.

Se você não está inteiramente familiarizado com **malloc( )** e **free( )**, aqui está uma breve revisão.

UMA REVISÃO DE MALLOC( ) E FREE( )

As funções **malloc( )** e **free( )** formam o sistema de alocação dinâmica do C e são parte da biblioteca-padrão C. Elas trabalham juntas, usando a região de memória livre que se encontra entre o fim do seu programa e o topo da pilha, a fim de estabelecer e manter uma região de armazenamento disponível. Cada vez que uma requisição memória **malloc( )** é feita, uma porção da memória livre restante é alocada. Cada vez que uma chamada a **free( )** é feita é restaurada uma porção da memória que retorna ao sistema. A maneira mais usual de implementar **malloc( )** e **free( )** é organizar a memória livre em uma lista encadeada.

A função **malloc( )** é a função de alocação de memória de caráter geral de C. Sua forma geral de chamada é

```
char *malloc( ), *p;
int number_of_bytes;
p=malloc(number_of_bytes);
```

Após chamada satisfatória, **p** contém um ponteiro para o primeiro byte da região de memória. Se não houver memória suficiente para satisfazer a requisição **malloc( )**, ocorre uma falha de alocação e **malloc( )** retorna o valor zero. A função **malloc( )** precisa sempre saber o número de bytes a serem alocados – mesmo se a informação que você armazenar for outro tipo de dado, tal como um caractere ou uma estrutura. Você pode usar **sizeof** para determinar o número necessário de bytes para cada tipo de dado. Isto ajuda a tornar os seus programas portáteis para muitos sistemas. Mesmo que um ponteiro caractere seja devolvido, ele pode ser designado a um ponteiro do tipo próprio para satisfazer qualquer necessidade de programação. Antes de usar o ponteiro devolvido por **malloc( )** certifique-se de que seu pedido de alocação teve sucesso, testando o valor de retorno contrário a zero. Não use um ponteiro de valor zero – provavelmente ele danificará o seu sistema.

A função **free( )** é o oposto da **malloc( )**: ele devolve memória anteriormente alocada ao sistema. Aquela parte de memória pode ser reutilizada por uma chamada subseqüente ao **malloc( )**. A forma geral do **free( )** é

```
char *p;
/* assume que p contém um ponteiro endereço válido */
free(p);
```

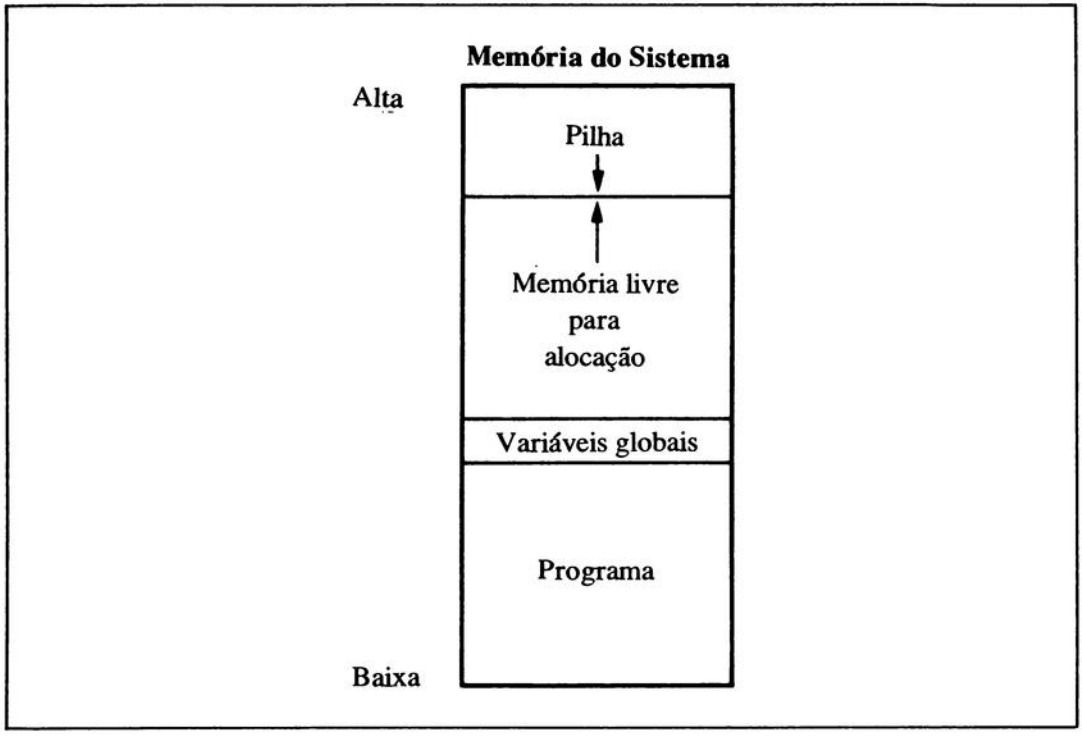


Figura 4.1. Uso de memória de programa em C.

Lembre-se de nunca chamar **free( )** com um argumento inválido, porque a lista livre seria destruída.

O programa seguinte aloca armazenamento suficiente para 40 caracteres, imprime seus valores e libera a memória de volta para o sistema. O **sizeof** é usado aqui para assegurar a portabilidade com outros tipos de computadores.

```
main()  /* exemplo de uma alocacao pequena */
{
        int *p,t;

        p=(int *) malloc(40*sizeof(int));
        if(p==0) {
```

```
            printf("estouro de memoria\n");
            exit(0);
        }
        for(t=0;t<40;++t) *(p+t)=t;

        for(t=0;t<40;++t) printf("%d ",*(p+t));

        free(p);
}
```

PROCESSAMENTO DO VETOR ESPARSO

Um dos principais usos da alocação dinâmica é para processar um *vetor esparso*. Em um vetor esparso, nem todos os elementos estão realmente presentes. Você pode querer criar um vetor como este quando as dimensões do vetor necessário forem maiores do que a memória de sua máquina e quando nem todas as posições de memória forem usadas. Vetores multidimensionais podem consumir uma grande quantidade de memória porque suas necessidades de armazenamento são exponencialmente proporcionais ao seu tamanho. Por exemplo, um vetor de caracteres de 10 x 10 precisa de somente 100 bytes de memória, e um de 100 x 100 precisa de 10.000, e um vetor de caracteres de 1000 x 1000 precisa de 1.000.000 de bytes de memória.

Um programa de "planilha" é um bom exemplo de um vetor esparso. Mesmo para uma matriz grande – digo, 999 por 999 – só uma parte deste total deverá ser realmente utilizada a cada momento. Planilhas utilizam uma matriz para armazenar fórmulas, valores, e série de caracteres associados a cada posição de memória. Em um vetor esparso, o armazenamento de cada elemento é alocado em um espaço de memória livre, quando necessário. Embora apenas uma pequena porção dos elementos esteja realmente sendo usada, o vetor pode parecer muito grande – maior do que normalmente caberia na memória do computador.

Nessa seção três técnicas distintas serão usadas para criar um vetor esparso: uma lista encadeada, uma árvore binária e um vetor ponteiro. Todos estes exemplos assumem que a matriz da planilha esteja organizada como o exemplo da Figura 4.2, com **X** localizado na célula B2.

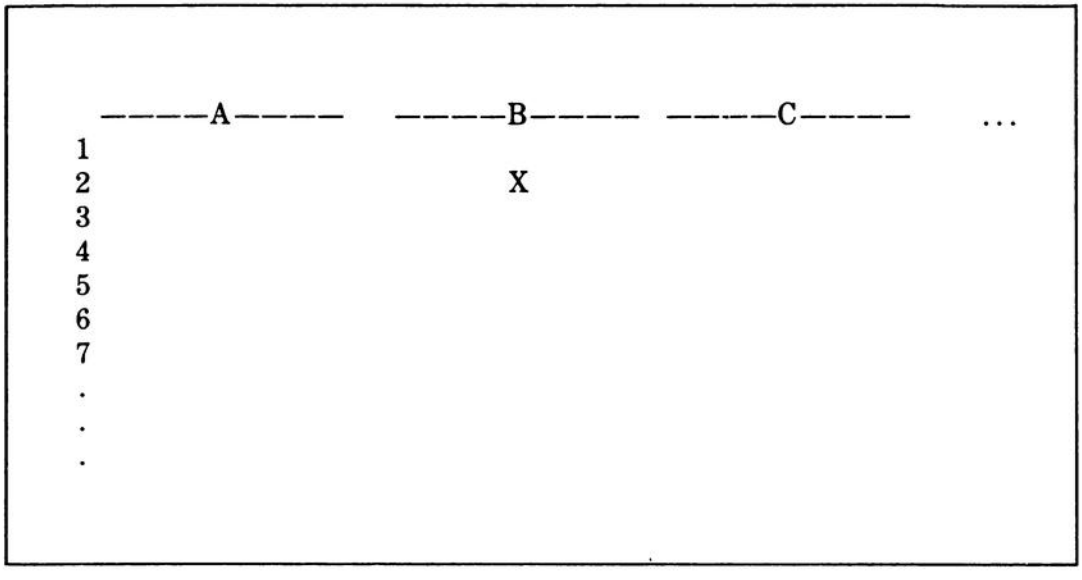


Figura 4.2. A organização de uma planilha simples.

O MÉTODO DA LISTA ENCADEADA PARA VETOR ESPARSO

Quando você implementar um vetor esparso que usa uma lista encadeada, uma estrutura será usada para guardar a informação de cada elemento no vetor, incluindo sua posição lógica nesse vetor e elos para o elemento precedente e o elemento seguinte. Cada estrutura é colocada na lista com os elementos em uma ordem baseada no vetor de indexação. O vetor é acessado através dos enlaces ou elos.

Por exemplo, você poderia usar a seguinte estrutura para criar um vetor esparso para uso num programa de planilha:

```
struct cell  {
        char cell_name[9];    /* celula do nome p.e. A1, B34 /
        char formula[128];    /* p.e. 10/B2 */
        struct cell *next;    /* ponteiro p/ proxima entrada */
        struct cell *prior;   /* ponteiro p/ registro anterior */
}  list_entry;
```

O campo **cell–name** possui uma seqüência de caracteres que contém o nome da célula (tal como A1, B34 ou Z19). A seqüência de caracteres **fórmula** possui a fórmula designada a cada local da planilha.

Aqui estão algumas amostras de funções que seriam usadas pela planilha, que usa um vetor esparso de lista encadeada. Lembre-se de que existem várias maneiras de implementar um programa planilha, e a estrutura de dados e rotinas usadas nesses exemplos devem servir apenas como exemplos de técnicas de vetor esparso. As variáveis globais serão usadas para apontar o começo e o fim da lista de vetor encadeada:

```
struct cell *start; /* primeiro elemento da lista */
struct cell *last; /* último elemento da lista */
```

Quando você carrega uma fórmula numa célula de uma planilha típica, na verdade, está criando um novo elemento dentro do vetor esparso. Se a planilha usa uma lista encadeada, aquela nova célula é inserida usando-se **dls-store( )**, que foi desenvolvido no Capítulo 3. (Graças à habilidade da linguagem C em criar e deixar independentes funções recusáveis, você pode aplicá-la aqui, sem nenhuma mudança fundamental.) Lembre-se: a lista é ordenada através do nome da célula – A12 precede A13, e assim por diante.

```
struct cell *dls_store(i)       /* armazena em ordem */
struct cell *i;
{
        static struct cell *last=0;   /* inicia com link nulo */
        struct cell *old,*p;

        if(last==0)  {  /* primeiro elemento da lista */
                i->next=0;
                i->prior=0;
                last=i;
                return i;
        }

        p=start;        /* comeca no topo da lista */

        old=0;
        while(p)  {
                if(strcmp(p->cell_name,i->cell_name)<0) {
                        old=p;
                        p=p->next;
                }
                else {
                        if(p->prior)  {
                                p->prior->next=i;
                                i->next=p;
                                i->prior=p->prior;
                                p->prior=i;
                                return start;
```

```
                        }
                        i->next=p;
                        i->prior=0;
                        p->prior=i;
                        return i;
                }
        }
        old->next=i;    /* coloca no final */
        i->next=0;
        i->prior=old;
        last=i;
        return start;
}
```

Para retirar uma célula da planilha, você deve remover a estrutura correta da lista e permitir que a porção de memória ocupada por esta estrutura volte ao sistema através da utilização de **free( )**. A função **delete( )**, aqui apresentada, retira uma célula da lista quando é fornecido o nome da célula:

```
delete(cell_name)
char *cell_name;
{
        struct cell *info, *find();

        info=find(cell_name);
        if(info)  {
                if(start==info)  {
                        start=info->next;
                        if(start) start->prior=0;
                        else last=0;
                }
                else  {
                        info->prior->next=info->next;
                        if(info!=last)
                                info->next->prior=info->prior;
                        else
                                last=info->prior;
                }
                free(info);     /* retorna memoria p/ sistema */
        }
}
```

A função **find( )** localiza qualquer célula específica. É uma função importante porque muitas fórmulas de planilhas fazem referência a outras células que necessitam ser encontradas para que seus valores possam ser atualizados. A função **find( )** requer o uso de uma busca linear para localizar cada item, e, como foi visto no Capítulo 2, a média de

comparações em uma busca linear é *n*/2, onde *n* é o número de elementos na lista. Além disso uma perda significativa na performance ocorre porque cada célula pode conter referências a outras células dentro da fórmula e cada uma dessas células precisam ser encontradas. Eis aqui um exemplo de **find()**.

```
struct cell *find(cell)
char *cell;
{
        struct cell *info;

        info=start;
        while(info) {
                if(!strcmp(cell,info->cell)) return info;
                info=info->next;    /* recolhe proxima celula */
        }
        printf("celula nao encontrada\n");
        return 0;       /* nao encontrou */
}
```

O método da lista encadeada para criar, manter e processar um vetor esparso tem uma grande desvantagem – ele precisa usar uma busca linear para acessar cada célula na lista. Sem usar informação adicional, o que requer mais memória disponível, você não pode usar uma busca binária para localizar uma célula. Até a rotina de armazenamento usa uma busca linear para encontrar a localização correta para inserir uma nova célula na lista. Você pode solucionar esses problemas usando uma árvore binária para operar o vetor esparso.

O MÉTODO DA ÁRVORE BINÁRIA PARA VETOR ESPARSO

Na realidade, a árvore binária nada mais é do que uma lista duplamente encadeada modificada. Sua maior vantagem sobre uma lista é que pode ser varrida rapidamente, o que significa que inserções e consultas podem ser rápidas. Em aplicações onde você deseja uma estrutura de lista encadeada mas necessita buscas rápidas, a árvore binária é a solução.

Para usar uma árvore binária para operar o exemplo da planilha, você deve mudar a **célula** estrutural como mostrado a seguir.

```
struct cell  {
        char cell_name[9];    /* celula do nome p.e. A1, B34 /
         char formula[128];    /* p.e. 10/B2 */
         struct cell *left;    /* ponteiro p/ proxima entrada */
         struct cell *right;   /* ponteiro p/ registro anterior */
}  list_entry;
```

Você pode modificar a **função stree( )** do Capítulo 3 para que ela construa uma árvore baseada no nome da célula. Perceba que ela assume que o parâmetro **new** é um ponteiro para uma nova entrada da árvore.

```
struct cell *stree(root,r,new)
struct cell *root;
struct cell *r;
struct cell *new;
{
     if(r==0) {       /* primeiro no da sub-arvore */
              new->left=0;
              new->right=0;
              if(root) {
                       if(strcmp(new->cell_name,root->cell_name)<0)
                                root->left=new;
                       else root->right=new;
              }
              else {   /* primeiro no da arvore */
                       new->right=0;
                       new->left=0;
              }
              return new;
     }

     if(strcmp(new->cell_name,r->cell_name)<=0)
              stree(r,r->left,new);
     if(strcmp(new->cell_name,r->cell_name)>0)
              stree(r,r->right,new);
     return root;
}
```

Stree( ) deve ser chamada com um ponteiro ao nó da raiz para os dois primeiros parâmetros, e um ponteiro para a nova célula como terceiro parâmetro. Ela devolve um ponteiro para a raiz.

Para deletar uma célula de uma planilha, você poderia modificar **dtree( )**, como visto aqui, para aceitar o nome da célula como uma chave:

```
struct cell *dtree(root,key)
struct cell *root;
char *key;
{
        struct tree *p,*p2;

         if(!strcmp(root->cell_name==key)Ê      /* deleta raiz */
                 /* arvore vazia */
                 if(root->left==root->right) {
                         free(root);
                         return 0;
                 }
                 /* ou se uma sub-arvore e' nula */
                 else if(root->left==0) {
                         p=root->right;
                         free(root);
                         return p;
                 }
                 else if(root->right==0) {
                         p=root->left;
                         free(root);
                         return p;
                 }
                 /* ou ambas arvores presentes */
                 else {
                         p2=root->right;
                         p=root->right;
                         while(p->left) p=p->left;
                         p->left=root->left;
                         free(root);
                         return p2;
                 }
         }
         if(strcmp(root->cell_name,key)<0)
                 root->right=dtree(root->right,key);
         else root->left=dtree(root->left,key);
         return root;
}
```

Finalmente, você pode usar a função **search( )** modificada para localizar qualquer célula na planilha rapidamente, dando o nome de sua célula.

```
struct cell *search_tree(root,key)
struct cell *root;
char *key;
{
        if(!root) return root;  /* arvore vazia */
        while(strcmp(root->cell,key)) {
```

```
                if(strcmp(root->cell_name,key)<0)
                        root=root->left;
                else root=root->right;
                if(root==0) break;
        }
        return root;
}
```

O aspecto mais importante em usar uma árvore binária em vez de uma lista encadeada é que ela resulta em tempos de busca muito mais rápidos. (Lembre-se: uma busca seqüencial requer $n/2$ comparações em média, onde n é o número de elementos em uma lista, ao passo que uma árvore binária requer somente $\log_2 n$ comparações.) No entanto, em algumas situações existem alternativas ainda melhores.

O MÉTODO DE VETOR PONTEIRO PARA VETOR ESPARSO

Suponha que as dimensões de sua planilha fossem de 26 por 100 (A1 até Z100), ou um total de 2600 elementos. Na teoria, então, o seguinte vetor de estruturas poderia ser usado para armazenar as entradas da planilha:

```
struct cell {
        char cell_name[9];
        char formula[128];
} list_entry[2600];     /* 2600 celulas */
```

Porém, 2600 células multiplicadas por 128 (levando em conta apenas a fórmula) requerem 332.800 bytes de memória para uma planilha razoavelmente pequena. Este método obviamente não é prático. Em vez disto, você pode criar um vetor de ponteiros para estruturas. Este método requer significativamente menos armazenamento permanente do que a criação de uma matriz completa e ofereceria performance superior aos métodos de lista encadeada e árvore binária. A declaração seria como está aqui apresentada.

```
struct cell {
        char cell_name[9];
        char formula[128];
} list_entry;

struct cell *sheet[10000];      /* matriz de 10000 ponteiros */
```

Você pode utilizar este vetor menor para armazenar ponteiros para a informação que é inserida pelo usuário da planilha. Conforme cada item é inserido, um ponteiro para a informação contida na célula é armazenado em local próprio dentro do vetor. A Figura 4.3 mostra como esse processo ocorreria na memória, com o vetor de ponteiros fornecendo suporte ao vetor esparso.

Antes de usar o vetor de ponteiros, você deve inicializar cada elemento com nulos, o que indica que não existe informação nessa posição. Use a seguinte função:

```
init_sheet()
{
     register  int  t;
     for(t=0; t<10000; ++t)  sheet[t]=0;
}
```

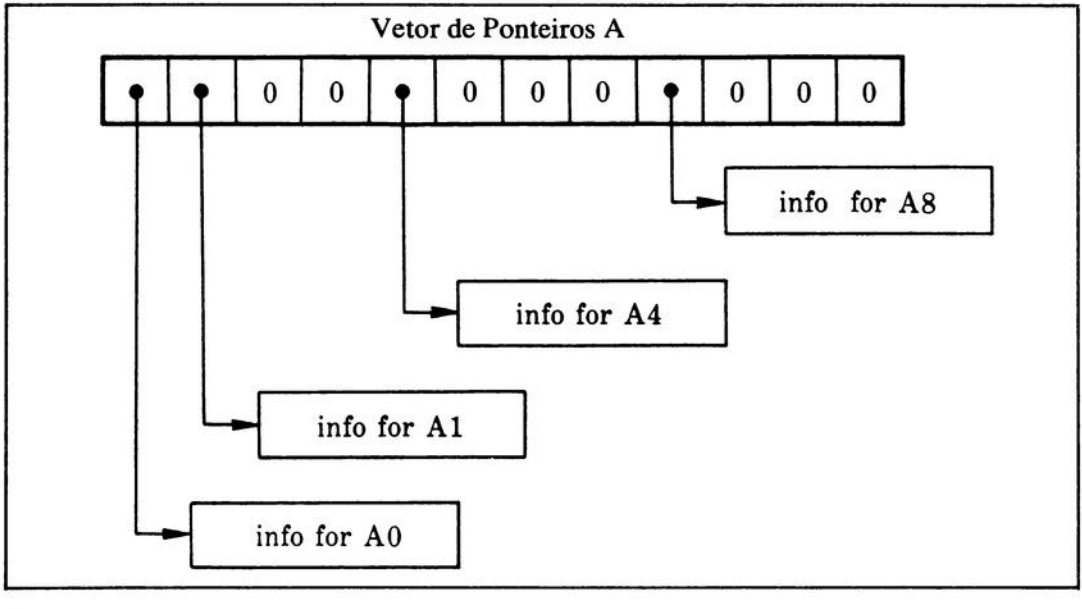


Figura 4.3. Um vetor de ponteiros como suporte a um vetor esparso.

Quando o usuário insere uma fórmula para uma célula na planilha, a localização da célula, definida pelo seu nome, produz um índice para **sheet** do vetor ponteiro. O índice é derivado do nome da célula através da conversão do nome em um número, como mostrado aqui na função **store( )**. Quando **store( )** computa o índice, ela admite que todos os nomes de célula começam com letras maiúsculas seguidos de um inteiro – por exemplo, B34 ou C19.

```
store(i)
struct cell *i;
{
        int loc;
        char *p;

        loc=(*(i->cell_name)-'A');
        p=&(i->cell_name[1]);
        loc+=atoi(p)*26;        /* 26 colunas * numero de linhas */

        if(loc)=10000) {
                printf("celulas fora do limite\n");
                return;
        }

        sheet[loc]=i;   /* coloca ponteiro na matriz */
}
```

Como cada nome de célula é único, cada índice também é único; como a seqüência de intercalação do ASCII é usada, o ponteiro para cada entrada é armazenado no elemento próprio do vetor. Se você comparar esse processo ao da versão de lista encadeada, verá o quanto este é muito mais curto e simples.

A função **delete()** também fica reduzida. Quando chamada pelo índice da célula, **delete()** zera o ponteiro para o elemento e retorna a memória ao sistema:

```
delete(cell_index)
int cell_index;
{
        free(sheet[cell_index]); /* retorna memoria p/ sistema */
        sheet[cell_index]=0;
}
```

Novamente, quando você comparar isto com a versão da lista encadeada, notará que este código é muito mais simples e rápido.

Através deste processo, localizar uma dada célula torna-se trivial pois o nome em si fornece o vetor de indexação; assim, a função **find( )** modifica-se como mostrado aqui.

```
struct cell *find(cell_name)
char *cell_name;
{
        int loc;
        char *p;
```

```
    loc=(*(i->cell_name)-'A');
    p=&(i->cell_name[1]);
    loc+=atoi(p)*26;        /* 26 colunas * numero de linhas */

    if(loc>=10000 || !sheet[loc]) { /* nao entra nesta celula */
            printf("celula nao encontrada\n");
            return -1;       /* nao encontrou */
    }
    else retuen sheet[loc];
}
```

Lembre-se de que um vetor de ponteiros por si só usa alguma memória para cada locação – estando ou não a posição de memória sendo utilizada. Isto pode se tornar uma séria limitação para alguns tipos de aplicação.

COMPARANDO UMA LISTA ENCADEADA, UMA ÁRVORE BINÁRIA, E OS MÉTODOS DE VETOR DE PONTEIROS

Ao decidir usar um método de lista encadeada, uma árvore binária, ou um vetor de ponteiros para implementar um vetor esparso, você deve considerar a eficiência e velocidade da memória.

Quando o vetor está muito espalhado, os métodos de memória eficientes são as implementações de lista encadeada e árvore binária, porque somente aqueles elementos que estão realmente em uso têm memória alocada para eles. Os enlaces requerem uma pequena memória adicional e geralmente têm efeito desprezível. A idéia requer que cada elemento do vetor tenha um ponteiro alocado para ele, exista ou não informação contida nele. Não somente a memória deve ser suficiente para todo o vetor de ponteiros, mas também precisa possuir memória restante o bastante para o uso da aplicação. Isso poderia ser um sério problema para certas aplicações, ao passo que não é problema para outras. Você pode decidir isso calculando a porção aproximada de memória livre e determinar se é suficiente para o seu programa.

A situação muda, no entanto, quando o vetor está completamente cheio. Neste caso, o método do vetor de ponteiros faz melhor uso da memória porque tanto a implementação da lista encadeada como da árvore binária precisam de dois ponteiros, ao passo que o vetor de ponteiros precisa só de um. Por exemplo, se um vetor de 1000 elementos estiver cheio, cada ponteiro terá extensão de 2 bytes, então tanto a árvore binária como a lista encadeada utilizariam 4000 bytes para os ponteiros – mas o vetor de ponteiros só precisa de 2000, fazendo uma economia de 2000 bytes.

O método de execução mais rápido é o vetor de ponteiros. Como no exemplo da planilha, existe um método fácil que indexa o vetor de ponteiros e o encadeia com os elementos do vetor esparso. Esse método faz acesso aos elementos do vetor esparso praticamente tão rápido quanto o acesso a um vetor normal. A versão de lista encadeada é muito lenta para comparações, porque precisa usar uma busca linear para localizar cada elemento. Mesmo se informação extra for acrescentada para permitir um acesso mais rápido aos elementos de uma lista encadeada, ainda assim seria mais lento do que a capacidade de acesso direto do vetor de ponteiros. A árvore binária certamente acelera o tempo de busca, mas quando comparada com a capacidade de acesso indexado direto do vetor de ponteiros, ainda parece moroso.

Sempre que possível, a implementação de um vetor de ponteiros é a melhor opção porque é muito mais rápido. Porém, se a utilização da memória é crítica, você não tem escolha e terá de usar a lista encadeada ou a árvore binária.

BUFFERS REUTILIZÁVEIS

Quando a memória é escassa, pode-se usar alocação dinâmica em lugar de variáveis normais. Como um exemplo, imagine dois processos **A( )** e **B( )** dentro de um programa. Assuma que **A( )** requer 60% de memória livre e **B( )** precisa de 55%. Se as necessidades de armazenamento tanto de **A( )** como de **B( )** derivam de variáveis locais, então **A( )** não pode chamar **B( )** e **B( )** não pode chamar **A( )** – seria necessário mais de 100% da memória. Se **A( )** nunca chamar **B( )**, então não haverá problema, a menos que você queira que **A( )** chame **B( )**. A única maneira disto ocorrer é usar armazenamento dinâmico para ambos e liberar essa memória antes de chamar a outra. Em outras palavras, se tanto **A( )** como **B( )** necessitarem de mais do que metade da memória disponível enquanto estiver sendo executado, e se **A( )** precisar chamar **B( )**, então deverão usar alocação dinâmica. Desta maneira, tanto **A( )** como **B( )** terão a memória necessária quando precisarem dela.

Imagine que existam 100.000 bytes de memória disponíveis em um computador que está "rodando" um programa com as funções a seguir:

```
A( )
{
        char a[60000];
        .
        .
        .
        B( );
        .
        .
        .
}

B( )

{
        char b[55000];
        .
        .
        .
}
```

Aqui, tanto **A( )** como **B( )** têm variáveis locais que requerem mais da metade da memória livre. Não existe forma de **B( )** poder ser executado, porque não existe memória disponível o bastante para alocar os 55.000 bytes necessários para o vetor local **b.**

Uma situação como esta é algumas vezes insuperável, mas sob certas circunstâncias você pode trabalhar em torno disto. Se A( ) não precisa conservar o conteúdo do vetor **a** enquanto **B( )** está sendo executado, então tanto **A( )** como **B( )** podem partilhar a memória. Você pode fazer isto, alocando dinamicamente os vetores **A( )** e **B( ).** Então **A( )** poderia liberar memória antes de chamar **B( )** e realocar esta memória mais tarde se necessário. A codificação seria vista assim:

```
A( )
{
 char *s;
        s=malloc(60000);
        .
        .
        .
        free(s); /* memória liberada para B( ) */
```

```
        B( );
        s=malloc(60000);
        .
        .
        .
        free(s); /*tudo feito*/
}
B( )
{
        char *b;
        b=malloc(55000);
        .
        .
        .
        free(55000);
}
```

Só o ponteiro s existe enquanto **B( )** é executado.

Embora você só vá precisar fazer algo parecido ocasionalmente, é útil conhecer a técnica porque é a única maneira encontrada para solucionar este tipo de problema.

O DILEMA DA "MEMÓRIA DESCONHECIDA"

Se você é um programador profissional, provavelmente já teve de enfrentar o dilema da "memória desconhecida". Isto ocorre quando você escreve um programa que tem parte de sua performance baseada na quantidade de memória disponível dentro do computador que executa este programa. Exemplos de programas que podem apresentar este problema são planilhas, programas de lista postal em RAM e ordenações. Por exemplo, uma ordenação executada na memória que pode gerenciar 10.000 endereços em uma máquina de 256k é capaz de ordenar somente 5000 endereços em uma máquina de 128k. Se este programa fosse usado em computadores com tamanho de memória desconhecida, seria difícil determinar o tamanho fixo ótimo do vetor que armazenaria a informação ordenada por duas razões: ou o programa não funcionaria em máquinas cuja capacidade de memória fosse pequena para conhecer o vetor, ou você teria de criar um vetor para o pior caso e não permitir a sua utilização por usuários que necessitassem de mais memória. A solução é usar memória alocada para armazenar a informação.

Um editor de texto é um programa que ilustra o dilema de memória e o soluciona bem. A maioria dos editores de texto não limita o número de caracteres que eles manipulam, mas em vez disto utiliza toda a memória disponível do computador para armazenar o texto que o usuário inseriu. Por exemplo, assim que cada linha é inserida, o armazenamento é alocado e uma lista encadeada é mantida. Quando uma linha é deletada, essa porção de memória é retornada ao sistema. Uma maneira de implementar tais editores de texto seria usar a seguinte estrutura para cada linha:

```
struct line {
        char text[81];
        int num;        /*numero da linha */
        struct line *next;  /* ponteiro para roxima entrada */
        struct line *prior; /* ponteiro para registro anterior */
} list_entry;

struct line *start; /* ponteiro para primeira entrada na lista */
struct line *last;  /* ponteiro para ultima entrada */
```

Para simplificar, esta estrutura sempre aloca memória o bastante para cada linha possuir extensão de 80 caracteres com uma terminação nula. Na realidade, só o comprimento exato da linha seria alterado. O elemento **num** contém o número da linha de cada linha do texto. Isto permite a você usar ordenações-padrão, função de armazenamento da lista duplamente encadeada **dls-store( )** para criar e manter o arquivo texto como uma lista encadeada.

O programa de um editor de texto simples, mostrado a seguir, possibilita a inserção de linhas em qualquer ponto baseado no número da linha especificada e permite a deleção de qualquer linha. Você também pode listar o texto e armazenar em disco.

O significado geral de operações para um editor está baseado em uma ordenação: a lista encadeada de linhas do texto. A chave de ordenação é o número de cada linha. Você não só pode inserir texto facilmente em qualquer ponto, especificando a linha onde se quer iniciar a inserção, mas também pode deletar texto com facilidade. A única função que pode não ser intuitiva é **patchup( )**. Ela renumera o elemento **num** para cada linha de texto quando inserções ou deleções se fizerem necessárias.

Neste exemplo, a quantidade de texto que o editor pode armazenar está diretamente baseada na quantidade de memória livre no sistema do usuário. Assim, o editor usa automaticamente memória adicional, sem precisar ser reprogramado.

Esta é a razão mais importante para usar alocação dinâmica quando você defrontar-se com um dilema de memória.

O programa como mostrado está limitado, mas o editor de texto básico tem uma estruturação sólida. Você pode melhorá-lo para criar um editor de texto comum.

```
#include "stdio.h"

struct line {
        char text[81];
        int num;        /*numero da linha */
        struct line *next; /* ponteiro para roxima entrada */
        struct line *prior;/* ponteiro para registro anterior */
} list_entry;

struct line *start; /* ponteiro para primeira entrada na lista */
struct line *last;  /* ponteiro para ultima entrada */

main(argc,argv)
int argc;
char *argv[];
{
        char s[80], choice,fname[80];
        struct line *info;
        int linenum=1;

        start=0; last=0;        /* lista de comprimento zero */
        if(argc==2) load(argv[1]);      /* le arquivo */

        do {
        choice=menu_select();
        switch(choice) {
                case 1: printf("entre com numero da linha: ");
                        gets(s);
                        linenum=atoi(s);
                        enter(linenum);
                        break;
                case 2: delete();
                        break;
                case 3: list();
                        break;
                case 4: printf("entre com o nome do arquivo: ");
                        gets(fname);
                        save(fname);    /* escreve no disco */
                        break;
                case 5: printf("entre com o nome do arquivo: ");
                        gets(fname);
                        load(fname);    /* le do disco */
                        break;
```

```
                case 6: exit(0);
                }
        }while(1);
}


menu_select()
{
        char s[80];
        int c;
        printf("1. entrar com o texto\n");
        printf("2. deletar a linha\n");
        printf("3. listar o arquivo\n");
        printf("4. salvar o arquivo\n");
        printf("5. carregar o arquivo\n");
        printf("6. sair\n");
        do {
                printf("\nEntrar com sua escolha: ");
                gets(s);
                c=atoi(s);
        } while(c<0 || c>6);
        return c;
}

enter(linenum)  /* entrada do texto */
int linenum;
{
        struct line *info,*dls_store(),*find();
        char r[81];

        do {
                info=malloc(sizeof(list_entry));
                if(info==0) {
                        printf("\nestouro de memoria");
                        return;
                }

                printf("%d : ",linenum);
                gets(info->text);
                info->num=linenum;
                if(find(linenum)) patchup(linenum,1);
                if(*info->text) start=dls_store(info);
                else break;
                linenum++;

        } while(1);     /* entra em loop */
        return linenum;
}

patchup(n,incr) /* quando o texto e inserido no meio do arquivo,
                   numero de linhas acima devem ser incrementadas
```

```
                    de um e linhas dele tadas precisam ser decremen-
                    tadas de um */

int n;
int incr;
{
        struct line *i, *find();

        i=find(n);

        while(i) {
                i->num=i->num+incr;
               i=i->next;
        }

}
struct line *dls_store(i)       /* armazena em ordem        */
struct line *i;                 /* numero base da linha */
{
        struct line *old,*p;

        if(last==0) {   /* primeiro elemento da lista */
                i->next=0;
                i->prior=0;
                last=i;
                return i;
        }

        p=start;        /* comeca no topo da lista */

        old=0;
        while(p) {
                if(p->num < i->num) {
                        old=p;
                        p=p->next;
                }
                else {
                        if(p->prior) {
                                p->prior->next=i;
                                i->next=p;
                                p->prior=i;
                                return start;
                        }
                        i->next=p;  /* primeiro novo elemento */
                        i->prior=0;
                        p->prior=i;
                        return i;
                }
        }
        old->next=i;    /* coloca no fim */
        i->next=0;
```

```
        i->prior=old;
        last=i;
        return start;
}

delete()
{
        struct line *info, *find();
        char s[80];
        int linenum;

        printf("entre com o numero da linha ");
        gets(s);
        linenum=atoi(s);
        info=find(linenum);

        if(info)  {
                if(start==info)  {
                        start=info->next;
                        if(start) start->prior=0;
                        else last=0;
                }
                else  {
                        info->prior->next=info->next;
                        if(info!=last)
                                info->next->prior=info->prior;
                        else
                                last=info->prior;
                }
                free(info);  /* retorna a memoria para sistema */
                patchup(linenum+1,-1);
                /* decrementa o numero de linhas */

        }
}

struct line *find(linenum)
int linenum;
{
        struct line *info;

        info=start;
        while(info)  {
                if(linenum==info->num) return info;
                info=info->next;      /*pega proximo endereco */
        }
        return 0;        /* nao achou */
}

list()
{
        struct line *info;
```

```
        info=start;

        while(info)  {
                printf("%d: %s\n",info->num,info->text);
                info=info->next;      /* pega proximo endereco */
        }
        printf("\n\n");
}

save(fname)
char *fname;
{
        register int t,size;
        struct line *info;
        char *p;

        FILE *fp;

        if((fp=fopen(fname,"w"))==0)  {
                printf("arquivo inacessivel\n");
                exit(0);
        }
       printf("\nsalvando arquivo\n");

        size=sizeof(list_entry);
        info=start;
        while(info)  {
                p=info->text;/* converte para ponteiro de char */
                while(*p) putc(*p++,fp); /* salva byte por byte */
                putc('\r',fp);  /* finalizador */
                putc('\n',fp);  /* finalizador */
                info=info->next;         /* pega proxima linha */
        }
        putc(EOF,fp);    /* envia EOF */
        fclose(fp);
}

load(fname)
char *fname;
{
        register int t,size,lnct;
        struct line *info, *temp;
        char *p;
        FILE *fp;

        if((fp=fopen(fname,"r"))==0)  {
                printf("arquivo inacessivel\n");
                return;
        }
        while(start)  {
                p=start;
```

```
            start=start->next;
            free(p);
        }

        printf("\ncarregando arquivo\n");

        size=sizeof(list_entry);
        start=malloc(size);
        if(!start)  {
                printf("estouro de memoria\n");
                return;
        }
        info =start;
        p=info->text;    /* converte para ponteiro de char */
        lnct=1;
        while((*p=getc(fp))!=EOF)  {
                while(*p++!='\r') *p=getc(fp);
                getc(fp);          /* linefeed */
                p--;     /* insere finalizador nulo */
                *p='\0';
                info->num=lnct++;
                info->next=malloc(size);      /* aloca memoria */
                if(!info->next)  {

                        printf("estouro de memoria\n");
                        return;
                }
                info->prior=temp;
                temp=info;
                info=info->next;
                p=info->text;
        }
        temp->next=0;    /* ultima entrada */
        last=temp;
        free(info);
        start->prior=0;
        fclose(fp);
}
```

FRAGMENTAÇÃO

Por não fazer tecnicamente parte da linguagem C, mas sim parte da biblioteca C, **malloc( )** e **free( )** têm implementações variadas de compilador para compilador. Os projetistas de alguns compiladores aplicaram grandes esforços nestas rotinas de alocação dinâmica, enquanto outros fizeram um trabalho mal e mal aceitável. Sob todas as

implementações das funções **malloc( )** e **free( )**, fragmentação da memória pode ocorrer, o que pode gradualmente fazer com que pedidos de alocação falhem, mesmo havendo memória suficiente. Em alguns compiladores C isto pode ocorrer rapidamente.

Fragmentação ocorre quando partes da memória livre se encontram entre blocos de memória alocada. Embora a memória livre seja geralmente grande o suficiente para preencher os requisitos de memória, o problema aparece quando as partes individuais são pequenas demais para preencher um pedido, mesmo que tenha memória suficiente, se todas forem somadas. A Figura 4.4 mostra como uma série de chamadas para **malloc( )** e **free( )** pode criar essa situação.

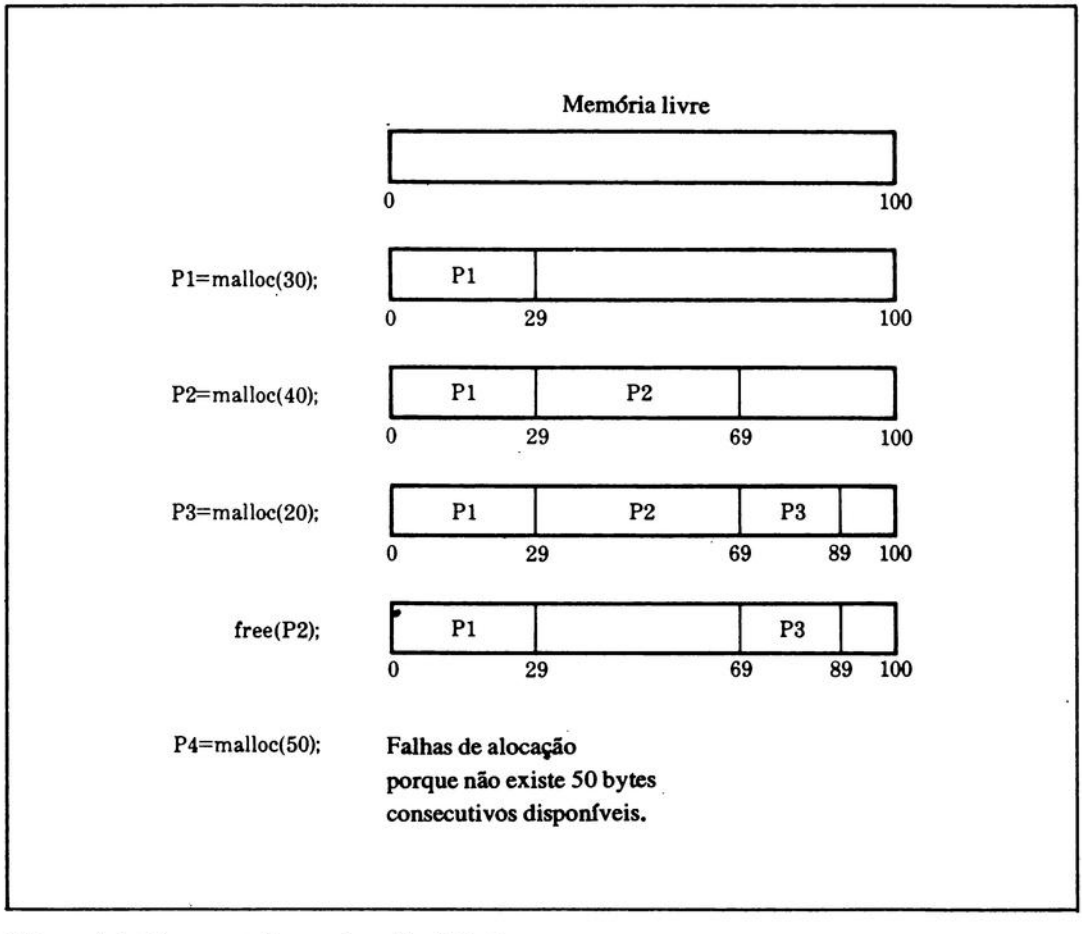
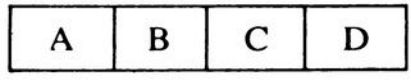


Figura 4.4. Fragmentação em alocação dinâmica.

Você pode evitar alguns tipos de fragmentação se as funções de alocação dinâmica misturam áreas adjacentes de memória. Por exemplo, se as áreas de memória A, B, C e D (mostrado aqui) fossem alocadas e depois as regiões B e C fossem liberadas, B e C teoricamente poderiam ser combinadas pois estão uma ao lado da outra. Porém, se B e D fossem liberadas, não haveria jeito de combiná-las, pois C está localizada entre elas e ainda está sendo usada.



Em primeira instância você pode perguntar por quê, já que B e D estavam livres enquanto C estava sendo alocada, você não poderia simplesmente mover o conteúdo de C para D e combinar B e C. O problema é que o seu programa não teria jeito de saber o que em C teria de ser transferido para D.

Uma maneira de evitar fragmentação em excesso é de sempre alocar quantidades de memória iguais: todas as áreas de alocação podem ser realocadas para requisições subseqüentes e toda memória livre pode ser usada. Se isso não é possível, tente limitar os tamanhos diferentes para apenas alguns. Às vezes, isto pode ser conseguido juntando várias requisições pequenas em uma requisição grande. É claro que você nunca deve alocar mais memória do que necessita, apenas para evitar fragmentação, pois a quantidade de memória gasta vai ultrapassar em muito quaisquer ganhos que você venha a receber por evitar fragmentação. Aqui está outra solução: quando receber um ponteiro nulo de **malloc( )**, escreva toda informação armazenada nas áreas da memória dinâmica para um disco temporário; use **free( )** para liberar toda memória, e depois leia a informação de volta. Isto geralmente funciona, já que a maioria das rotinas de alocação dinâmica mescla áreas de memória livre adjacentes.

ALOCAÇÃO DINÂMICA E INTELIGÊNCIA ARTIFICIAL

Embora C não seja uma linguagem de desenvolvimento de inteligência artificial, você pode usá-lo para experimentar. Uma característica comum de muitos programas de inteligência artificial é a existência de uma lista de itens de informação que podem ser automaticamente estendidos pelo programa enquanto ele "aprende" coisas novas. Em uma linguagem tipo LISP, considerada a primeira de inteligência artificial, a linguagem em si faz manutenção da lista. Em C você precisa programar tais procedimentos, usando listas

encadeadas e alocação dinâmica. Embora o exemplo aqui seja bem simples, os conceitos podem ser aplicados a programas inteligentes mais sofisticados.

Uma área interessante de inteligência artificial tem programas que se comportam como pessoas. O famoso programa Eliza, por exemplo, parece ser um psiquiatra. Seria maravilhoso ter um programa de computador que teria uma conversa sobre qualquer assunto – um ótimo programa para quando você está cansado de programar e se sente só! O exemplo usado aqui é uma versão extremamente simples de tal programa. Usa palavras e suas definições para ter uma conversa com o usuário. Uma técnica comum de muitos programas de "IA" é encadear um item informacional com seu significado. Neste caso, o programa encadeia palavras com os seus significados. A estrutura seguinte armazena cada palavra, sua definição, sua categoria gramatical, e suas conotações:

```
struct vocabulary  {
        char type[3];   /* substantivo,verbo,artigo */
        char word[30];  /* string que contem a palavra */
        char def[128];  /* significado */
        char connotate[3];      /* bom, ruim, neutro */
        struct vocabulary *next;
        /* ponteiro para a proxima entrada */
        struct vocabulary *prior;
        /* ponteiro para o registro anterior */
}  smart;

struct vocabulary *start; /* ponteiro para a primeira entrada */
struct vocabulary *last;  /* ponteiro para a ultima entrada */
```

No programa que segue, você digita uma palavra, seu significado, que tipo de palavra que é e sua conotação de mau, bom ou indiferente. Para armazenar estas entradas de dicionário, uma lista encadeada é criada usando alocação dinâmica. O **dls-store( )** cria e mantém uma lista ordenada duplamente encadeada de dicionário. Após você ter inserido algumas palavras no dicionário, pode começar a sua conversa com o computador. Por exemplo, você digita uma sentença, do tipo **Está um dia bonito.** O programa varre a sentença procurando uma palavra conhecida. Se a encontra, faz um comentário sobre a palavra com base em seu significado.

Se o programa encontra uma palavra que não conhece, ele estimula você a inserir sua definição. Você digita **quit** para sair do modo de conversação.

A função **talk( )** é a parte do programa que mantém a conversa. Uma função de apoio **dissect( )** observa sua sentença palavra por palavra. A variável global **token** armazena sua sentença porque **dissect( )** retorna apenas uma palavra de cada vez, assim a variável que armazena sua sentença precisa ser estável entre chamadas e compartilhada por duas funções. Aqui estão as funções **talk( )** e **dissect( )**:

```
talk()
{
        char w[128],*s, *dissect();
        struct vocabulary *word;

        printf("modo conversacao (quit para sair)\n");
        do  {
                s=token;
                printf(": ");
                gets(token);
                do  {
                        s=dissect(s,w);
                        /* procura substantivo primeiro */
                        if(!*w) break;
                        word=find(w);
                        if(word->type=='n')  {
                                switch(word->connotate)  {
                                        case 'b':
                                                printf("eu
                                                gosto ");
                                                break;
                                        case 'r':
                                                printf("eu
                                                nao gosto ");
                                                break;
                                        case 'i':
                                                break;
                                }
                                printf(word->def);
                                printf("%\n");
                        }
                } while(strcmp(w,"quit"));
        } while(strcmp(w,"quit"));
}

char *dissect(s,w)
char *s,*w;
{
        while(*s== ' ') ++s;

        while(isalpha(*s)) *w++=*s++;
        *w=0;  return s;
}
```

O programa completo é apresentado aqui

```
#include "malloc.h"
#include "stdio.h"
#include "ctype.h"
```

```
struct vocabulary  {
        char word[30];
        char def[128];
        char connotate;
        char type;
        struct vocabulary *next;
        /* ponteiro para a proxima entrada */
        struct vocabulary *prior;
}  smart;

struct vocabulary *start; /* ponteiro para a primeira entrada */
struct vocabulary *last;  /* ponteiro para a ultima entrada */

char token[256];

main()
{


        char s[80], choice;
        struct vocabulary *info;

        start=0;        /* tamanho da lista */
        last=0;
        for(;;)  {
                choice=menu_select();
                switch(choice)  {
                        case 1: enter();
                                break;
                        case 2: delete();
                                break;
                        case 3: list();
                                break;
                        case 4: search();
                                /* procurar uma palavra */
                                break;
                        case 5: save();
                                /* salva a lista no disco */
                                break;
                        case 6: load(); /* le do disco */
                                break;
                        case 7: talk();
                                break;
                        case 8: exit(0);
                }
        }
}

menu_select()
{
        char s[80];
        int c;
```

```
        printf("1. Entrar com uma palavra\n");
        printf("2. Deletar uma palavra\n");
        printf("3. Listar um arquivo\n");
        printf("4. Procurar\n");
        printf("5. Salvar um arquivo\n");
        printf("6. Carregar um  arquivo\n");
        printf("7. Conversar\n");
        printf("8. Sair\n");
        do  {
                printf("\nEntre com sua escolha: ");
                gets(s);
                c=atoi(s);
        }  while(c<0 || c>8);
        return c;
}

enter()
{
        struct vocabulary *info,*dls_store();
        char t[10];

        for(;;)  {
                info=malloc(sizeof(smart));
                if(info==0)  {
                        printf("\nestouro de memoria");
                        return;
                }

                inputs("entre com a palavra: ",info->word,30);
                if(!info->word[0]) break;
                inputs("entre com o tipo (s,v,a): ",t,3);
                info->type=*t;
                inputs("entre com a conotacao (b,r,i): ",t,3);
                info->connotate=*t;
                inputs("entre com a definicao:
                \n",info->def,128);
                start=dls_store(info);
        }
}

inputs(prompt,s,count)

char *prompt;
char *s;
int count;
{
        char p[255];

        do  {
                printf(prompt);
                gets(p);
```

```
                if(strlen(p))count) printf("\nmuito comprida\n");
        }  while(strlen(p))count);
        strcpy(s,p);
}

struct vocabulary *dls_store(i) /*armazena em ordem */
struct vocabulary *i;
{
        struct vocabulary *old,*p;
        if(last==0)  {  /* primeiro elemento da lista */
                i->next=0;
                i->prior=0;
                last=i;
                return i;
        }
        p=start;        /* comeca do topo da lista */

        old=0;
        while(p)  {
                if(strcmp(p->word,i->word)<0){
                        old=p;
                        p=p->next;
                }
                else  {
                        if(p->prior)  {
                                p->prior->next=i;
                                i->next=p;
                                p->prior=i;
                                return start;
                        }
                        i->next=p;
                        i->prior=0;
                        p->prior=i;
                        return i;
                }
        }
        old->next=i;    /*coloca no final */
        i->next=0;
        i->prior=old;
        last=i;
        return start;
}

delete()
{
        struct vocabulary *info, *find();
        char s[80];

        printf("entre com a palavra: ");
        gets(s);
        info=find(s);
```

```
        if(info)  {
                if(start==info)  {
                        start=info->next;
                        if(start) start->prior=0;
                        else last=0;
                }
                else  {
                        info->prior->next=info->next;
                        if(info!=last)
                                info->next->prior=info->prior;
                        else
                                last=info->prior;
                }
                free(info);  /* retorna memoria para o sistema */
        }
}

struct vocabulary *find(word)
char *word;
{
        struct vocabulary *info;
        char  s[80];
        info=start;
       while(info)  {
                if(!strcmp(word,info->word)) return info;
                info=info->next;
        }
        printf("%s desconhecido, por favor entre\n",word);
        enter();
        return 0;       /* nao achou */
}

list ()
{
        register int t;
        struct vocabulary *info;

        info=start;
        while(info)  {
                display(info);
                info=info->next;     /* pega proximo endereco */
        }
        printf("\n\n");
}

display(info)
struct vocabulary *info;
{
        printf("palavra: %s\n",info->word);
        printf("tipo: %c\n",info->type);
        printf("conotacao: %c\n",info->connotate);
```

```
        printf("definicao: %s\n",info->def);
        printf("\n\n");
}

search()
{
        char word[40];
        struct vocabulary *info,*find();

        printf("entre com a palavra para achar: ");
        gets(word);
        if(!(info=find(word))) printf("nao achou\n");
        else display(info);
}

save()
{
        register int t, size;
        struct vocabulary *info;
        char *p;


        FILE *fp;
        if((fp=fopen("smart.dic","w"))==0)  {
                printf("arquivo inacessivel\n");
                exit(0);
        }
        printf("\nsalvando arquivo\n");
       size=sizeof(smart);
        info=start;
        while(info)  {
                p=info; /* converte para ponteiro char */
                for(t=0;t(size;++t)
                        putc(*p++,fp); /* salva byte a byte */
                info=info->next;       /* pega proximo endereco */
        }
        putc(EOF,fp);    /* envia EOF */
        fclose(fp);
}

load()
{
        register int t,size;
        struct vocabulary *info;
        char *p;
        FILE *fp;

        if((fp=fopen("smart.dic","r"))==0)  {
                printf("arquivo inacessivel\n");
                return;
        }
```

```
        while(start)  {
                p=start;
                start=start->next;
                free(start);
        }
        last=0;
        printf("\ncarregando o arquivo\n");

        size=sizeof(smart);
        start=malloc(size);
        if(!start)  {
                printf("estouro de memoria\n");
                return;
        }
        info=start;
        p=info; /* converte para ponteiro char */
        while((*p++=getc(fp))!=EOF)  {
                for(t=0;t<(size-1);++t)
                        *p++=getc(fp);
                        /* carrega  byte por vez */
                dls_store(info);
                info=malloc(size);

                /* aloca memoria para o proximo */
                if(!info)  {
                        printf("estouro de memoria\n");
                        return;
                }
                p=info;
        }
        start->prior=0;
        fclose(fp);
}

talk()
{
        char w[128],*s, *dissect();
        struct vocabulary *word;

        printf("modo conversacao (quit para sair)\n");
        for(;;)  {
                s=token;
                printf(": ");
                gets(token);
                if(!strcmp(token,"quit")) return;
                for(;;)  {
                        s=dissect(s,w);
                        /* procura substantivo primeiro */
                        if(!*w) break;
                        word=find(w);
                        if(word->type=='n')  {
                                switch(word->connotate)  {
```

```
                                        case 'b':
                                                printf("eu
                                                gosto ");
                                                break;
                                        case 'r':
                                                printf("eu nao
                                                gosto ");
                                                break;
                                        case 'i':
                                                break;
                                }
                                printf(word->def);
                                printf("%\n");
                        }


                        else
                                printf("%s ",word->def);
                }
        }
}
char *dissect(s,w)
char *s,*w;
{
        while(*s== ' ') ++s;

        while(isalpha(*s)) *w++=*s++;
        *w=0;  return s;
}
```

Este programa é gostoso e fácil de escrever. Você pode fazê-lo parecer um tanto mais inteligente. Uma maneira é levar o programa a varrer sua sentença para verbos e depois substituir um verbo alternativo em seu comentário. Você também pode fazê-lo perguntar algo de vez em quando.

Capítulo 5

INTERFACE PARA ROTINAS EM LINGUAGEM ASSEMBLY E AO SISTEMA OPERACIONAL

Mesmo com toda a capacidade da linguagem C, existem momentos em que você precisa ou escreve uma rotina usando linguagem assembly, ou usa uma chamada ao sistema operacional. A maneira de fazer ambas varia em alguns aspectos conforme o compilador, mas os procedimentos gerais descritos neste capítulo aplicam-se à maioria dos compiladores C.

Cada processador tem uma linguagem assembly diferente, e cada sistema operacional tem uma estrutura de interface diferente. Além disso, muitos compiladores C têm diferentes "convenções de chamada" que definem como a informação é passada para e de cada função. Este capítulo é baseado no sistema operacional do IBM-PC, no compilador Aztec C, e na linguagem assembly do 8086. Mesmo se você tiver um equipamento diferente, poderá usar as discussões seguintes como guia.

INTERFACE COM A LINGUAGEM ASSEMBLY

Existem três razões para usar uma rotina escrita em assembler:

- Para aumentar a velocidade e eficiência
- Para executar uma função de máquina específica que não está disponível em C
- Para usar uma rotina pronta de caráter geral da linguagem assembly

Embora os compiladores C possam produzir códigos-objetos compactos extremamente rápidos, nenhum compilador cria um código consistente que seja tão rápido ou compacto como um escrito por um programador competente em linguagem assembly.

Normalmente uma pequena diferença não tem muita importância, nem fará muita diferença o tempo extra necessário para se escrever em assembler. No entanto, existem casos especiais em que uma função específica precisa ser codificada em assembler, desta forma será executada mais rapidamente. Isto é válido para uma função se ela vai ser usada freqüentemente e afetará de maneira significativa o tempo de execução em última instância. Um bom exemplo é um pacote matemático de ponto flutuante. Algumas vezes o hardware e alguns periféricos necessitam de um *timming* exato, e você precisa codificá-los em assembler para solucionar a requisição de um *timming* restrito.

Muitos computadores, incluindo as máquinas baseadas no 8086, têm capacidades úteis que não podem ser executadas usando-se os operadores da linguagem C diretamente. Por exemplo, você não pode trocar o segmento de dados com qualquer das instruções da linguagem C, e você não pode implementar um software de interrupção ou controlar o conteúdo de registradores específicos, usando a linguagem C.

Em ambientes profissionais de programação, sub-rotinas de biblioteca são freqüentemente utilizadas por suas capacidades de manipulação com gráficos e matemática de ponto flutuante.

Algumas vezes você precisa utilizá-las em código-objeto porque o projetista não fornece o código-fonte. Ocasionalmente, você pode simplesmente encaixar essa rotina com o código compilado, outras vezes, precisa escrever um módulo de interface para corrigir quaisquer diferenças na interface usadas pelo seu compilador e das rotinas por você obtidas.

Existem primeiramente duas maneiras de integrar módulos codificados em assembler em seus programas em C. A primeira é codificar a rotina separadamente, e juntá-la ao código em assembler, e enlaçá-la com o resto do seu programa. A segunda, é usar a capacidade de escrever diretamente em assembler de muitos compiladores de C.

Está fora do escopo deste livro ensinar programação em linguagem assembly. Este capítulo assume que você já está familiarizado com a linguagem assembly de seu computador; os exemplos dados servem somente como referência.

CHAMADAS CONVENCIONAIS DO COMPILADOR C

Uma chamada convencional é um método com o qual o compilador C passa valores de parâmetros pela função. A solução usual usa ou os registradores internos da CPU ou a pilha de sistema para passagem de informação entre funções independentes. Geralmente, os compiladores C usam a pilha para passar argumentos para as funções. Se o

argumento é um dos sete tipos de dados intrínsecos (**char, short int, int, long int, unsignal int, float,** ou **double**), o valor real do dado é colocado na pilha. No caso de uma função terminal da linguagem C, ela passa o valor retornado de volta para rotina chamadora. Esse valor retornado é normalmente colocado em um registrador, embora teoricamente ele pudesse ser passado na pilha.

Um aspecto interessante em uma convenção de chamada consiste exatamente no fato que os registradores precisam ser conservados pela função enquanto ela é executada e que algumas podem ser usadas livremente. Freqüentemente o compilador requer que determinados registradores do processador fiquem intactos.

Módulos em linguagem assembly devem preservar o conteúdo dos registradores ou por não usá-los ou colocando-os na pilha antes de usá-los. Quaisquer outros registradores estão geralmente livres para serem usados.

Quando você escreve um módulo em linguagem assembly que precisa ser interfaceado com o código compilado por seu compilador C, precisa seguir todas as convenções que estão definidas e usadas por seu compilador. Somente assim você tem rotinas em linguagem assembly corretamente interfaceadas com o seu código em C.

CRIANDO UMA FUNÇÃO EM CÓDIGO ASSEMBLY

Em seu manual do compilador C deve haver uma seção onde é descrito como os parâmetros são passados para as funções, quais registradores precisam ser rearmazenados e como o valor retornado pela função é passado de volta à rotina chamadora. Antes de você tentar escrever uma função em linguagem assembly para o seu compilador C, precisa ter acesso a essas informações. Depois de saber as convenções de chamada do seu compilador, você simplesmente escreve a função em linguagem assembly e enlaça-a ao seu programa, usando o programa enlaçador.

Por exemplo, assuma que por alguma razão é necessário codificar a seguinte função em assembler:

```
sample(a,b)
int a,b;
{
        a=a+b;
        return a;
}
```

Para que essa função seja chamada pelo seu programa em C, seu primeiro passo é determinar como os argumentos são passados para a função (você encontrará a informação no manual do seu compilador C). Em quase todos os compiladores, os argumentos são passados na pilha. No compilador Aztec C, os argumentos são empurrados na pilha na ordem inversa em que aparecem na chamada. Por exemplo, se **sample( )** é chamada com

```
sample(10,20);
```

o valor 20 é colocado primeiro e o valor 10 é colocado em segundo. Lembre-se de que se os argumentos de chamada são variáveis simples, seus valores são passados na pilha. Para uma série de caracteres ou um vetor, um ponteiro (um endereço) é passado na pilha.

Seu segundo passo é saber quais registradores devem ser preservados pela sua função em linguagem assembly. Se você usar o compilador Aztec em um IBM-PC, os registradores de segmento – assim como BP, BX, SI e DI – devem ser preservados. Você pode fazer isto colocando-os na pilha e retirando-os da pilha antes de retornar.

Seu último passo é saber como retornar o valor da função. O compilador Aztec requer que o valor seja colocado em Ax e que o flag Z seja setado de acordo com seu valor. A função **sample( )** em linguagem assembly é

```
codeseg segment para public 'code'
dataseg segment para public 'data'
dataseg ends

        assume cs:codeseg,ds:dataseg,es:dataseg,ss:dataseg
        public sample_
sample_proc  near
        push    bx
        push    bp
        push    si
        push    di
        mov     bx,sp    ;coloca tos em bx
        mov     ax,word ptr 10[bx]      ;este e' o parametro a
        mov     ax,word ptr 12[bx]      ;este e' o parametro b
        mov     word ptr 10[bx],ax      ;resultado em a
        pop     di
        pop     si
        pop     bp
        pop     bx
        ret
sample_endp
```

```
codeseg ends
dataseg segment para public 'data'
dataseg ends
        end
```

Observe que todos os registradores apropriados são salvos com instruções **push** e **pop** e que os argumentos são acessados na pilha. Se você não está familiarizado com o código do 8086, aqui está um exemplo. O código

```
mov  bx, sp                ; põe tos (topo da pilha) corrente em bx
mov  ax, word ptr10 [bx] ; esse é o parâmetro a
```

coloca o endereço do topo da pilha no registrador **bx** e então move a décima palavra abaixo na pilha, que é o parâmetro **a,** no registrador **ax.** Parâmetros são as décima e vigésima palavras abaixo porque o endereço de retorno e as instruções **push** para salvar registradores usam dez bytes; no entanto, os parâmetros são encontrados a começar pelo décimo byte abaixo da pilha.

Você pode usar este pequeno programa

```
main()
{
        printf("%d ",sample(10,20));

        printf("%d ",sample(30,40));
}
```

desta forma, **sample( )** é incluído na linha de enlace. Se faz isso e executa o programa, os números **30** e **70** serão impressos na tela.

Lembre-se, no entanto, de que cada compilador e cada processador é diferente. Você precisa estudar seus manuais do usuário.

USANDO # ASM E # ENDASM

Muitos compiladores C acrescentam uma extensão às diretivas do pré-processador C que permitem que códigos em assembler façam parte de diferentes

funções-padrão C. (O compilador UNIX não suporta esta extensão.) Existem duas vantagens: primeiro não é pedido ao programador para escrever todo o código da interface necessário para fazer que um módulo em assembly funcione como uma função da linguagem C; segundo, todo o código está em um lugar, tornando mais fácil o suporte.

As duas diretivas do pré-processador que tornam isto possível são **#asm** e **#endasm.** **#asm** inicia um bloco de código em assembler e **#endasm** finaliza o bloco. Todo código dentro de um bloco **#asm** precisa ser corretamente de acordo com o assembler de seu computador. O compilador C simplesmente passa este código, sem alteração, para a fase assembler do seu compilador.

Por exemplo, a função **init–port1()** envia 255 e então 0 para o port 26, usando código assembly do 8086.

```
init_port1()
{
        printf("Initializing Port\n");
#asm
        out 26,256
        out 26,0
#endasm
}
```

O compilador C produz automaticamente o código apropriado para salvar registradores e para retornar quaisquer valores necessários da função.

Se você desejar usar este método para codificar **sample,** poderá usar o compilador C para preencher todo o suporte. Você precisaria somente possuir o corpo da função, como visto a seguir:

```
sample(a,b)
int a,b;
{
#asm
     mov  ax,word ptr 8[bp]
     add  ax,word ptr 10[bp]
     mov  word ptr 8[bp], ax
     mov  ax,word ptr 8[bp]
#endasm
}
```

O compilador C contém todo o suporte de costume para montar e retornar de uma chamada de função. Tudo o que você precisa fazer é possuir o corpo da função e seguir as convenções de chamada para acessar os argumentos. Qualquer que seja o método utilizado, você está criando situações dependentes da máquina, o que tornará seu programa difícil de ser instalado em outra máquina. Para situações que exigirem o uso do código assembly, no entanto, vale a pena o esforço, normalmente.

QUANDO CODIFICAR EM ASSEMBLER

A maioria dos programadores só codifica em assembler quando é absolutamente necessário, porque é uma codificação difícil. Como regra geral, não use – ele cria muitos problemas. No entanto, existem dois casos nos quais codificar em assembler faz sentido. Um dos casos é quando não existe nenhuma outra maneira de se fazer – por exemplo, quando você precisa interfacear-se diretamente com o hardware do periférico que não pode ser operado usando-se C.

Outro caso para o assembler é quando o tempo de execução do programa em C precisa ser reduzido. Neste caso, você deve cuidadosamente escolher as funções que codificará em assembler. Se você as codificar erradamente, observará uma pequena mudança na velocidade. Se escolher a forma correta, seu programa voará. Para determinar qual função precisa ser recodificada, você precisa rever o fluxo operacional de seu programa. Geralmente são as funções usadas dentro dos laços (loops) que você deve programar em assembler porque elas são executadas repetidamente. Usar assembler para codificar a função que é usada somente uma vez ou duas não oferecerá uma melhoria significante de velocidade, mas usar assembler para codificar uma função que é usada muitas vezes oferece um acréscimo. Por exemplo, considere a seguinte função

```
main()
{
     register  int t;
     init()
     for (t=0; t<1000; ++t) {
               phase1();
               phase2();
               if (t==10 phase3();
}
byebye()
}
```

Recodificar **init( )** e **byebye( )** não deve afetar de forma mensurável a velocidade deste programa, porque eles são executados somente uma vez. Tanto **phase1( )** como **phase2( )** são executados 1000 vezes, e recodificá-los é mais apropriado para ter-se um efeito maior no tempo de processamento do programa. A **phase3( )** é executada uma só vez, mesmo estando dentro do loop; assim, recodificar esta função em assembler provavelmente não valerá a pena.

Refletindo com cuidado você pode aumentar a velocidade do seu programa, recodificando somente poucas funções em assembler.

INTERFACE COM O SISTEMA OPERACIONAL

Em razão de muitos programas em C caírem na categoria de programas de sistemas, é freqüentemente necessário se interfacear diretamente com o sistema operacional, desconsiderando a interface normal do programa em C, que executa as operações de E/S. Existem também funções particulares do sistema operacional, que você queira usar, que não podem ser acessadas por seu compilador C. Por estas razões, usar recursos de baixo nível do sistema operacional é ocorrência comum em programação C.

Muitos sistemas operacionais largamente em uso hoje em dia em microcomputadores incluem:

- PC-DOS ou MS-DOS
- CP/M
- UNIX
- Apple DOS

Todos os sistemas operacionais têm um conjunto de funções que podem ser usadas, por exemplo, para abrir arquivos em disco, ler e escrever caracteres de e para o console, e alocar memória para executar um programa. A maneira como estas funções são acessadas varia de sistema para sistema, mas eles tendem a usar o conceito geral de uma *tabela de jumps*. Em um sistema operacional como o CP/M, chamadas ao sistema são executadas através de uma instrução CALL a uma região específica de memória, com o código da função desejado em um registrador. No PC-DOS, é usado um software de interrupção. Em ambos os casos, a tabela de *jumps* direciona a função apropriada para o seu programa. A Figura 5.1 mostra como um sistema operacional e sua tabela de *jumps* devem aparecer na memória. Não é possível discutir aqui todos os sistemas operacionais. Este capítulo

focaliza somente o PC-DOS porque é mais largamente utilizado. No entanto, as técnicas gerais aplicadas aqui são aplicadas na maioria dos sistemas operacionais.

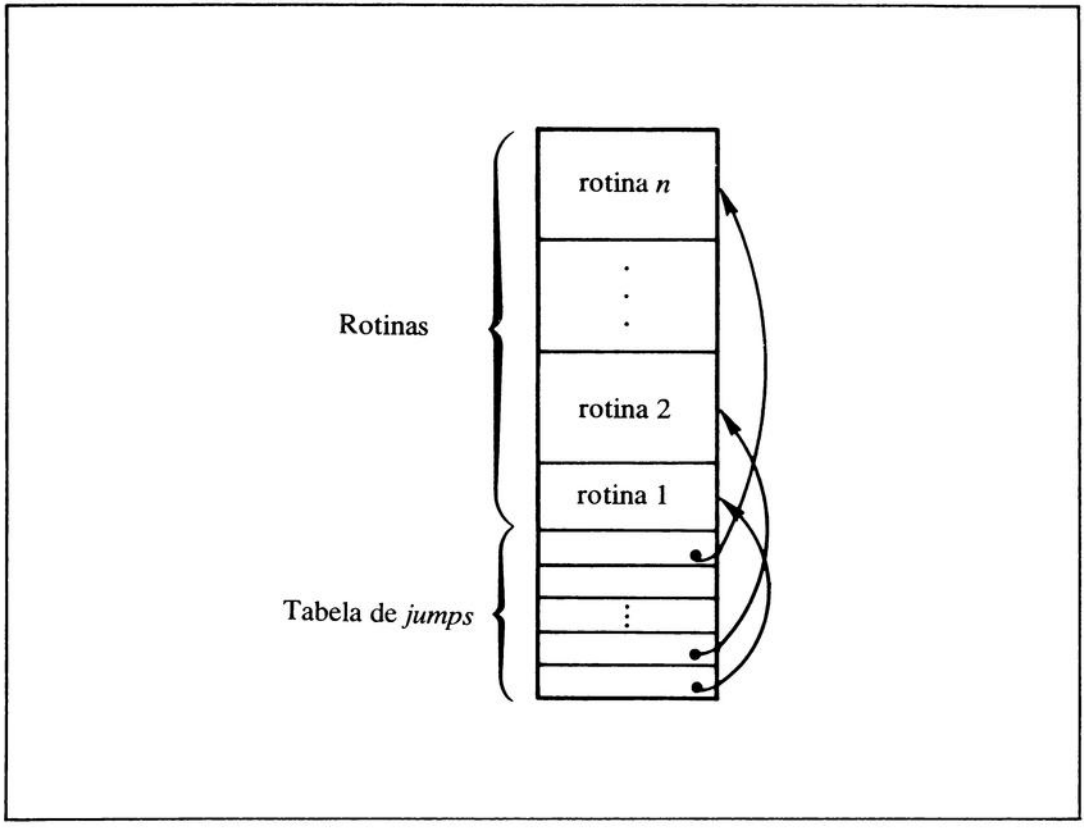


Figura 5.1. Tabela de *jumps* do sistema operacional.

ACESSANDO OS RECURSOS DO PC-DOS

No PC-DOS as funções do sistema operacional são acessadas através de interrupções do software. Cada interrupção tem sua categoria própria de funções que é acessada, e o valor do registrador AH determina estas funções. Se mais informação for necessária é passada via os registradores AL, BX, CX e DX. O sistema operacional PC-DOS é dividido no BIOS (Basic I/O System) e DOS (Disk Operating System). O BIOS executa as rotinas de mais baixo nível que o DOS usa para executar as funções de nível mais alto. No entanto, as duas sobrepõem-se. Felizmente para os nossos propósitos, elas são acessadas basicamente da mesma maneira. Uma lista parcial dessas interrupções é vista aqui.

Interrupt	Function
10h	vídeo E/S
13h	disco E/S
16h	teclado E/S
17h	impressora E/S
21h	chamadas do DOS

Para obter uma lista completa com explicações, referencie ao manual IBM *Technical Reference*.

Cada uma dessas interrupções está associada a um número de opções que podem ser acessadas, dependendo do valor do registrador AH, quando chamada. A Tabela 5-1 mostra uma lista parcial de opções disponíveis para cada uma dessas interrupções.

Tabela 5.1. Opções disponíveis para algumas interrupções

Registrador AH	Função
Funções do BIOS para E/S do vídeo – Interrupção 10h	
0	Seleciona modo de operação do vídeo para AL = 0: 40 x 25 branco e preto 1: 40 x 25 colorido 2: 80 x 25 branco e preto 3: 80 x 25 colorido 4: 320 x 200 gráfico colorido 5: 320 x 200 gráfico branco e preto 6: 340 x 200 gráfico branco e preto
1	Seleciona linhas do cursor CH os bits de 0 a 4 contêm o início da linha os bits de 5 a 7 são 0 CL os bits de 0 a 4 contêm o fim da linha os bits de 5 a 7 são 0
2	Seleciona posição do cursor DH: linha DL: coluna BH: número da página do vídeo
3	Lê a posição do cursor BH: número da página do vídeo

Tabela 5.1. Opções disponíveis para algumas interrupções (continuação)

Registrador AH	Função
	Retorna DH: linha DL: coluna CX: modo do vídeo
4	Lê a posição da caneta *(light-pen)* Retorna se AH = 0 caneta não conectada se AH = 1 caneta conectada DH: linha DL: coluna CH: linha de rastejo (0 a 199) BX: ponto da coluna (0 a 319 ou 0 a 639)
5	Seleciona página de vídeo ativa AL pode ser de 0 a 7
6	Movimentação ascendente do vídeo *(scroll page up)* AL: linhas a movimentar, 0 para mover todas CH: linha do canto superior esquerdo da movimentação CL: coluna do canto superior esquerdo da movimentação DH: linha do canto superior direito da movimentação DL: coluna do canto superior direito da movimentação BH: atributo a ser usado em uma linha em branco
7	Movimentação descendente do vídeo *(scroll page down)* Idêntico ao anterior
8	Lê caractere da posição do cursor BH: página do vídeo Retorna AL: caractere lido AH: atributo
9	Escreve caractere e atributo na posição do cursor BH: página do vídeo BL: atributo CX: número de caracteres a escrever AL: caractere

Tabela 5.1. Opções disponíveis para algumas interrupções (continuação)

Registrador AH	Função
10	Escreve caractere na posição corrente do cursor BH: página do vídeo CX: número de caracteres a escrever AL: caractere
11	Seleciona "pallete" de cor BH: número do "pallete" BL: cor
12	Escreve um ponto DX: número da linha CX: número da coluna AL: cor
13	Lê um ponto DX: número da linha CX: número da coluna Retorna AL: ponto lido
14	Escreve um caractere na tela e avança o cursor AL: caractere BL: cor do primeiro plano BH: página do vídeo
15	Lê estado do vídeo Retorna AL: modo corrente AH: número de colunas da tela BH: página de vídeo ativa
Funções do BIOS para E/S do disco – Interrupção 13h	
0	Desliga o sistema a disco
1	Lê o estado do disco Retorna AL: estado (veja no manual IBM *Technical Reference*)
2	Lê setores da memória DL: número do acionador *(driver)* DH: número da cabeça

Tabela 5.1. Opções disponíveis para algumas interrupções (continuação)

Registrador AH	Função
	CH: número da trilha CL: número do setor AL: número de setores a ler ES:BX: endereço do buffer Retorna AL: número de setores lidos AH: 0 se bem-sucedidas, senão estado encontrado
3	Escreve setores no disco (idem ao item anterior)
4	Verificação (idem ao item anterior)
5	Formatação de uma trilha DL: número do acionador DH: número da cabeça CH: número da trilha ES:BX: setor de informações
	Funções do BIOS de E/S do teclado – Interrupção 16h
0	Lê código de varredura Retorna AH: código de varredura AL: código do caractere
1	Obtém estado do buffer Retorna ZF: 1 então o buffer está vazio 0 então existem caracteres esperando em AX como descrito acima
2	Obtém estado do teclado (veja manual IBM *Technical Reference)*
	Funções do BIOS de E/S da impressora – Interrupção 17h
0	Imprime um caractere AL: caractere DX: número da impressora Retorna AH: estado
1	Inicializa impressora DX: número da impressora

Tabela 5.1. Opções disponíveis para algumas interrupções (continuação)

Registrador AH	Função
	Retorna AH: estado
2	Lê estado DX: número da impressora Retorna AH: estado
	Chamadas as funções de mais alto nível do DOS – Interrupção 21h (Lista parcial)
1	Lê caractere do teclado Retorna AL: caractere
2	Exibe um caractere na tela DL: caractere
3	Lê um caractere de uma porta assíncrona Retorna AL: caractere
4	Escreve um caractere em uma porta assíncrona DL: caractere
5	Imprime um caractere em um periférico de impressão DL: caractere
7	Lê caractere do teclado mas não o exibe Retorna AL: caractere
B	Checa estado do teclado Retorna AL: OFFH se existir tecla pressionada; de outra forma retorna 0
D	Desliga ("resseta") o disco
E	Seleciona o *driver default* DL: número do *driver* (0 = A, 1 = B, ...)
11	(4E para versões anteriores a 2.X) Procura arquivo DX: endereço do FCB

Tabela 5.1. Opções disponíveis para algumas interrupções (continuação)

Registrador AH	Função
	Retorna AL: 0 se encontrado, FFh se não existir o nome no endereço de transferência a disco
12	(4F para versões anteriores a 2.X) Procura por uma nova ocorrência do arquivo O mesmo que o item anterior
1A	Seleciona endereço de transferência a disco DX: endereço de transferência a disco
2A	Pega data do sistema Retorna CX: ano (1980-2099) DH: mês (1-12) DL: dia (1-31)
2B	Seleciona data do sistema CX: ano (1980-2099) DH: mês (1-12) DL: dia (1-31)
2C	Pega hora do sistema Retorna CH: hora (0-23) CL: minutos (0-59) DH: segundos (0-59) DL: centésimos (0-99)
2D	Seleciona hora do sistema CH: hora (0-23) CL: minutos (0-59) DH: segundos (0-59) DL: centésimos (0-99)

Existem basicamente duas maneiras de acessar as funções encontradas na Tabela 5.1. A primeira é através do uso de funções de chamada intrínsecas do sistema, freqüentemente chamadas do **dos( )** ou **bdos( )**, que são oferecidas com a maioria dos compiladores. A segunda é através de uma interface em linguagem assembly.

USANDO DOS() PARA ACESSAR FUNÇÕES DO SISTEMA

A maioria dos compiladores para o sistema operacional PC-DOS contém uma função na sua biblioteca-padrão chamada **dos( )** ou **bdos( )**. É usada para executar a interrupção de chamada 21h para funções de alto nível no sistema operacional. Essa função é dependente da máquina; leia o manual do seu compilador para determinar a exata seqüência de chamada. Essa discussão usa a função **dos( )** do Aztec C. Ela leva a forma geral.

dos (*número da função*, **BX, CX, DX, DI, SI**)

onde o *número da função*, **BX, CX, DX, DI** e **SI** são inteiros que contêm o valor a ser colocado nestes registradores, no instante da chamada. O valor retornado do **dos( )**é o valor do registrador AL.

Em um capítulo anterior foi feito uso de uma chamada ao sistema que observa se alguma tecla foi pressionada. Aqui é a função chamada **kbhit( )** que retorna VERDADEIRO se a tecla foi pressionada, e FALSO se não, através do uso da interrupção 21h número Bh como visto aqui. Lembre-se de que números hexadecimais são precedidos de **0x**, que avisa ao compilador que um número hexa vem a seguir.

```
kbhit()    /* específico para PC-DOS */
{
      return (dos (0xB, 0, 0, 0, 0, 0));
}
```

Observe que zeros foram usados para todos os argumentos, com exceção do primeiro porque nenhuma outra informação foi necessária. Geralmente, quando um registrador específico não é usado na chamada, ele pode ter qualquer valor a ele atribuído como conteúdo deste registrador.

Você freqüentemente precisará se interfacear com o sistema operacional para usar uma função que não está na biblioteca-padrão C.

Uma função que parece ser deixada de fora por muitos compiladores para IBM-PC é a função para impressão de caracteres em uma impressora! No entanto, você pode desenvolver uma função para fazer isto usando a função chamada número 5 do DOS, que imprime um caractere na printer. Essa pequena função imprime uma série de caracteres com terminação nula padrão C para um periférico de impressão-padrão:

```
prints(s)
char *s;
{
     while (*s) dos(0x5, 0,0, *s++,0, 0)
{
```

Outra função que normalmente falta na biblioteca-padrão é uma que leria e escreveria caracteres de uma porta serial assíncrona. Você usa essa porta se quer escrever um programa de modem, por exemplo. Aqui estão duas funções que você poderia usar para acessar a porta serial, usando a função 3 do DOS para ler um caracteres e a função 4 para escrever um caractere:

```
put_async(ch)
char ch;
{
     dos (0x4, 0, 0, ch, 0, 0)
}

get_async()
{
     return (dos (0x3, 0, 0, 0, 0, 0)
}
```

Um exemplo mais complexo usando a chamada de interrupção de sistema 21h é a função **dir–list()** aqui mostrada. O programa lista todos os arquivos no diretório de trabalho corrente. Funciona para os sistemas PC-DOS 2.X e maiores.

```
dir_list()      /* lista i diretorio para PCDOS2.X */
{
        char pp[44];
        char done;

        dos(0x1a,0,0,pp,0,0);

        dos(0x4e,bx,0,"*.*",di,si);
        printf("%s\n",&pp[30]);
        do  {
```

```
            done=dos(0x4f,bx,cx,dx,di,si);
            if(done==-1)  break;
            printf("%s\n",&pp[30]);
        } while(1);
}
```

Essa função trabalha da seguinte maneira. De acordo com o manual IBM *Technical Reference,* um buffer de transferência de disco precisa ser "inicializado" primeiro. Isso pode ser feito passando o endereço do vetor **pp** para DOS via a chamada na qual o endereço é passado no registrador DX.

```
dos(0x1a,0,0,pp,0,0);
```

Como o restante dos registradores não são utilizados, são substituídos por zero.

Agora execute uma chamada à função 4Eh para encontrar o primeiro arquivo que combina *(matches)* com o caractere-chave *.* :

```
dos(0x4e,0,0,"*.*",0,0);
```

Arquivos subseqüentes são encontrados usando a função 4Fh da seguinte maneira:

```
done=dos(0x4f,0,0,0,0,0);
```

Quando a função retorna –1, o último nome de arquivo foi encontrado. Os nomes dos arquivos são colocados no buffer de transferência do disco definido pela primeira chamada ao DOS. Os nomes dos arquivos são impressos pela assertiva comum **printf( )**, porém, você terá de fazer um pequeno cálculo, pois o nome do arquivo no buffer de transferência do disco começa no byte 30.

Você pode usar o **dos( )** para executar qualquer interrupção 21h, mas aquelas interrupções que retornam informação em outros registradores além do AL não funcionarão – a informação será perdida. Embora alguns compiladores C tenham também um tipo de chamada de função **bios( )** para executar a rotina BIOS, muitos não têm, então,

para interfacear chamadas DOS que devolvem valores múltiplos em vários registradores e usam rotinas BIOS, você precisa de uma interface para linguagem assembly.

USANDO ASSEMBLY PARA INTERFACE ÀS FUNÇÕES BIOS E DOS

Vamos supor que você queira mudar o modo de tela durante a execução de um programa. Para o PC-DOS, os sete modos que os adaptadores gráficos coloridos podem ter são

0	40×25 BW
1	40×25 color
2	80×25 BW
3	80×25 color
4	320×200 color graphics
5	320×200 BW graphics
6	340×200 BW graphics

Embora alguns compiladores C tenham funções especiais que alteram o modo de tela, muitos não têm. A função **mode( )**, aqui apresentada, executa uma chamada ao BIOS através da interrupção 10h número 1 – modo "setado" – usando o parâmetro **c** como o modo a ser atribuído. Esse código funciona apenas para o Aztec C, mas você pode modificá-lo para que ele se adapte às convenções de chamada do seu compilador.

```
mode (c)
char c;
{
#asm
            mov   bp,sp
            mov   ax,word ptr 8[bp]
            mov   ah,0
            int   010h
#endasm
}
```

O modo limpa a tela usando a função 6 da interrupção 10h do BIOS:

```
clear()
{
#asm
            mov   al,0
            mov   ah,6
            int   010h
#endasm
}
```

Outro exemplo de interface ao BIOS através da linguagem assembly é a função **goto-xy( )**, que posiciona o cursor nas coordenadas X e Y especificadas:

```
goto_xy(c, r)
char c, r;
{
#asm
            mov   dl, 8[bp]
            mov   cl, 10[bp]
            mov   ah, 2
            mov   bh, 0
            int   010h
#endasm
}
```

Para o IBM-PC (0, 0) é o canto superior esquerdo da tela.

USANDO CÓDIGOS DE VARREDURA DO TECLADO DO PC

Uma das mais frustrantes experiências que você pode ter ao trabalhar com o IBM-PC ou compatível é tentar usar as teclas de seta (assim como INS, DEL, PGUP, PGDN, END, e HOME) e a teclas de função. Estas teclas não retornam os usuais caracteres de 8 bits como o restante das teclas o fazem. Quando você pressiona uma tecla, o PC gera um valor de 16 bits chamado *código de varredura*. O código de varredura consiste em um byte de menor ordem, que, no caso de uma tecla normal, contém o código ASCII da tecla, e um byte de maior ordem que contém a posição da tecla. Para a maioria das teclas, esses códigos de varredura são convertidos em valores ASCII de 8 bits pelo

sistema operacional. Mas, para algumas teclas, tais como as de função e as de seta, isto não acontece, porque o código de caractere para uma tecla especial é 0. Isto significa que você precisa usar a posição do código para determinar qual tecla foi pressionada. A rotina para ler um caractere do teclado lê as teclas especiais. Embora poucos compiladores C tenham rotinas implementadas para ler estas teclas especiais, a maioria não as tem.

A maneira mais fácil de acessar as teclas do C é escrever uma função em linguagem assembly que chame a interrupção 16h para ler o código de varredura.

```
getarrow()
{
#asm
      mov   ah,0
      int   016h
#endasm
}
```

Depois da chamada, o código de varredura e o código do caractere já estão em AX, que é o registrador usado para retorno de informação. Depois da chamada para interromper 16h função 0, o código da posição está em AH e o código do caractere em AL.

O truque para usar **getarrow( )** é saber que quando uma tecla especial é pressionada, o código do caractere é 0. Neste caso você então decodifica código de posição para determinar qual tecla foi digitada. Usar **getarrow( )** para fazer todas as entradas por teclado requer que a rotina de chamada tome decisões com base nos conteúdos de AH e AL. Aqui está um pequeno programa que ilustra uma maneira de se fazer isto.

```
main()     /* exemplo de código de varredura */
{
      union      scan {
            int c;
            char ch[2];
      } sc;
      {
      do     { /* leitura do teclado */
                  sc.c=getarrow();
                  if(sc.ch[0]==0 { /* é uma tecla especial */
                        print ("numer d tecl especia %d,sc.ch[1]);
            }
            else putchar(sc.ch[0]);
      }while (sc.ch[0]!='q');
}
```

O uso da **união** permite-lhe decodificar as duas metades do código de varredura retornados de **getarrow( )**.

Você pode determinar os códigos de varredura ou consultando o manual IBM *Technical Reference* ou usando o pequeno programa que acabamos de mostrar para determinar os valores experimentalmente (o segundo método é mais divertido). Para ajudá-lo a começar, aqui estão os códigos de varredura das teclas de seta:

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

Para a completa integração das teclas especiais com as teclas normais é necessário escrever funções de entrada especiais e passar sobre as funções usuais **gets( )** encontradas na biblioteca-padrão do C. Embora isto seja uma boa solução, é a única maneira. Por outro lado, a vantagem é que seu programa pode assim permitir ao usuário trabalhar com todo o teclado do PC.

MÚSICA

O IBM-PC tem um falante que é normalmente usado para "bipar" em condições de erro. Você também pode fazer com que este falante produza tons musicais. O método aqui apresentado produz um timbre interessante e pode ser usado para produzir efeitos especiais únicos.

Para produzir um tom, é necessário que você crie uma pulsação: quando uma corrente é aplicada às bobinas do falante, e este faz um movimento para dentro; quando a corrente é desligada o falante se movimenta para fora. Se você provocar uma série de sinais liga/desliga enviados a, digamos, 100 vezes por segundo, então um tom de 100 hz é produzido.

No IBM-PC, o falante é ligado através do envio de um valor 2 à porta 61h. O falante é desligado ao enviar-se um 0 à porta 61h. A maioria dos compiladores C tem funções de saída de um byte para uma porta em sua biblioteca-padrão. A função aqui usada tem a forma geral

outportb *(port, value);*

onde *port* é o número da porta e *value* é o byte a ser enviado. O pequeno programa aqui apresentado converte o que você digita no teclado em tons musicais de *pitchs* variados. Você pode facilmente modificar a função **play( )** para alterar a duração ou o *pitch*.

```
main()  /* musica */
{
        int note, length;

        length=10;
        do  {
                note=getch();
                play(note,length);
        } while(note!='q');
}

play (note,l)
int note,l;
{
        int t, tone;

        l=l+1000/note;
        for(;l;l--)  {
                tone=note;
                outp(0x61,2);
                outp(0x61,0);
                for(;tone;--tone);
        }
}
```

Com alguma prática é possível aprender a tocar algumas musiquinhas no teclado.

CONCLUSÕES FINAIS SOBRE INTERFACE AO SISTEMA OPERACIONAL

Este capítulo só dá uma idéia superficial do que pode ser feito através do uso dos recursos do sistema criativamente. Para integrar seu programa com o sistema operacional completamente, você precisa ter acesso às informações que descrevem todas as funções com detalhes.

Existem muitas vantagens em usar funções do sistema operacional. A primeira é que elas podem fazer melhor uso das características especiais de um dado sistema de computação; isto faz com que seu programa pareça mais profissional. A segunda, por passar sobre algumas funções internas do C, em favor das funções do sistema operacional, possibilita criar programas que rodem mais rápido e usem menos memória. A terceira, por você acessar as funções que não estão disponíveis na biblioteca-padrão do C.

No entanto, usar funções do sistema operacional tem seu preço. Você está criando mais problemas para você mesmo quando usa funções do sistema operacional em vez das funções-padrão do C, porque seu código não é muito portátil. Você também pode tornar-se dependente de versões específicas de um sistema operacional e de um compilador C, que criará problemas de compatibilidade quando você distribuir seus programas. Somente você poderá decidir quando e se deve introduzir dependências da máquina e do sistema operacional em seus programas.

Capítulo **6**

ESTATÍSTICA

Todos que possuem ou têm acesso freqüente a um computador usam-no em algum ponto para efetuar *análises estatísticas*. Esta análise pode aplicar-se em monitorar ou tentar prever o movimento de preços de ações em uma carteira de investimentos, efetuar testes clínicos para estabelecer limites seguros para uma nova droga, ou mesmo calcular a média de gols da seleção dente-de-leite no último campeonato. O ramo da matemática com a condensação, manipulação e extrapolação de dados é chamado *estatística*.

Como disciplina, análise estatística é bem recente. Nasceu em meados de 1700 de estudos de jogos de sorte. Realmente, probabilidade e estatística estão relativamente relacionadas. A análise estatística moderna surgiu na virada deste século, quando tornou-se possível amostrar e trabalhar com grandes conjuntos de dados. O computador tornou possível correlacionar e manipular ainda maiores quantidades de dados, rapidamente, e converter estes dados para uso imediato. Hoje, por causa do aumento crescente de informação criada e usada pelo governo e pela *mídia* todos os aspectos da vida são carregados de muita informação estatística. É difícil escutar o noticiário pelo rádio ou pela televisão ou ler um artigo no jornal sem ouvir alguma informação estatística.

Embora a linguagem C não tenha sido projetada especificamente para programação estatística, ela se adapta muito bem a esta tarefa. Ela até oferece alguma flexibilidade não encontrada em linguagens administrativas mais comuns como COBOL ou BASIC. Uma vantagem sobre Cobol é a velocidade e facilidade com que os programas em C podem interfacear-se a funções gráficas do sistema para produzir diagramas e gráficos de dados. Além disto, dependendo do seu compilador, as rotinas matemáticas em C podem ser muito mais rápidas do que aquelas no interpretador BASIC.

Este capítulo enfoca vários conceitos de estatística, incluindo

- Média
- Mediana
- Desvio-Padrão
- Equação de Regressão *(line of best fit)*
- Coeficiente de Correlação

Ele também explora algumas técnicas gráficas simples.

AMOSTRAGENS, POPULAÇÕES, DISTRIBUIÇÕES E VARIÁVEIS

Antes de você usar estatísticas, é preciso entender alguns pontos-chaves. Informação estatística é derivada primeiramente de uma *amostra* de dados de pontos específicos e depois fazendo uma generalização dessa mostra. Cada amostra vem de uma *população,* que consiste em todos os resultados possíveis para a situação em estudo. Por exemplo, se você quisesse medir a produção de uma fábrica de caixotes em um ano mas usando apenas os índices de produção das quartas-feiras e generalizar a partir desses, então consistiria em um valor anual de índices de quartas-feiras levantados de uma vasta população de cada resultado diário no ano.

É possível a amostra igualar-se à população se ela for exaustiva. No caso da fábrica de caixotes a sua amostra se igualaria à população se você usasse os índices reais – cinco dias por semana no ano – quando a amostra é menor que a população, existe sempre a possibilidade de erro; no entanto, em muitos casos você pode determinar a probabilidade para esse erro. Este capítulo assume que a amostra seja igual à população, e assim não explica o problema de erro da amostra.

Para projeções de eleições e pesquisas de opinião, uma amostra proporcionalmente pequena é usada para projetar informações para a população como um todo. Por exemplo, você pode usar informação estatística sobre as ações Picco Mezzolo para fazer uma dedução (inferir) sobre o mercado de ações em geral. É claro que a validade dessas conclusões varia muito. Em outros usos de estatística, uma amostra que se iguala, ou quase, à população é usada para sumarizar um conjunto grande de números para facilitar o manuseio. Por exemplo, uma comissão de educadores geralmente relata sobre a *média de notas* de uma classe, em vez da nota individual de cada aluno.

Estatísticas são afetadas pela maneira como os eventos são distribuídos na população. Das várias distribuições comuns na natureza, a mais importante (e a única

usada neste capítulo) é a *curva da distribuição normal,* ou a famosa "curva sino" como mostrado na Figura 6.1. Como sugere o gráfico na Figura 6.1, os elementos na curva de distribuição normal são encontrados em sua maioria no meio. De fato, a curva é completamente simétrica em torno de seu pico – o qual é também a média de todos os elementos. Quanto mais distante do meio em qualquer das direções da curva, menos elementos são lá encontrados.

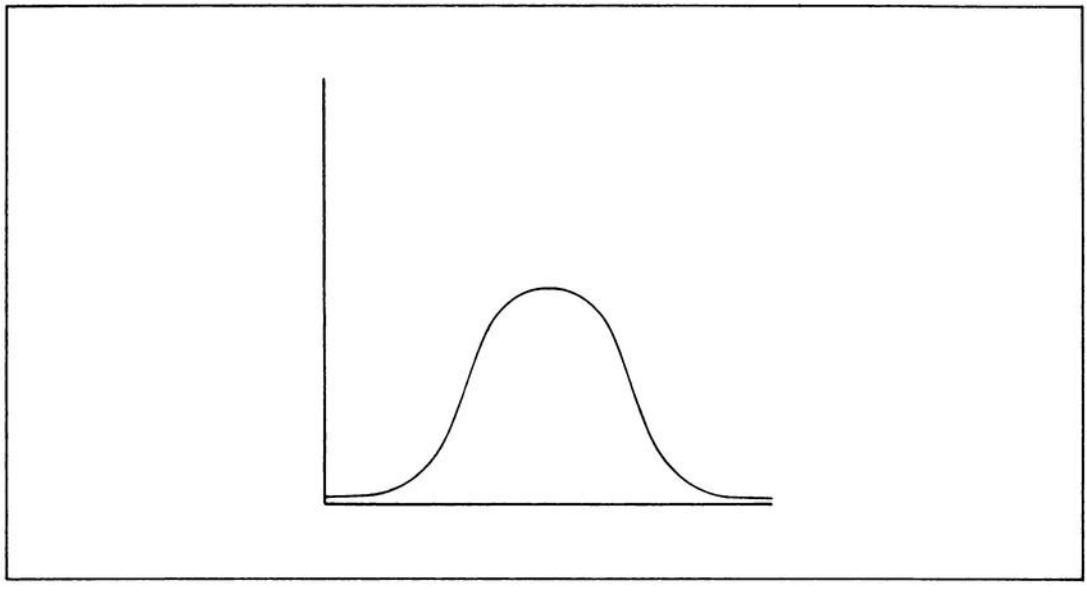


Figura 6.1. Curva de distribuição normal.

Em qualquer processo estatístico existe sempre uma *variável independente,* que é o número em questão, e uma *variável dependente,* que é o fator que determina a variável independente. Este capítulo usa *tempo* – a largura do passo incremental de ocorrência de eventos – para a variável dependente. Por exemplo, ao observar uma carteira de investimentos você deve querer ver o movimento das ações em bases diárias. Assim, você estaria interado com o movimento de preço de ações sobre um dado período de tempo, não com a data real de calendário de cada preço.

Através deste capítulo, funções estatísticas individuais serão desenvolvidas e então unidas em um programa simples dirigido por menu. Você pode usar este programa para efetuar uma grande variedade de análises estatísticas, bem como dispor informação na tela.

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

A ESTATÍSTICA BÁSICA

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

A MÉDIA

A média, ou a média aritmética, é o valor mais comum de toda estatística. Este simples valor numérico pode ser usado para representar um conjunto de dados – a média pode ser chamada de "centro de gravidade" do conjunto. Para computar a média, todos elementos na amostra são somados e o resultado é dividido pelo número total de elementos. Por exemplo, a soma do conjunto

1 2 3 4 5 6 7 8 9 10

é igual a 55. Quando este número é dividido pelo número de elementos da amostra, que é 10, a média é 5.5.

A fórmula geral para o cálculo da média é

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

ou

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

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

Como as funções estatísticas são desenvolvidas em C, você poderia assumir que todos os dados são armazenados em um vetor de números de ponto flutuante e que o número de elementos da amostra é conhecido. A seguinte função computa a média de um vetor com *num* números de ponto flutuantes e retorna a média em ponto flutuante:

```
float mean(data,num)
float *data;
int num;
{
        int t;
        float avg;

        avg=0;
        for(t=0;t(num;++t)
                avg+=data[t];
        avg/=num;

        return avg;
}
```

Por exemplo, se você chamasse **mean( )** com um vetor de 10 elementos que contém os números de 1 a 10, então **mean( )** retornaria o resultado 5.5.

A MEDIANA

A mediana de uma amostra é o valor médio baseado em ordem de magnitude. Por exemplo, no conjunto amostrado

1 2 3 4 5 6 7 8 9

5 é a mediana porque está no meio. No conjunto

1 2 3 4 5 6 7 8 9 10

você poderia usar ou 5 ou 6 como mediana. Em uma amostra bem ordenada, que tem uma distribuição normal, a mediana e a média são muito similares. No entanto, quando a amostra afasta-se da curva normal de distribuição, a diferença entre a mediana e a média aumenta. Calcular a média de uma amostra é tão simples quanto ordenar uma amostra em ordem ascendente e então selecionar o elemento mediano, que é indexado por *N*/2.

A função **median( )** aqui apresentada retorna o valor do elemento mediano de uma amostra. Uma versão modificada do *Quicksort,* desenvolvido no Capítulo 2, é usada para ordenar um vetor de dados.

```
float median(data,num)
float *data;
int num;
{
        register int t;
        float dtemp[MAX];

       for(t=0;t<num;++t)  dtemp[t]=data[t];
                           /* copia dados p/ ordenacao */
        quick(dtemp,num); /* oredena dados em ordem ascenderte */
        return dtemp[num/2];
}

quick(item,count)
float *item;
int count;
{

        qs(item,0,count-1);

}

qs(item,left,right)
float *item;
int left,right;
{
        register int i,j;
        float x,y;
        i=left; j=right;
        x=item[(left+right)/2];
        do  {
                while(item[i]<x && i<right) i++;
                while(x<item[j] && j>left) j--;

                if(i<=j)  {
                        y=item[i];
                        item[i]=item[j];
                        item[j]=y;
                        i++;  j--;
                }
        }  while(i<=j);

        if(left<j)  qs(item,left,j);
        if(i<right)  qs(item,i,right);
}
```

A MODA

A moda de uma amostra é um valor do elemento de ocorrência mais freqüente.

Por exemplo, no conjunto

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

a moda seria 6 porque ocorre três vezes. Pode haver mais do que uma moda; por exemplo, a amostra

10 20 30 30 40 50 60 60 70

tem duas modas – 30 e 60 – porque ambos ocorrem duas vezes.

A função a seguir, **find-mode( )**, retorna a moda da amostra. (Tenha cuidado; *moda* é um nome de função comum em muitas bibliotecas de C.) Se existir mais do que uma moda, então ela retorna o último elemento encontrado.

```
float find_mode(data,num)
float *data;
int num;
{
        register int t,w;
        float md,oldmode;
        int count,oldcount;

        oldmode=0; oldcount=0;
        for(t=0;t<num;++t)  {
                md=data[t];
                count=1;
                for(w=t+1;w<num;++w)
                        if(md==data[w])  count++;
                if(count>oldcount)  {
                        oldmode=md;
                        oldcount=count;
                }
        }
        return oldmode;
}
```

UTILIZANDO A MÉDIA, A MEDIANA E A MODA

A média, a mediana e a moda dividem o mesmo propósito: fornecer um valor que é a condensação de todos os valores da amostra. Porém, cada um representa a amostra de uma maneira diferente. A média de uma amostra é geralmente o valor mais utilizado. Porque utiliza todos os valores em sua computação, a média reflete todos os elementos da amostra. A principal desvantagem da média é a sua sensibilidade a um valor extremo. Por

exemplo, numa firma imaginária, Incorporação Oliveira, o salário do dono é de US$ 100,000 por ano, enquanto o salário de cada um dos nove empregados é de US$ 10,000. O salário médio na Oliveira é de US$ 19,500, mas este valor não representa a situação real.

Em amostras como a dispersão salarial na Oliveira, em alguns casos a moda é usada em vez da média. A moda dos salários na Oliveira é US$ 10,000 – um valor que reflete mais precisamente a situação real. Porém, a moda pode ser enganosa. Considere uma companhia que fabrica carros em cinco cores diferentes. Em uma certa semana, eles fabricaram

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

aqui a moda do exemplo seria preto, pois 200 carros pretos foram fabricados, mais do que qualquer outra cor. Porém, seria falso pensar que a companhia de carros fabrica principalmente carros pretos.

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

1 2 3 4 5 6 7 8 9 10

então a mediana é 5 ou 6, e a média é 5.5. Assim, a mediana e a média são similares neste caso. Porém, na amostra

1 1 1 1 5 100 100 100 100

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

Em certas circunstâncias, nem a média, a moda, ou a mediana podem sugerir um valor significativo. Isto leva a dois dos mais importantes valores estatísticos – a *variança* e o *desvio-padrão*.

VARIANÇA E DESVIO-PADRÃO

Embora o valor numérico sumário (tais como a média e a mediana) seja

conveniente, ele pode facilmente ser enganoso. Falando um pouco a respeito deste problema, você pode ver que a causa da dificuldade não é o número em si, mas o fato de não conter nenhuma informação sobre as variações dos dados. Por exemplo, na amostra

1 1 1 1 9 9 9 9

a média é 5; no entanto, não existe nenhum elemento na amosta que esteja próximo de 5. O que você provavelmente gostaria de saber é o quão perto cada elemento está da média. Se você souber em quanto os dados variam, poderá interpretar melhor a média, a mediana e a moda. Você pode encontrar a variabilidade de uma amostra computando sua variança e seu desvio-padrão.

A variança e sua raiz quadrada, o desvio-padrão, são números que lhe informam o desvio médio da média de uma amostra. Das duas, o desvio-padrão é o mais importante. Ele pode ser entendido como sendo a média da distância entre os elementos e a média da amostra. A variança é calculada como

$$V = \frac{1}{N} \sum_{i=1}^{N} (D_i - M)^2$$

onde N é o número de elementos na amostra e M é a média da amostra. É preciso quadrar a diferença entre a média e cada elemento para obter somente números positivos. Se os números não fossem elevados ao quadrado, eles somariam sempre zero.

A variança V, obtida por esta fórmula, não é muito utilizada pois é difícil de ser entendida. Porém, sua raiz quadrada, o desvio-padrão, é na verdade o valor que interessa.

O desvio-padrão é calculado achando-se em primeiro lugar a variança e depois extraindo-se sua raiz quadrada:

$$std = \sqrt{\frac{1}{N} \sum_{i=1}^{N} (D_i - M)^2}$$

onde N é o número de elementos da amostra e M a média da amostra.

Como exemplo, para a amostra seguinte

11 20 40 30 99 30 50

você calcula a variança como segue:

	D	D – M	$(D - M)^2$
	11	– 29	841
	20	– 20	400
	40	0	0
	30	– 10	100
	99	59	3481
	30	– 10	100
	50	10	100
soma	280	0	5022
média	40	0	717,42

Neste caso, a média do quadrado das diferenças é 717,42. Para obter o desvio-padrão, bastaria tirar a raiz quadrada deste número; o resultado é aproximadamente 26,78. Ao interpretar o desvio-padrão, lembre-se de que ele representa a *distância média que existe entre cada elemento da amostra e a média da amostra.*

O desvio-padrão simboliza o quanto a média é representativa da amostra. Por exemplo, se você fosse dono de uma fábrica de chocolates e seu gerente o informasse de que a média diária de barras de chocolate o mês passado foi de 2500, mas o desvio-padrão foi de 2000, você poderia concluir que sua linha de produção necessita de melhor supervisão!

Se a sua amostra segue uma distribuição normal, então mais ou menos 68% da amostra estará dentro de um desvio-padrão da média, enquanto uns 95% estará dentro de dois desvios-padrões.

A função seguinte calcula e retorna o desvio-padrão de uma dada amostra. Perceba que **sqrt()** requer um argumento do tipo **double** e retorna conversões explícitas do tipo **double.** Embora fosse possível usar um atribuidor *(cast),* muitos compiladores C não avaliam corretamente um atribuidor numa chamada de função, assim optou-se pela alternativa mais segura

```
float std_dev(data,num)
float *data;
int num;
{
        register int t;
        float std,avg;
        double temp,sqrt();

        avg=mean(data,num);     /* pega a media */
        std=0;
        for(t=0;t<num;++t)
        {
              std+=((data[t]-avg)*(data[t]-avg));
        }
        std/=num;
        temp=std;
        temp=sqrt(temp);
        std=temp;
        return std;
}
```

PLOTAGEM SIMPLES NA TELA

A vantagem em se usar gráficos em estatística é que juntos eles podem dar uma idéia clara e precisa. Um gráfico também mostra com rapidez como a amostra foi realmente distribuída e qual é a variação dos dados.

Essa discussão está limitada a gráficos de duas dimensões que usam o sistema de coordenadas X-Y. (Criação de gráficos tridimensionais é uma discussão à parte e fora do escopo deste livro.)

Existem duas formas básicas de gráficos bidimensionais: *gráficos de barras* e *gráficos de scatter*. O gráfico de barra usa barras sólidas para representar a magnitude de cada elemento, enquanto o gráfico de *scatter* usa pontos únicos para cada elemento, localizados nas suas coordenadas X-Y. A Figura 6.2 mostra um exemplo de cada tipo.

Aplica-se o diagrama de barras geralmente quando existe pouca informação, tal como o produto interno bruto dos últimos 10 anos ou a porcentagem da produção de uma fábrica em uma base mensal.

O gráfico de *scatter* é geralmente usado para dispor um grande número de pontos de dados, tal como o preço diário das ações de uma companhia durante o ano. Uma modificação do gráfico de *scatter*, unindo-se os pontos através de uma linha contínua, é útil para plotar projeções.

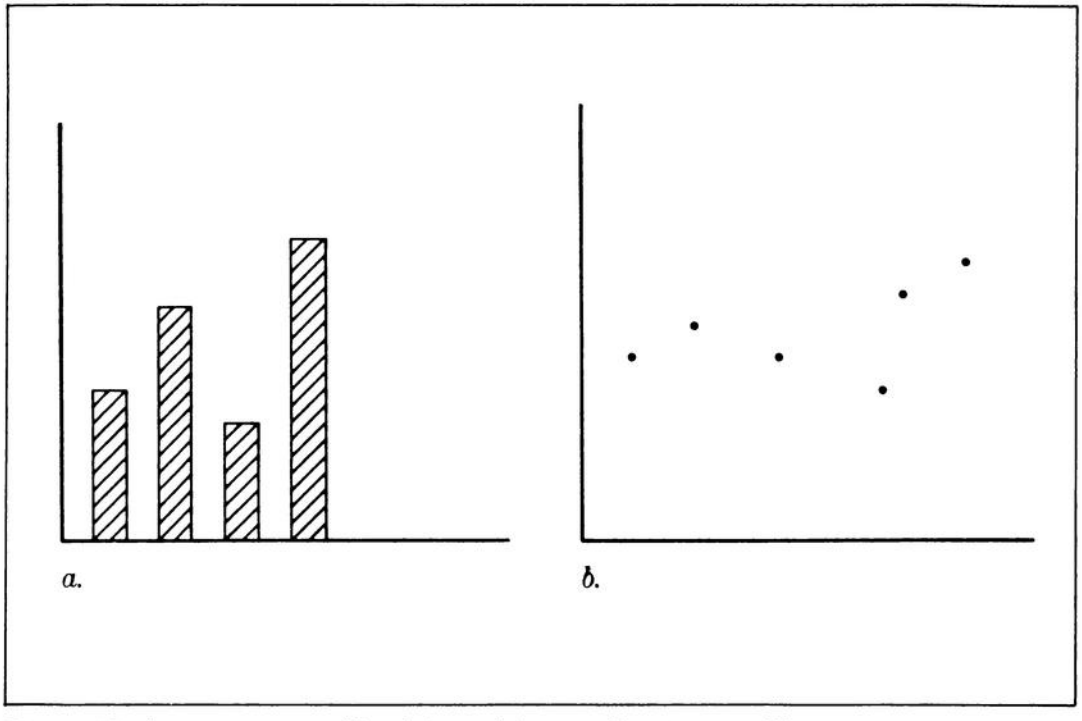


Figura 6.2. Amostragem em gráfico de barras *(a)* e em gráfico de *scatter (b)*.

Eis aqui uma função simples de representação gráfica (plotagem) que cria um gráfico de barras num IBM-PC. As funções da biblioteca C, encontradas de alguma forma na maioria dos compiladores C, permitem o uso de capacidades gráficas integradas do IBM-PC.

As funções **line( )** e **mode( )** são fornecidas pela biblioteca gráfica do compilador Aztec C; outros compiladores podem chamar estas funções por outros nomes. Se o seu compilador não as possui, você pode criá-las através de uma interface ao sistema operacional, como descrito no Capítulo 5.

```
simple_plot(data,num)
float *data;
int num;
{
        int a,t;
        mode(6);          /* modo grafico b&w 640x200 */

        scr_curs(24,0);  printf("%d",0);
        scr_curs(0,0);   printf("%d",200);
```

```
        scr_curs(24,76);  printf("%d",580);
        for(t=0;t(num;++t)  {
                a=data[t];
                if(a(0) a=0;
                line((t*10)+20,0,(t*10)+20,a);
        }
        getchar();        /* espera antes de retornar p/ o modo
                             caracter de 25x80 */
        mode(3);
}
```

No IBM-PC, o modo máximo de resolução gráfica é o modo 6, que oferece uma resolução de 640 x 200. A função **scr-curs( )** é fornecida pelo fabricante do compilador para posicionar o cursor na posição X-Y desejada. A função **line()** usada em **simple–plot** tem a forma geral

line *(start–X, start–Y, end–X, end–Y);*

onde todos os valores devem ser inteiros.

Esta rotina simples de plotagem tem uma séria limitação – ela assume que todos os dados estarão entre 0 e 199 porque os únicos números válidos que podem ser usados para chamar uma **line()** estão no intervalo entre 0 e 199. Esse compromisso é ótimo para um evento onde é improvável que seus dados elementares caiam fora desse intervalo. Para fazer a rotina de plotagem trabalhar com unidades arbitrárias você deve *normalizar* os dados antes de plotar para que os valores dos dados sejam encaixados no intervalo permitido. O processo de normalização envolve a determinação de uma razão entre o intervalo real dos dados e o intervalo físico de resolução da tela. Cada dado elementar ao ser multiplicado por essa razão produz um valor que se encaixa no intervalo da tela. A fórmula para fazer isso para o eixo Y no PC é

$$Y' = Y * \frac{200}{(max - min)}$$

onde Y' será o valor usado quando a função de plotagem for chamada. A mesma função pode ser usada para aumentar a escala quando o intervalo de dados for muito pequeno. Isto produz um gráfico que enche toda a tela.

A função seguinte **barplot( )** escala os eixos X e Y e plota um gráfico de barras

com até 580 elementos. Assume-se o tempo para o eixo X e ele é incrementado de uma unidade. Geralmente, a normalização encontra o maior e o menor valor da amostra e então calcula sua diferença. Esse número, que representa a distância entre o máximo e o mínimo, é usado para dividir a resolução da tela. No caso do IBM-PC, os números são 200 para o eixo Y e 580 para o eixo X (para ter algum espaço para as bordas). A razão é então usada para converter os dados da amostra na escala apropriada.

```
barplot(data,num)
float *data;
int num;
{
        int y,t,max,min,incr;
        float a,norm,spread;
        char s[80];

        mode(6);        /* modo grafico 640x200 b&w */
        /* acha o primeiro valor maximo
        p/ habilitar a normalizacao */
       max=getmax(data,num);
        min=getmin(data,num);
        if(min>0) min=0;
        spread=max-min;
        norm=200/spread;
        scr_curs(24,0);  printf("%d",min);
        scr_curs(0,0);  printf("%d",max);
        scr_curs(24,76);  printf("%d",num);
        for(t=1;t<24;++t)  {
                scr_curs(t,0);
                printf("-");
        }
        for(t=0;t<num;++t)  {
                a=data[t];
                a=a-min;
                a*=norm;        /* normaliza  */
                y=a;    /* tipo de conversao */
                incr=580/num;
                line((t*incr)+20+incr,0,(t*incr)+20+incr,y);
        }
        gets(s);
        mode(3);
}
```

Esta versão também imprime pequenos traços ao longo do eixo Y que representam 1/24 da diferença entre os valores mínimo e máximo. A Figura 6.3 mostra um exemplo de saída de **barplot( )** com 20 elementos. De nenhuma maneira **barplot( )** fornece todas as

características que você possa desejar, mas será satisfatório para exibir uma amostragem simples.

Com uma pequena modificação na função **barplot( )** você pode criar uma função para plotar um gráfico de *scatter*. A principal alteração modifica **line( )**, de maneira que ela plote apenas um ponto.

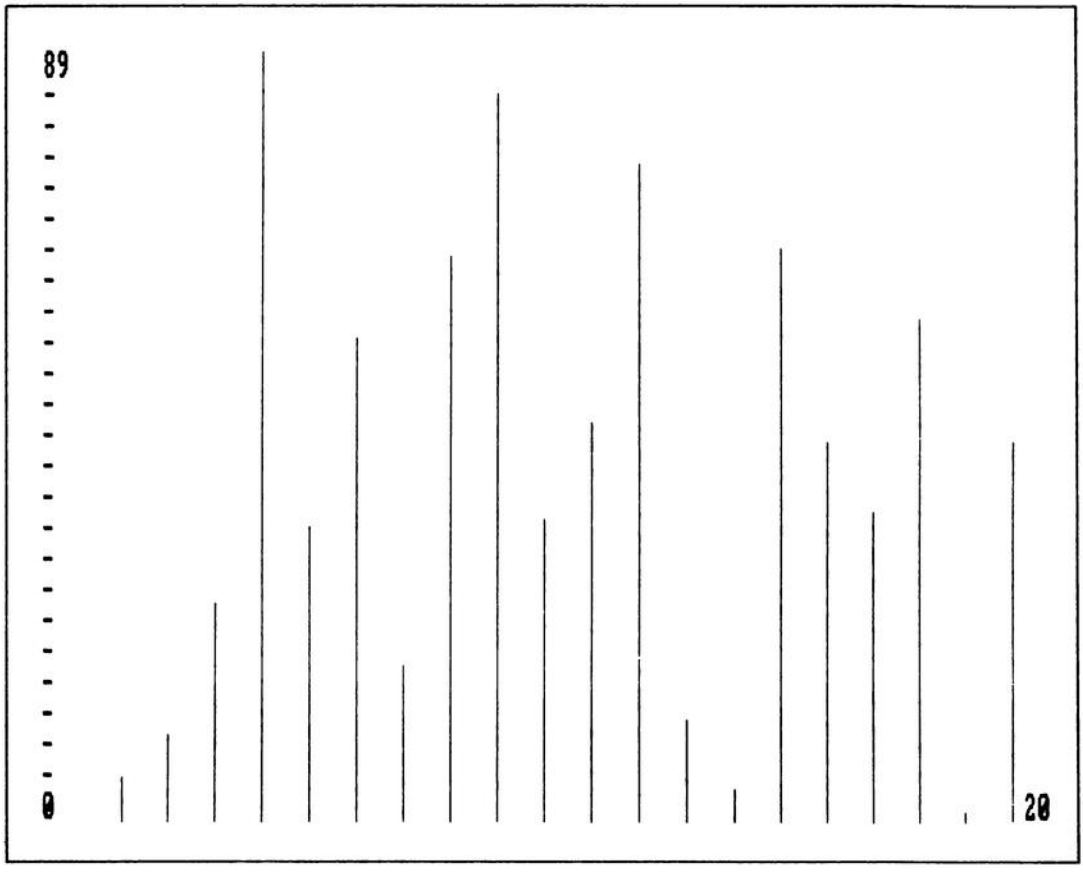


Figura 6.3. Um exemplo de gráfico de barras fornecido por **barplot()**.

Na biblioteca gráfica do compilador Aztec C, esta função tem o nome de **point( )**, mas seu compilador pode nomeá-la diferentemente. A forma geral de **point( )** é

point *(x, y),*

onde *x* e *y* são números inteiros. Aqui está a nova função **scatterplot( )**:

```
scatterplot(data,num,ymin,ymax,xmax)
float *data;
int num,ymin,ymax,xmax;
{
        int y,t,incr;
        float norm,a,spread;

        /* encontra primeiro valor maximo
        p/ habilitar a normalizacao */
        if(ymin>0) ymin=0;
        spread=ymax-ymin;
        norm=200/spread;
        scr_curs(24,0);  printf("%d",ymin);
        scr_curs(0,0);  printf("%d",ymax);
        scr_curs(24,76);  printf("%d",xmax);
        for(t=1;t<24;++t)  {
                scr_curs(t,0);
                printf("-");
        }
        incr=580/xmax;
        for(t=0;t<num;++t)  {
                a=(data[t]-ymin)*norm;  /* normalizacao */
                y=a;    /* tipo de conversao */
                x=(t*incr)+20+incr;
                point(x,y);
        }
}
```

Na **scatterplot( )** os valores mínimo e máximo dos dados são passados para a função, em vez de serem calculados pela função como em **barplot( )**. Isso permite que você "plote" um conjunto de dados múltiplos na mesma tela, sem mudar a escala. A Figura 6.4 mostra um gráfico de *scatter* de 30 elementos, produzidos por esta função.

PROJEÇÕES E A EQUAÇÃO DE REGRESSÃO

Informação estatística é freqüentemente usada para fazer "previsões" sobre o futuro. De qualquer maneira todo mundo sabe que o passado não produz necessariamente o futuro e que existem exceções para todas as regras, dados históricos ainda são usados dessa maneira, porque, muito freqüentemente, o passado e o presente tendem a continuar no futuro. Quando eles assim se comportam, você pode tentar determinar valores específicos em pontos futuros. Este processo é chamado de *projeção ou análise de tendências*.

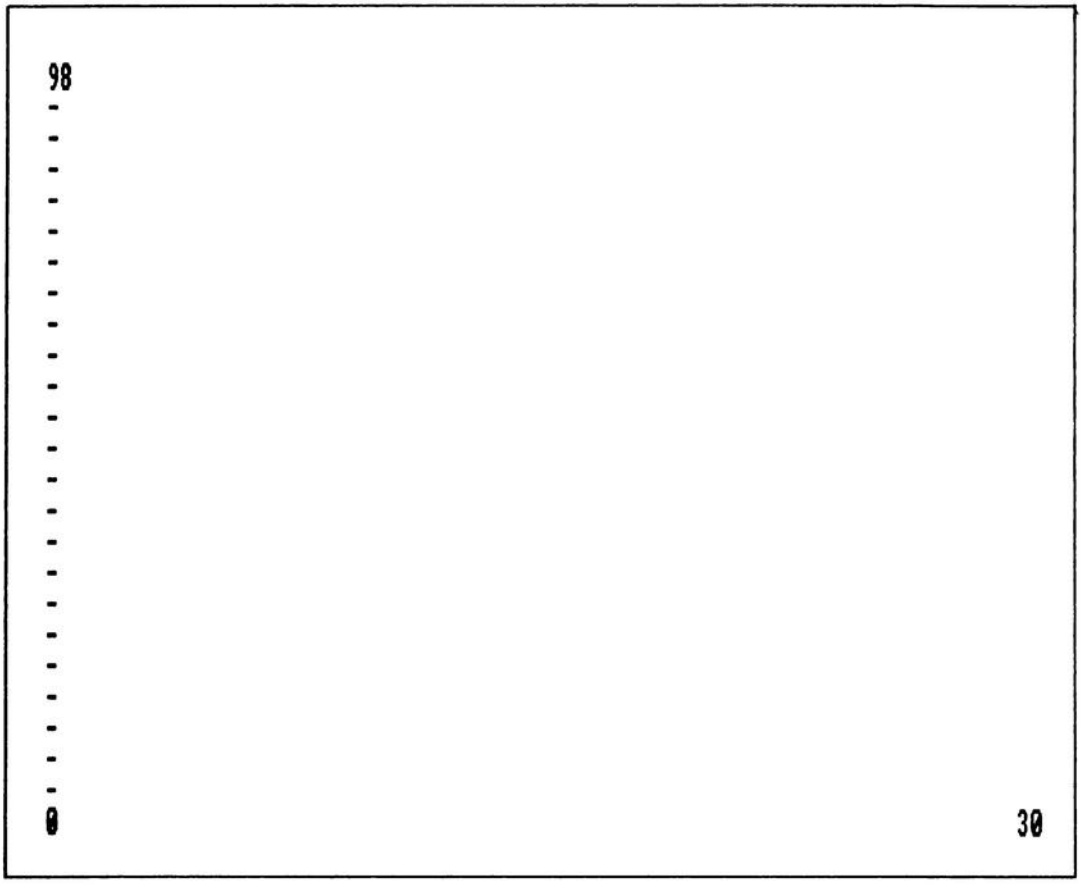


Figura 6.4. Exemplo de gráfico de *scatter* produzido por **scatterplot()**.

Por exemplo, considere um estudo fictício de 10 anos do tempo de vida médio, onde foram coletados os seguintes dados:

Ano	Tempo de Vida
1970	69
1971	70
1972	72
1973	68
1974	73
1975	71

1976	75
1977	74
1978	78
1979	77

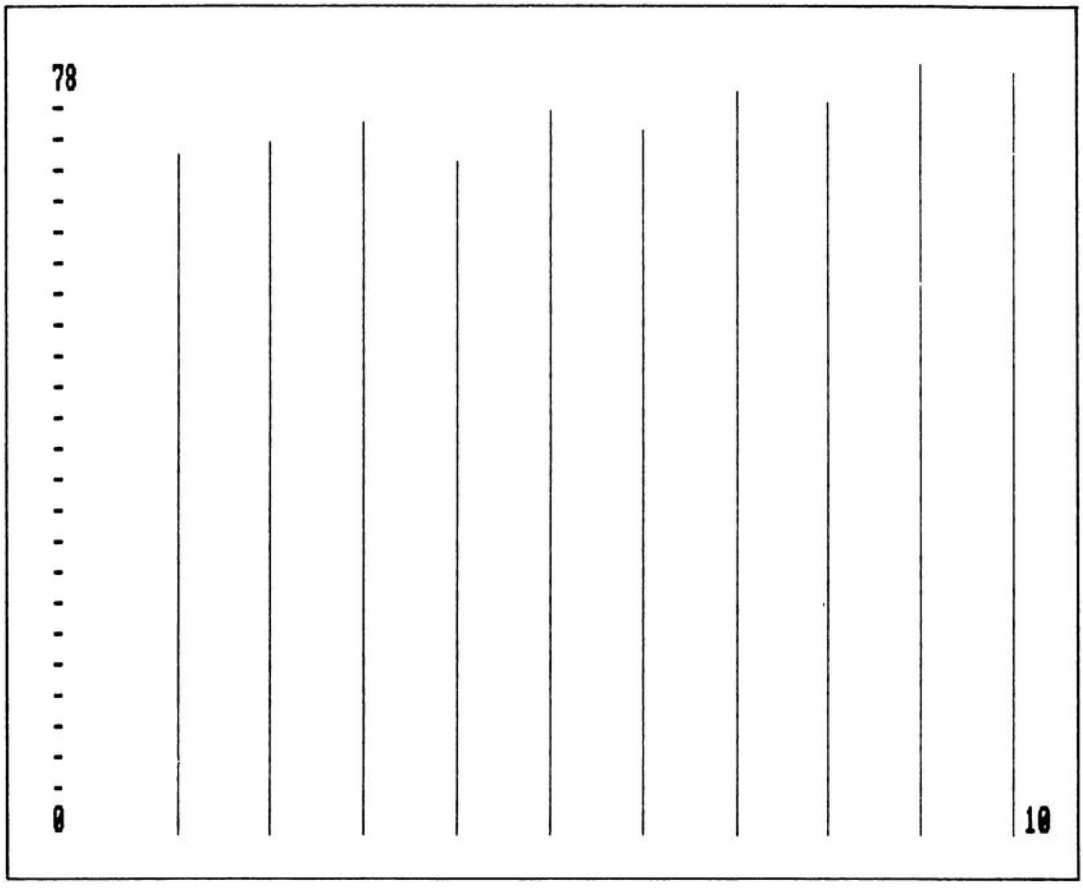


Figura 6.5. Gráfico de barras de vida estimada.

Você deve, em primeiro lugar, perguntar-se se existe uma tendência neste caso. Se houver, você irá querer saber como ela prosseguirá. Finalmente, se houver com certeza uma tendência, você desejará saber qual o tempo de vida estimado para, por exemplo, 1985.

Em primeiro lugar, observe o gráfico de barras destes dados, como mostrado na Figura 6.5. Examinando o gráfico você pode concluir que o tempo de vida está crescendo de maneira geral. Além disso, se você colocar uma régua no gráfico para tentar encaixar

os dados e desenhar uma linha que se estenda até 1985, poderá concluir que o tempo de vida médio será de 82 anos em 1985. No entanto, para sentir-se mais seguro sobre suas análises intuitivas, você provavelmente usará um método mais formal e exato para projetar o tempo de vida médio.

Dado um conjunto de dados históricos, a melhor maneira de fazer projeções é encontrar a *linha de melhor ajuste* em relação a estes dados. Isso é o que você fez com a régua. A linha de melhor ajuste mais próxima representa cada ponto dos dados e suas tendências. Embora alguns ou mesmo todos os pontos de dados reais não devam cair sobre a linha, a melhor linha os representa. A validade da linha está baseada em quão próximo dela estão os pontos de dados da amostra.

A linha em um espaço de duas dimensões tem a equação básica

$$Y = a + bX$$

onde X é uma variável dependente, Y é uma variável independente, a é o interceptador em Y, e b é a inclinação da linha. Assim, para encontrar a linha de melhor ajuste da amostra, você precisa encontrar a e b.

Qualquer método pode ser usado para determinar o valor de a e b, mas o mais comum e geralmente o melhor é o *método dos mínimos quadrados*. Ele procura minimizar a distância entre os pontos de dados reais e a linha. O método envolve dois passos: o primeiro calcula b, a inclinação da linha, e o segundo encontra a, o interceptador em Y. Para encontrar b, use a fórmula

$$b = \frac{\sum_{i=1}^{N} (X_i - M_x)(Y_i - M_y)}{\sum_{i=1}^{N} (X_i - M_x)^2}$$

onde M_x é a média das coordenadas X e M_y é a média das coordenadas Y. A demonstração desta fórmula está fora do escopo deste livro, tendo encontrado b, você pode usá-lo para calcular a, como visto a seguir:

$$a = M_y - bM_x$$

Depois de ter calculado a e b, você pode substituir qualquer valor de X e obter o valor de Y. Por exemplo, se você usar os dados do tempo de vida médio estimado, verá que a equação de regressão fica

$$Y = 67.46 + 0.95 * X$$

Assim, para saber o tempo de vida estimado para 1985, 15 anos além de 1970, você terá

$$\textit{tempo de vida estimado} = 67.46 + 0.95 * 15 \cong 82$$

No entanto, mesmo com a linha de melhor ajuste dos dados, você ainda irá querer saber com que precisão a linha corresponde aos dados. Se a linha e os dados só têm uma pequena correlação, então a linha de regressão terá pouca utilidade. Porém, se a linha adapta-se bem aos dados, então é um bom indicador. A maneira mais comum de determinar e representar a correlação entre os dados e a linha de regressão é calcular o *coeficiente de correlação*, que é a distância relativa entre cada ponto de dado na amostra e a linha. Se o coeficiente de correlação é 1, então os dados correspondem perfeitamente à linha. Um coeficiente 0 significa que não existe nenhuma correlação entre a linha e os pontos – na verdade, qualquer linha é tão boa (ou má) quanto a utilizada. A fórmula para encontrar o coeficiente de correlação é

$$Cor = \frac{\frac{1}{N}\sum_{i=1}^{N}(X_i - M_x)(Y_i - M_y)}{\sqrt{\frac{1}{N}\sum_{i=1}^{N}(X_i - M_x)^2}\ \sqrt{\frac{1}{N}\sum_{i=1}^{N}(Y_i - M_y)^2}}$$

onde M_x é a média de X e M_y a média de Y. Geralmente o valor 0.81 é considerado de grande correlação. Indica que aproximadamente 66% dos dados encaixam-se na linha de regressão. Para converter qualquer coeficiente de correlação em porcentagem, você simplesmente eleva-o ao quadrado.

Aqui está a função **regress( )**. Ela usa os métodos descritos para encontrar a equação de regressão e o coeficiente de correlação, assim como para fazer a plotagem *scatter* de ambos os dados amostrados e a linha:

```
regress(data,num)
float *data;
int num;
{
        float a,b,x_avg,y_avg,temp,temp2;
        float data2[580],cor;
        float std_dev();
        int t,min,max;
        char s[80];

        /* acha a media de y */
        y_avg=0;
        for(t=0;t<num;++t)
                y_avg+=data[t];
        y_avg/=num;

        /* acha a media de x */
        x_avg=0;
        for(t=1;t<=num;++t)
                x_avg+=t;
        x_avg/=num;

        /* achar b */
        temp=0; temp2=0;
        for(t=1;t<=num;++t)  {
                temp+=(data[t-1]-y_avg)*(t-x_avg);
                temp2+=(t-x_avg) * (t-x_avg);
        }
        b=temp/temp2;

        /* achar a */
        a=y_avg-(b*x_avg);

        /* calcular o coeficiente de correlacao */
        for(t=0;t<num;++t)  data2[t]=t+1;
        cor=temp/(num);
        cor=cor/(std_dev(data,num) * std_dev(data2,num));

        printf("equacao da regressao e: Y = %f + %f * X\n",a,b);
        printf("coeficiente de correlacao: %f\n",cor);
        printf("mostra os pontos dos dados e a linha de regressao?
        (s/n) ");
        gets(s);
        if(toupper(*s)=='N')  return;
        mode(6);          /* modo grafico 640x200 b&w */
        /* mostra os pontos dos dados e a linha de regressao */
        for(t=0;t<num*2;++t)
                data2[t]=a+(b*(t+1));
        min=getmin(data,num)*2;
        max=getmax(data,num)*2;
        scatterplot(data,num,min,max,num*2);
```

```
scatterplot(data2,num*2,min,max,num*2);
gets(s);
mode(3);
```

A disposição dos pontos para ambas as amostragens de tempo de vida e a linha de regressão é vista na Figura 6.6. Um ponto importante a salientar quando usar projeções como esta, é que o passado não prevê necessariamente o futuro – se fizesse seria ótimo!

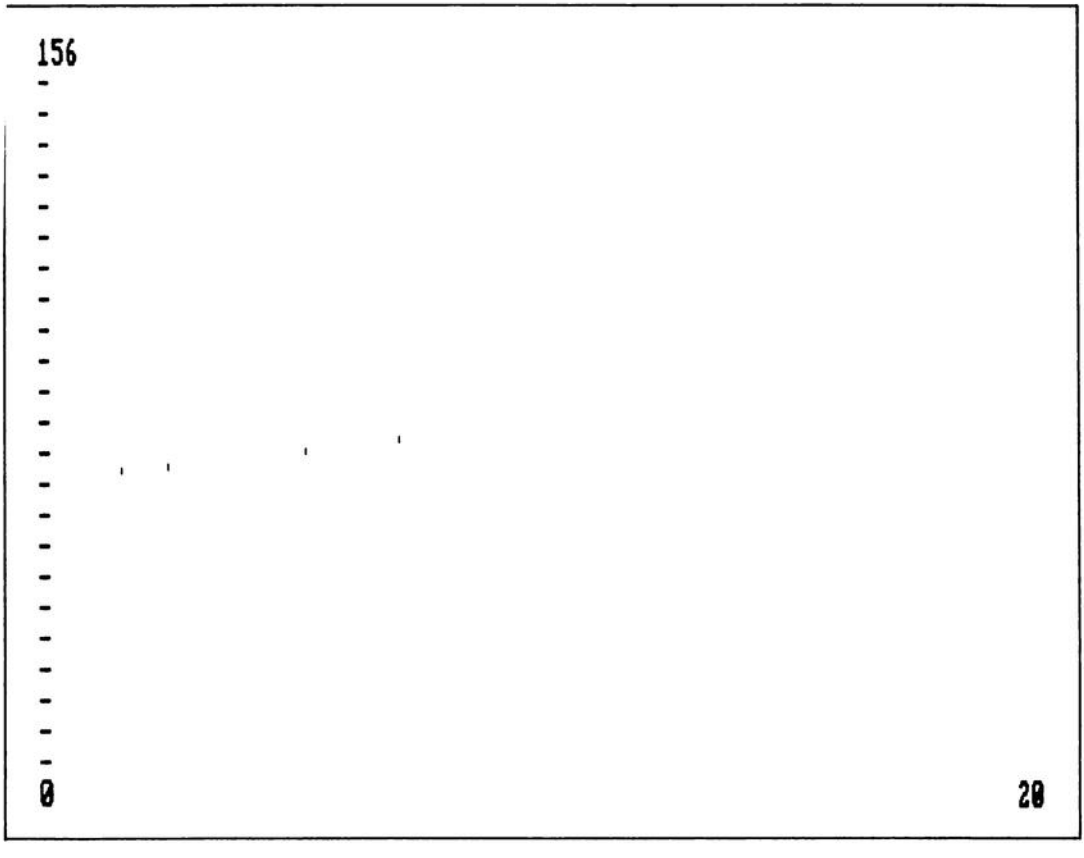


Figura 6.6. Linha de regressão para tempo de vida médio estimado.

FAZENDO UM PROGRAMA COMPLETO DE ESTATÍSTICA

Até agora, este capítulo desenvolveu muitas funções que executam cálculos estatísticos em populações de variáveis simples. Nesta seção você juntará as funções para

formar um programa completo para analisar dados, imprimir diagramas de barra ou *scatterplot* e fazer projeções. Antes de você projetar um programa completo, deve definir uma estrutura de dados para armazenar os dados de informações variáveis, assim como algumas funções de suporte necessárias.

Primeiro você precisa de um vetor para armazenar a informação amostrada. Você pode usar um vetor de dimensão única de ponto flutuante chamado **data** de tamanho **MAX. MAX** é definido de maneira que contenha a maior amostra desejada, que no caso é 100. A função **main( )** junto com a função menu de seleção **menu( )** e as funções de suporte **is-in( )** são aqui apresentadas

```
#define MAX 100
#include "stdio.h"
float mean(),std_dev(),median(),find_mode();
main()
{
        char ch;
        float data[MAX];        /* esta matriz tera os dados */
        float a,m,md,std;
        int num;        /* numero de items de dados */
        num=0;

        for(;;)  {
                ch=menu();
                switch(ch)  {
                        case 'A': num=enter_data(data);
                                  break;
                        case 'B': a=mean(data,num);
                                  std=std_dev(data,num);
                                  m=median(data,num);
                                  md=find_mode(data,num);
                                  printf("Media: %f\n",a);
                                  printf("Desvio padrao: %f\n",std);
                                  printf("Mediano: %f\n",m);
                                  printf("Moda: %f\n",md);
                                  break;
                        case 'C': regress(data,num);
                                  break;
                        case 'G': display(data,num);
                                  break;
                        case 'F': num=load(data);
                                  break;
                        case 'E': save(data,num);
                                  break;
                        case 'D': barplot(data,num);
                                  break;
                        case 'H': exit(0);
                }
```

```
        }
}

menu()
{
        char ch;
        do  {
                printf("\nA - Entre com os dados\n");
                printf("B - Estatistica basica \n");
                printf("C - Regressao linear e grafico      \n");
                printf("D - bar graph\n");
                printf("E - Salvar\n");
                printf("F - Carregar\n");
                printf("G - Mostra os dados na tela\n");
                printf("H - Sair\n\n");
                printf("\nEntre com sua escolha: ");
                ch=toupper(dos(1,0,0));
        }  while(!is_in(ch,"ABCDEFGH"));
        printf("\n");
        return ch;
}
is_in(ch,s)
char ch,*s;
{
        while(*s)  {
                if(ch==*s)  return ch;
               else s++;
        }
        return 0;
}
```

A função **is-in( )** retorna TRUE se o caractere estiver na série de caracteres e FALSE se não estiver.

Além das funções estatísticas já desenvolvidas você também necessitará de rotinas para salvar e carregar dados. A rotina **save()** também deve armazenar o número de elementos de dados, e a **load( )** deve fazer a consistência deste número.

```
save(data,num)
float *data;
int num;
{
        FILE *fp;
        int t;
        char s[80];
        printf("entre com o nome do arquivo: ");
        gets(s);
```

```
        if((fp=fopen(s,"w"))==0)  {
                printf("arquivo inacessivel\n");
                exit(1);
        }
        putw(num,fp);

        for(t=0;t(num;++t)  fprintf(fp,"%f ",data[t]);

        fclose(fp);
}

load(data)
float *data;
{
        FILE *fp;
        int t,num;
        char s[80];

        printf("entre com o nome do arquivo: ");
        gets(s);
        if((fp=fopen(s,"r"))==0)  {
                printf("arquivo inacessivel\n ");
                exit(1);
        }
       num=getw(fp);
        for(t=0;t(num;++t) fscanf(fp,"%f",&data[t]);
        fclose(fp);
        return num;
}
```

Para sua conveniência eis um programa inteiro:

```
#define MAX 100
#include "stdio.h"
float mean(),std_dev(),median(),find_mode();

main()
{
        char ch;
        float data[MAX];        /* esta matriz tera os dados */
        float a,m,md,std;
        int num;        /* numero de items de dados */
        num=0;

        for(;;)  {
                ch=menu();
                switch(ch)  {
                        case 'A': num=enter_data(data);
                                  break;
```

```
                        case 'B': a=mean(data,num);
                                  std=std_dev(data,num);
                                  m=median(data,num);
                                  md=find_mode(data,num);
                                  printf("Media: %f\n",a);
                                  printf("Desvio padrao: %f\n",std);
                                  printf("Mediano: %f\n",m);
                                  printf("Moda: %f\n",md);
                                  break;
                        case 'C': regress(data,num);
                                  break;
                        case 'G': display(data,num);
                                  break;
                        case 'F': num=load(data);
                                  break;
                        case 'E': save(data,num);
                                  break;
                        case 'D': barplot(data,num);
                                  break;
                        case 'H': exit(0);
                    }
        }

menu()

        char ch;
        do  {
                printf("\nA - Entre com os dados\n");
                printf("B - Estatistica basica \n");
                printf("C - Regressao linear e grafico     \n");
                printf("D - bar graph\n");
                printf("E - Salvar\n");
                printf("F - Carregar\n");
                printf("G - Mostra os dados na tela\n");
                printf("H - Sair\n\n");
                printf("\nEntre com sua escolha: ");
                ch=toupper(dos(1,0,0));
        }  while(!is_in(ch,"ABCDEFGH"));
        printf("\n");
        return ch;
}

is_in(ch,s)
char ch,*s;



        while(*s)  {
                if(ch==*s)  return ch;
                else s++;
```

```
        }
        return 0;
}

display(data,num)
float *data;
int num;
{
        int t;

        for(t=0;t<num;++t)
                printf("item %d: %f\n",t+1,data[t]);

        printf("\n");
}

enter_data(data)
float *data;
{
        int t,num;

        printf("numero de itens?: ");
        num=get_num();

        for(t=0;t<num;++t) {
                printf("entre com o item %d: ",t+1);
                scanf("%f",&data[t]);
        }
        return num;
}

float mean(data,num)
float *data;
int num;
{
        int t;
        float avg;

        avg=0;
        for(t=0;t<num;++t)
                avg+=data[t];
        avg/=num;

        return avg;
}

float std_dev(data,num)
float *data;
int num;
{
        register int t;
        float std,avg;
```

```
        double temp,sqrt();

        avg=mean(data,num);     /* pega a media */
        std=0;
        for(t=0;t<num;++t)
        {
                std+=((data[t]-avg)*(data[t]-avg));

        }
        std/=num;
        temp=std;
        temp=sqrt(temp);
        std=temp;
        return std;
}

float median(data,num)
float *data;
int num;
{
        register int t;
        float dtemp[MAX];

        for(t=0;t<num;++t)
        dtemp[t]=data[t];
        /* copia dados p/ ordenacao */
        quick(dtemp,num);
        /* ordena dados em ordem ascendente */
        return dtemp[num/2];
}

float find_mode(data,num)
float *data;
int num;
{
        register int t,w;
        float md,oldmode;
        int count,oldcount;

        oldmode=0; oldcount=0;
        for(t=0;t<num;++t)  {
                md=data[t];
                count=1;
                for(w=t+1;w<num;++w)
                        if(md==data[w])  count++;
                if(count>oldcount)  {
                        oldmode=md;
                        oldcount=count;
                }
        }
        return oldmode;
```

```
}

regress(data,num)
float *data;
int num;
{
        float a,b,x_avg,y_avg,temp,temp2;
        float data2[580],cor;
        float std_dev();
        int t,min,max;
        char s[80];

        /* acha a media de y */
        y_avg=0;
        for(t=0;t<num;++t)
                y_avg+=data[t];
        y_avg/=num;

        /* acha a media de x */
        x_avg=0;
        for(t=1;t<=num;++t)
                x_avg+=t;
        x_avg/=num;

        /* achar b */
        temp=0; temp2=0;
        for(t=1;t<=num;++t)  {
                temp+=(data[t-1]-y_avg)*(t-x_avg);
                temp2+=(t-x_avg) * (t-x_avg);
        }
        b=temp/temp2;

        /* achar a */
        a=y_avg-(b*x_avg);

        /* calcular o coeficiente de correlacao */
        for(t=0;t<num;++t)  data2[t]=t+1;
        cor=temp/(num);
        cor=cor/(std_dev(data,num) * std_dev(data2,num));

        printf("equacao da regressao e: Y = %f + %f * X\n",a,b);
        printf("coeficiente de correlacao: %f\n",cor);
        printf("mostra os pontos dos dados
        e a linha de regressao? (s/n) ");
        gets(s);
        if(toupper(*s)=='N')  return;
       mode(6);         /* modo grafico 640x200 b&w */
        /* mostra os pontos dos dados e a linha de regressao */
        for(t=0;t<num*2;++t)
                data2[t]=a+(b*(t+1));
        min=getmin(data,num)*2;
        max=getmax(data,num)*2;
```

```
        scatterplot(data,num,min,max,num*2);
        scatterplot(data2,num*2,min,max,num*2);
        gets(s);
        mode(3);
}

barplot(data,num)
float *data;
int num;
{
        int y,t,max,min,incr;
        float a,norm,spread;
        char s[80];

        mode(6);        /* modo grafico 640x200 b&w */
        /* acha o primeiro valor maximo
        p/ habilitar a normalizacao */
        max=getmax(data,num);
        min=getmin(data,num);
        if(min>0) min=0;
        spread=max-min;
        norm=200/spread;
        scr_curs(24,0);  printf("%d",min);
        scr_curs(0,0);  printf("%d",max);
        scr_curs(24,76);  printf("%d",num);
        for(t=1;t<24;++t)  {
                scr_curs(t,0);
                printf("-");
        }
        for(t=0;t<num;++t)  {
                a=data[t];
                a=a-min;
                a*=norm;        /* normaliza  */
                y=a;    /* tipo de conversao */
                incr=580/num;
                line((t*incr)+20+incr,0,(t*incr)+20+incr,y);

        }
        gets(s);
        mode(3);
}

scatterplot(data,num,ymin,ymax,xmax)
float *data;
int num,ymin,ymax,xmax;
{
        int y,t,incr;
        float norm,a,spread;

        /* encontra primeiro valor maximo
        p/ habilitar a normalizacao */
        if(ymin>0) ymin=0;
```

```
        spread=ymax-ymin;
         norm=200/spread;
         scr_curs(24,0);  printf("%d",ymin);
         scr_curs(0,0);  printf("%d",ymax);
         scr_curs(24,76);  printf("%d",xmax);
         for(t=1;t(24;++t)  {
                 scr_curs(t,0);
                 printf("-");
         }
         incr=580/xmax;
         for(t=0;t(num;++t)  {
                 a=data[t];
                 a=a-ymin;
                 a*=norm;          /* normalizacao */
                 y=a;     /* tipo de conversao */
                 point((t*incr)+20+incr,y);
         }
}

getmax(data,num)
float *data;
int num;
{
         int t,max;

         for(max=data[0],t=1;t(num;++t)
                 if(data[t]>max)  max=data[t];
         return max;
}

getmin(data,num)
float *data;
int num;
{
         int t,min;

         for(min=data[0],t=1;t(num;++t)
                 if(data[t](min)  min=data[t];
         return min;
}

save(data,num)
float *data;
int num;
{
         FILE *fp;
         int t;
         char s[80];
         printf("entre com o nome do arquivo: ");
         gets(s);

         if((fp=fopen(s,"w"))==0)  {
```

```
                printf("arquivo inacessivel\n");
                exit(1);
        }
       putw(num,fp);

        for(t=0;t(num;++t)  fprintf(fp,"%f ",data[t]);

        fclose(fp);
}

load(data)
float *data;
{
        FILE *fp;
        int t,num;
        char s[80];

        printf("entre com o nome do arquivo: ");
        gets(s);
        if((fp=fopen(s,"r"))==0)  {
                printf("arquivo inacessivel\n ");
                exit(1);
        }

        num=getw(fp);
        for(t=0;t(num;++t) fscanf(fp,"%f",&data[t]);
        fclose(fp);
        return num;
}

get_num()
{
        char s[80];
        gets(s);
        return(atoi(s));
}

quick(item,count)
float *item;
int count;
{

        qs(item,0,count-1);

}

qs(item,left,right)
float *item;
int left,right;
{
        register int i,j;
        float x,y;
```

```
        i=left; j=right;
        x=item[(left+right)/2];
        do  {
                while(item[i]<x && i<right) i++;
                while(x<item[j] && j>left) j--;
                if(i<=j)  {
                        y=item[i];
                        item[i]=item[j];
                        item[j]=y;
                        i++;  j--;
                }
        }  while(i<=j);
        if(left<j)  qs(item,left,j);
        if(i<right)  qs(item,i,right);
}
```

USANDO UM PROGRAMA ESTATÍSTICO

Para dar-lhe uma idéia de como você pode utilizar o programa desenvolvido neste capítulo, eis aqui o exemplo de uma análise simples de mercado de ações para a Incorporação Oliveira. Como investidor, você estará tentando decidir se esta é uma boa hora para investir na Oliveira, comprando ações; se você vendesse a termo *(short)* (processo de vender ações que você não possui, esperando uma queda de preço rápida para poder comprá-las mais tarde por um preço menor); ou se você deve investir de outra maneira.

Nos últimos 24 meses, os preços das ações da Oliveira foram os seguintes

Mês	Preço da Ação
1	10
2	10
3	11
4	9
5	8
6	8
7	9
8	10
9	10
10	13
11	11
12	11

Mês	Preço da Ação
13	11
14	11
15	12
16	13
17	14
18	16
19	17
20	15
21	15
22	16
23	14
24	16

Você deve primeiro saber se as ações da Oliveira estabelecem alguma tendência. Depois de inserir os valores, você encontra as seguintes bases estatísticas:

Média:	12.08
Desvio-padrão:	2.68
Mediana:	11
Moda:	11

A seguir você deve plotar um gráfico de barras dos preços das ações, como mostra a Figura 6.7.

O gráfico de barras parece mostrar a existência de uma tendência, mas é melhor executar uma análise de regressão formal. A equação de regressão é

$$Y = 7.90 + 0.33 * X$$

com um coeficiente de correlação de 0.86 ou aproximadamente 74%.

Isso é bom – de fato, uma tendência clara. Imprimindo o gráfico de *scatter*, como mostrado na Figura 6.8, faz com que este crescimento fique bem evidente. Tais resultados podem fazer com que o investidor jogue longe suas preocupações e compre 1000 *shares* o mais rápido possível.

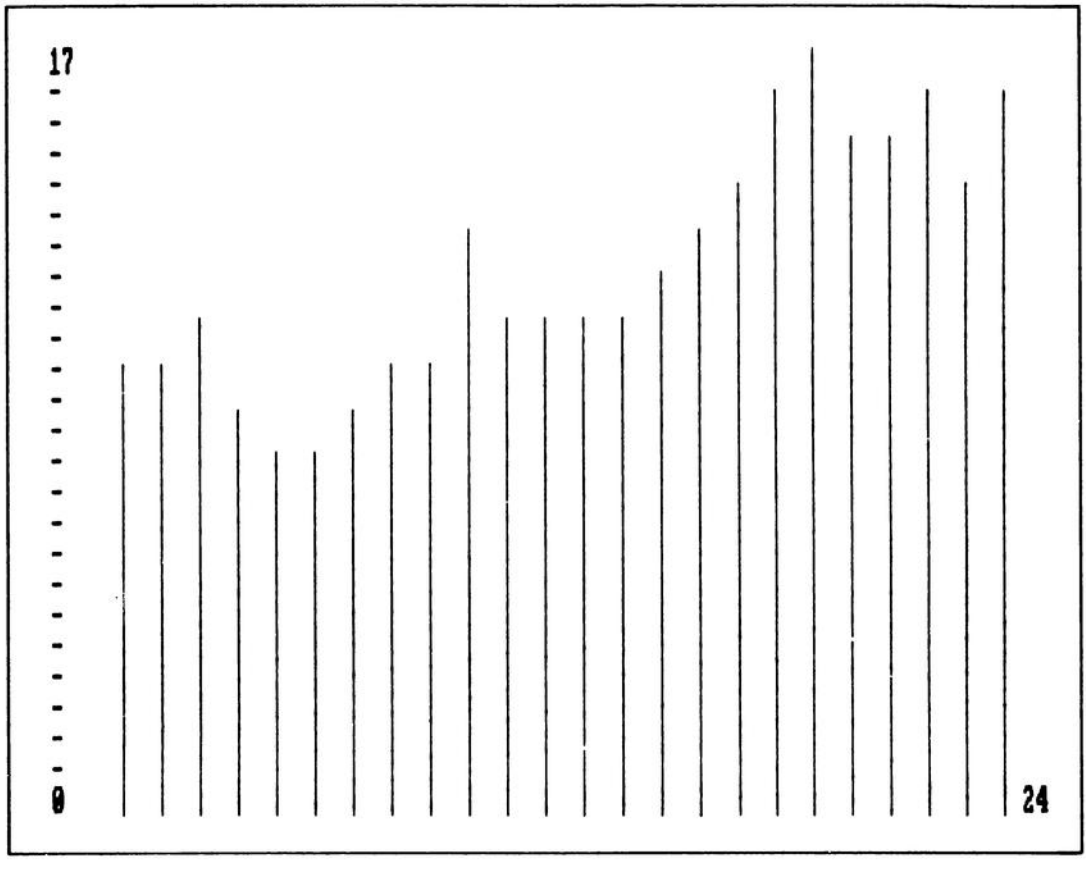


Figura 6.7. Diagrama de barras dos preços das ações da Oliveira.

CONCLUSÕES FINAIS

O uso correto da análise estatística requer uma compreensão geral de como os resultados são obtidos e seu significado real. Como no exemplo da Oliveira, é fácil esquecer que os acontecimentos passados não podem garantir em certas circunstâncias que o resultado final seja radicalmente afetado. A dependência cega nas evidências estatísticas pode causar resultados desagradáveis.

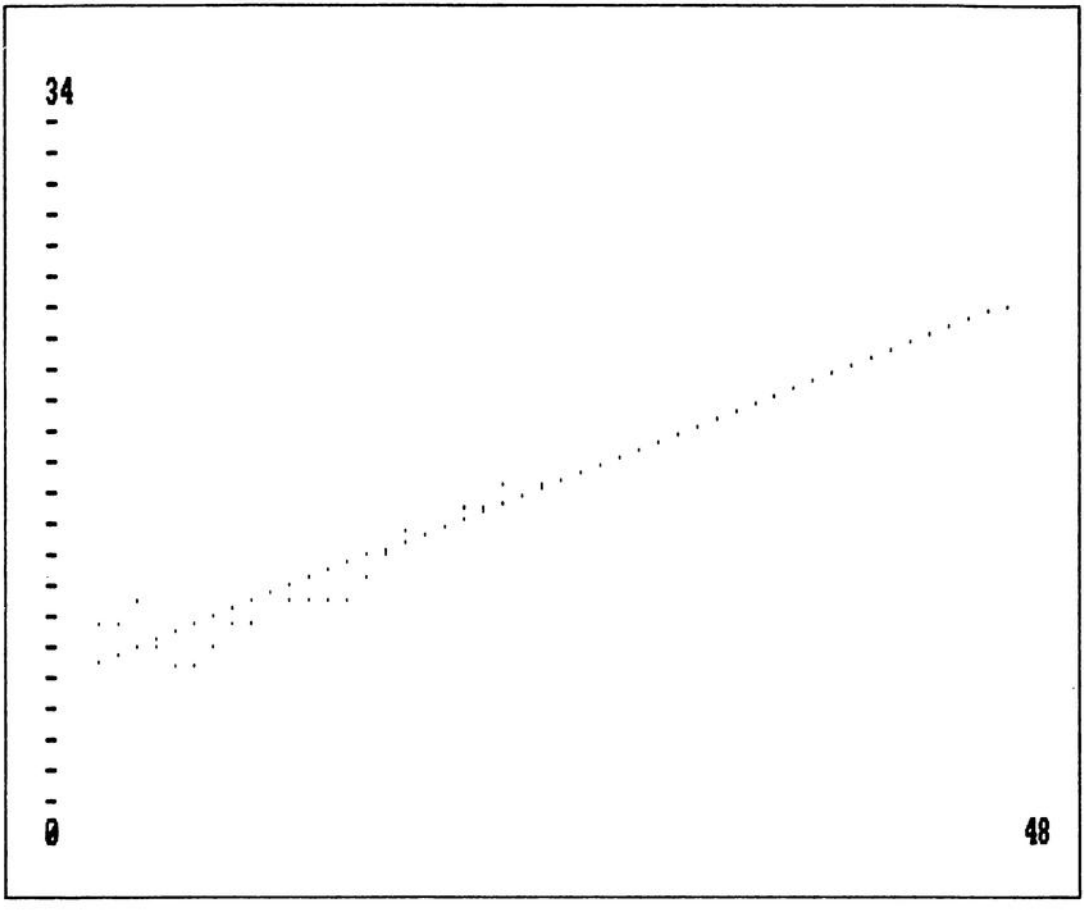


Figura 6.8. Gráfico de *scatter* das ações da Oliveira com linha de regressão.

Capítulo 7

ENCRIPTAÇÃO E COMPACTAÇÃO

Pessoas que gostam de computadores e programação também gostam de brincar com códigos e cifragens. Talvez a razão disto seja porque toda codificação envolve algoritmos, justamente o que os programas fazem. Ou talvez essas pessoas tenham uma afinidade com coisas criptografadas que a maioria das pessoas não consegue entender. Todos os programadores parecem experimentar uma grande satisfação quando um não-programador olha para uma listagem de programação e diz alguma coisa como, "Meu Deus, isto parece mesmo complicado!" Afinal de contas, o ato de escrever um programa é chamado "codificar".

Associado ao tópico de criptografia está a *compressão de dados*. Compressão de dados é compactar a informação em um espaço menor do que realmente é usado. Como a compressão de dados tem um papel importante na encriptação e usa muitos dos mesmos princípios, está incluída neste capítulo.

Criptografia baseada em computadores é importante por duas razões principais. A mais óbvia é a necessidade de manter dados confidenciais seguros em sistemas compartilhados. Embora proteção através de senha seja adequada para muitas situações, arquivos confidenciais são rotineiramente codificados para oferecer um nível mais alto de proteção.

A segunda razão é que códigos de computação são usados em transmissão de dados. Os códigos são usados não apenas para coisas tais como informação secreta do governo, mas também por alguns locutores para proteger suas transmissões via satélite. Como esses processos de codificação são muito complexos, eles são geralmente feitos por computador.

Compressão de dados é normalmente usada para aumentar a capacidade de

armazenamento de vários dispositivos. Embora o custo dos dispositivos tenha caído vertiginosamente nos últimos anos, ainda existe a necessidade de armazenar mais informação em áreas menores.

UM PEQUENO HISTÓRICO DA CRIPTOGRAFIA

Embora ninguém saiba quando começou a escrita secreta, um dos exemplos mais antigos é um tablete cuneiforme feito por volta de 1500 a.C. Ele contém uma fórmula codificada para fazer cobertura vitrificada em cerâmica. Os gregos e espartanos usavam códigos em 475 a.C., e a classe alta romana freqüentemente usava cifras simples durante o reino de Júlio César. Durante a Idade Média, o interesse pela criptografia (bem como muitas outras áreas intelectuais) diminuiu, exceto entre os monges, que a usavam ocasionalmente. Com a vinda do renascimento italiano a arte da criptografia novamente floresceu. Na época de Luis XIV da França, um código baseado em 587 chaves randomicamente selecionadas era usado em mensagens governamentais. Por volta de 1800, dois fatores ajudaram no desenvolvimento da criptografia. Primeiro eram as estórias de Edgar Allan Poe, tais como *The Gold Bug,* que apresentava mensagens em código e excitava a imaginação dos leitores. O segundo foi a invenção do telégrafo e do código Morse. O código Morse foi a primeira representação binária (pontos e traços) do alfabeto que teve grande aplicação.

Ao chegar a I Guerra Mundial, várias nações construíram "máquinas de codificação" mecânicas que permitiam facilmente codificar e decodificar textos usando cifragens sofisticadas e complexas. Aqui, a estória da criptografia dá uma leve virada para a estória da decifragem de códigos. Antes da utilização de dispositivos mecânicos para codificar e decodificar mensagens, cifragens complexas eram raramente usadas por causa do tempo e do trabalho necessário para codificar e decodificar. Assim, a maioria dos códigos era quebrada num período relativamente curto de tempo.

Porém, a arte de quebrar códigos tornou-se muito mais difícil quando as máquinas de codificação foram usadas. Embora modernos computadores possam facilmente quebrar esses códigos, nem computadores poderiam diminuir o incrível talento de Herbert Yardley, ainda considerado o maior "decifrador de códigos" de todos os tempos. Ele não só quebrou o código diplomático dos EUA, em 1915, em suas horas de folga, como também quebrou o código diplomático japonês em 1922 – sem nem mesmo saber falar japonês! Ele conseguiu isso usando tabelas de freqüência da língua japonesa.

Com a II Guerra Mundial, o principal método usado para decifrar códigos era roubar a máquina de codificação do inimigo, eliminando assim o processo tedioso (se

intelectualmente satisfatório) de decifrar os códigos. Por sinal, a aquisição de uma máquina de codificação alemã pelos aliados (sem o conhecimento dos alemães) muito contribuiu para o resultado final da guerra.

Com o advento dos computadores – especialmente os multiusuários – códigos inquebráveis e seguros têm se tornado ainda mais necessários. Não só arquivos ocasionalmente precisam ser mantidos em sigilo mas o acesso ao computador em si deve também ser gerenciado e regulado. Vários métodos de encriptação de arquivos de dados têm sido desenvolvidos e o algoritmo DES (Data Encryption Standart), aceito pelo National Bureau of Standarts, é geralmente considerado seguro em relação a tentativas de decifragens. Porém, o DES é muito difícil de ser implementado e pode não servir para muitas situações.

TIPOS DE CIFRAGENS

Dos mais tradicionais métodos de codificação, existem dois tipos básicos: *substituição* e *transposição*. Uma cifragem por substituição troca um caractere por outro, mas mantém a mensagem na mesma ordem. Uma cifragem por transposição essencialmente embaralha os caracteres de uma mensagem seguindo alguma regra. Estes tipos de códigos podem ser usados em qualquer grau de complexidade desejado. O computador digital adiciona ainda uma terceira técnica de encriptação, chamada de *manipulação de bits*, que altera a representação computadorizada de dados através de um algoritmo.

Todos os três métodos podem usar uma *chave*. A chave é uma série de caracteres necessários para decodificar a mensagem. Mas não confunda a chave com o método, pois a chave por si só nunca é suficiente para decodificar – é preciso conhecer o algoritmo de encriptação também. A chave "personaliza" a mensagem codificada para que apenas aquelas pessoas que a conhecem possam decodificá-la, embora o método utilizado para codificação seja acessível.

Existem dois termos com os quais você deve se familiarizar: *plaintext* e *ciphertext*. O *plaintext* – texto pleno de uma mensagem – é o texto que você consegue ler; o *ciphertext* – texto cifrado – é a versão codificada.

Este capítulo apresenta métodos computadorizados que usam cada um dos três métodos básicos para codificar textos de arquivos. Você verá vários programas simples que codificam e decodificam textos de arquivos. Com uma exceção, todos estes programas têm as duas funções **code( )** e **decode( )**: a função **decode( )** sempre reverte o processo **code( )** usado para criar o texto cifrado.

CIFRAGEM POR SUBSTITUIÇÃO

Uma das cifragens por substituição mais simples é deslocar o alfabeto uma quantidade específica. Por exemplo, se cada letra fosse deslocada de três, então

abcdefghijklmnopqrstuvwxyz

ficaria

defghijklmnopqrstuvwxyzabc

Observe que as letras *abc* saíram do começo e foram colocadas no fim. Para codificar uma mensagem usando este método, você simplesmente substitui o alfabeto deslocado pelo alfabeto real. Por exemplo, a mensagem

encontre-me ao anoitecer

ficaria

gpeqpvtg-og cq cpqkvgegt

O programa mostrado aqui permite que você codifique qualquer texto usando qualquer deslocamento após especificar com qual letra começará o alfabeto.

```
# include "ctype.h"
# include "stdio.h"

main(argc,argv)
int argc;
char *argv[];
{
        if(argc!=5)  {
                printf("uso:entrada saida
                codificacao/decodificacao offset\n"); exit();
        }
        if(!isalpha(*argv[4]))  {
                printf("o offset tem que ser
                um caracter alfabetico\n"); exit();
        }

        if(toupper(*argv[3])=='C') code(argv[1],argv[2],*argv[4]);
        else decode(argv[1],argv[2],*argv[4]);
}
```

```
code(input,output,start)
char *input,*output;
char start;
{
        int ch;
        FILE *fp1,*fp2;

        if((fp1=fopen(input,"r"))==0)  {
                printf("arquivo inacessivel para entrada\n");
                exit();
        }

        if((fp2=fopen(output,"w"))==0)  {
                printf("arquivo inacessivel para saida\n");
                exit();
        }

        start=tolower(start);
        start=start-'a';
        do  {
                ch=getc(fp1);
                if(ch==EOF)  break;
                if(isalpha(ch))  {
                        ch+=start;
                         if(ch)'z') ch-=26;
                }
                putc(ch,fp2);
        }  while(1);
        fclose(fp1); fclose(fp2);
}

decode(input,output,start)
char *input,*output;
char start;
{
        int ch;
        FILE *fp1,*fp2;

        if((fp1=fopen(input,"r"))==0)  {
                printf("arquivo inacessivel para entrada\n");
                exit();
        }



        if((fp2=fopen(output,"w"))==0)  {
                printf("arquivo inacessivel para saida\n");
                exit();
        }

        start=tolower(start);
```

```
        start=start-'a';
        do  {
                ch=getc(fp1);
                if(ch==EOF)  break;
                if(isalpha(ch))  {
                        ch-=start;
                        if(ch<'a') ch+=26;
                }
                putc(ch,fp2);
        }  while(1);
        fclose(fp1);fclose(fp2);
}
```

Como um exemplo, você poderia usar este programa para codificar um arquivo chamado "mensagem", para colocar a versão codificada em um arquivo chamado "cmess", e para deslocar o alfabeto de duas posições. Você primeiro digitaria a linha de comando

>code mensagem cmess codificao c

Para decifrar, você digitaria

>code cmess mensagem decodificacao c

Embora cifragem por substituição baseada em um deslocamento constante engane estudantes do primeiro grau, não é apropriada para a maioria dos casos pois é muito fácil de ser decifrada. Afinal, existem apenas 26 deslocamentos possíveis e é fácil experimentar todas as possibilidades em pouco tempo. Uma alternativa para dificultar a decifragem é misturar o alfabeto em vez de simplesmente deslocar.

Outra desvantagem da cifragem por substituição simples é que ela preserva o espaçamento entre as palavras, o que facilita muito o trabalho do decifrador. Uma melhoria seria codificar os espaços. (Na verdade, toda a pontuação deverá ser codificada, mas para simplificar não faremos isto nos exemplos.) Por exemplo, você poderia mapear esta série de caracteres do alfabeto e mais o espaço, randomicamente

abcdefghijklmnopqrstuvwxyz <espaço>

e teria

qazwsxedcrfvtgbyhnujm <espaço> ikolp

Você pode estar em dúvida sobre se existe uma vantagem significativa na segurança de uma mensagem codificada usando uma versão randômica do alfabeto em vez da versão de deslocamento simples. A resposta é *sim* – existem 26 fatorial (26!) maneiras de arranjar o alfabeto; com o espaço, este número torna-se 27 fatorial (27!) maneiras. O fatorial de um número é aquele número multiplicado por todos os números inteiros menores que ele, até 1. Por exemplo, 6! é 6*5*4*3*2*1 = 720. Assim, 23! é um número bem grande.

O programa aqui apresentado é uma cifragem por substituição, melhorada, que usa um alfabeto randômico, visto anteriormente. Se você codificasse a mensagem

encontre-me ao anoitecer

usando o programa de cifragem por substituição melhorado ficaria assim

sgzbgjns-tspqbpqgbcjszsn

o que é um código definitivamente mais difícil de ser quebrado.

```
#include "ctype.h"
#include "stdio.h"

char sub[28]=     "qazwsxedcrfvtgbyhnujm ikolp";
char alphabet[28]="abcdefghijklmnopqrstuvwxyz ";

main(argc,argv)
int argc;
char *argv[];
{
        if(argc!=4)  {
                printf("uso:entrada saida
                codificacao/decodificacao\n"); exit();
        }
        if(toupper(*argv[3])=='C') code(argv[1],argv[2]);
        else decode(argv[1],argv[2]);
}

code(input,output)
char *input,*output;
{
        int ch;
        FILE *fp1,*fp2;

        if((fp1=fopen(input,"r"))==0)  {
                printf("arquivo inacessivel para entrada\n");
                exit();
```

```
        }

        if((fp2=fopen(output,"w"))==0)  {
                printf("arquivo inacessivel para saida\n");
                exit();
        }

        do  {
                ch=getc(fp1);
                if(ch==EOF)  break;
                if(isalpha(ch) || ch==' ')  {
                        ch=sub[find(alphabet,ch)];
                }
                putc(ch,fp2);
        }  while(1);
        fclose(fp1); fclose(fp2);
}


decode(input,output)
char *input,*output;
{
        int ch;
        FILE *fp1,*fp2;

        if((fp1=fopen(input,"r"))==0)  {
                printf("arquivo inacessivel para entrada\n")
                exit();
        }

        if((fp2=fopen(output,"w"))==0)  {
                printf("arquivo inacessivel para saida\n");
                exit();
        }

        do  {

                ch=getc(fp1);
               if(ch==EOF)  break;
                if(isalpha(ch)  ||  ch==' ')  {
                        ch=alphabet[find(sub,ch)];
                }
                putc(ch,fp2);
        }  while(1);
        fclose(fp1);fclose(fp2);
}


find(s,ch)
char *s;
char ch;
```

```
{
        register int t;

        for(t=0;t<28;t++)
                if(ch==s[t]) {
                        return t;
                }
}
```

Embora a decifragem seja explicada mais adiante neste capítulo, você deve saber que mesmo este código por substituição melhorada ainda pode ser quebrado facilmente, usando-se uma tabela de freqüência da língua portuguesa, na qual a informação estatística do uso de cada letra do alfabeto está gravada. Olhando a mensagem codificada do exemplo você pode deduzir que o "s" representa o "e", uma das letras mais usadas em português, que o "p" representa o espaço. Provavelmente você consegue decifrar o resto. (Você usa o mesmo processo para resolver "criptogramas" em livros de palavras cruzadas.) Além do mais, quanto maior a mensagem codificada mais fácil é quebrar o código com uma tabela de freqüência.

Para impedir o progresso de um decifrador de códigos que aplica tabelas de freqüência a uma mensagem codificada, você pode usar uma *cifragem por substituição múltipla,* na qual a letra na mensagem plena não seria necessariamente a mesma na mensagem codificada. Você pode fazer isto adicionando um segundo alfabeto randomizado, e entre ele e o primeiro alfabeto alternar cada vez que a letra for repetida. Por exemplo, o programa pode ser reescrito para usar esta segunda série:

poi uytrewqasdfghjklmnbvcxz

A cifragem por substituição múltipla vista aqui usará apenas letras do alfabeto, alternando os alfabetos randomizados após a letra ser repetida duas vezes. Ele requer que todas as mensagens tenham comprimento de apenas uma linha.

```
#include "ctype.h"
#include "stdio.h"

char sub[28]=      "qazwsxedcrfvtgbyhnujm ikolp";
char sub2[28]=     "poi uytrewqasdfghjklmnbvcxz";
char alphabet[28]="abcdefghijklmnopqrstuvwxyz ";
char count[27];

main(argc,argv)
int argc;
```

```
char *argv[];
{
        register t;

        for(t=0;t<27;++t)  count[t]=0;
        if(argc!=4)  {
                printf("uso:entrada saida
                codificacao/decodificacao\n");
                exit();
        }
        if(toupper(*argv[3])=='C') code(argv[1],argv[2]);
        else decode(argv[1],argv[2]);
}

code(input,output)
char *input,*output;
{
        int ch,change,t;
        FILE *fp1,*fp2;

        if((fp1=fopen(input,"r"))==0)  {
                printf("arquivo inacessivel para entrada\n");
                exit();
        }

        if((fp2=fopen(output,"w"))==0)  {
                printf("arquivo inacessivel para saida\n");
                exit();
        }

        change=1;
        do  {
                ch=getc(fp1);
                if(ch==EOF)  break;
                t=index(ch);
                count[t]++;
                if(isalpha(ch) || ch==' ')  {
                        if(change)
                                ch=sub[find(alphabet,ch)];
                        else  {
                                ch=sub2[find(alphabet,ch)];
                        }
                }
                putc(ch,fp2);
                if(count[t]==2)  {
                        change=!change;
                       count[t]=0;
                }
        }  while(1);
        fclose(fp1); fclose(fp2);
}
```

```
decode(input,output)
char *input,*output;

{
        int ch,change;
        FILE *fp1,*fp2;

        if((fp1=fopen(input,"r"))==0)  {
                printf("arquivo inacessivel para entrada\n");
                exit();
        }

        if((fp2=fopen(output,"w"))==0)  {
                printf("arquivo inacessivel para saida\n");
                exit();
        }

        change=1;
        do  {
                ch=getc(fp1);
                if(ch==EOF)  break;
                if(isalpha(ch)  ||  ch==' ')  {
                        if(change)  {
                                ch=alphabet[find(sub,ch)];
                                count[index(ch)]++;
                        }
                        else  {
                                ch=alphabet[find(sub2,ch)];
                                count[index(ch)]++;
                        }
                }
                putc(ch,fp2);
                if(count[index(ch)]==2)  {
                        change=!change;
                        count[index(ch)]=0;
                }
        }  while(1);
        fclose(fp1);fclose(fp2);
}

find(s,ch)
char *s;
char ch;
{
        register int t;

        for(t=0;t<28;t++) if(ch==s[t]) return t;
}

index(ch)
char ch;
```

```
{
        ch=tolower(ch);
        if(isalpha(ch)) return ch-'a';
        else return 26;
}
```

Por exemplo, depois de usar o programa na mensagem

encontre-me ao anoitecer

ele aparece como

sgzbglju-tsppfpqdfelsisn

Para ver como ele trabalha, examine o alfabeto ordenado e os dois alfabetos randomizados (chamados R1 e R2) montados um sobre o outro:

alfabeto	abcdefghijklmnopqrstuvwxy <espaço>
R1	qazwsxedcrfvthbyhnujm ikolp
R2	poi uytrewqasdfghjklmnbvcxz

Aqui está como o programa opera: no início, R1 é usado. A primeira letra na mensagem é "e", que corresponde a "s" em R1. A posição "e" é incrementada de 1 no vetor **count.** A próxima letra é "n", que torna-se "g", e a posição "n" é incrementada de 1 no vetor **count.** Quando o segundo "n" é encontrado, torna-se "g" porque R1 ainda está sendo usado e a posição "n" é incrementada de 2 no vetor **count.** Isto faz com que o programa mude para o alfabeto R2 e a posição "n" no vetor **count** é zerada. O "t" então torna-se "l". Depois do segundo "e" ser traduzido para "u" o alfabeto R1 é novamente usado, pois uma letra repetida foi encontrada. O processo de alternar alfabetos continua até o final da mensagem.

Usar a cifragem por substituição múltipla dificulta a quebra de códigos através de tabelas de freqüência. Seria possível usar vários alfabetos randomizados diferentes e uma rotina de troca mais complexa para ter todas as letras no texto codificado, ocorrendo com a mesma freqüência. Neste caso, uma tabela de freqüência seria inútil para quebrar o código.

CIFRAGEM POR TRANSPOSIÇÃO

Um dos mais antigos códigos de transposição conhecidos foi idealizado pelos espartanos por volta de 475 a.C. Usava um dispositivo chamado *skytale*. Um *skytale* é basicamente uma fita que é amarrada ao redor de um cilindro sobre a qual a mensagem é escrita transversalmente. A fita é, então, desamarrada e passada ao recipiente da mensagem que também possui um cilindro do mesmo tamanho. Teoricamente é impossível ler a fita sem o cilindro. Na prática, porém, este método deixa muito a desejar pois muitos cilindros de vários tamanhos podem ser experimentados até que a mensagem faça sentido.

Você pode criar uma versão computadorizada do *skytale*, colocando a sua mensagem em um vetor de uma certa maneira e escrevendo-a de maneira diferente. Por exemplo, se a seguinte **union**

```
union       message {
            char s[100];
            char s2[20][5];
}skytale;
```

é iniciada com nulos, e se você colocar a mensagem

encontre-me ao anoitecer

no vetor **skytale.s** mas vê-la como um vetor de duas dimensões **skytale.s2,** ela ficará assim

e	n	c	o	n
t	r	e	–	m
e	x	a	o	x
a	n	o	i	t
e	c	e	r	

Se você escrevesse o vetor por colunas, a mensagem ficaria assim:

eteae...nr nc...ceaoe...o-oir...nm t

onde os pontos indicam as posições nulas. Para decodificar a mensagem, colunas preenchem **skytale.s2.** Então o vetor **skytale.s** pode ser exibido na ordem normal. O vetor **skytale.s** pode ser impresso como uma série de caracteres porque a mensagem terá terminação nula. O programa Skytale de Cifragem usa este método para codificar e decodificar mensagens.

```
#include "ctype.h"
#include "stdio.h"
union message  {
        char s[100];
         char s2[20][5];
} skytale;

main(argc,argv)
int argc;
char *argv[];
{
        int t;

        for(t=0;t<100;++t)  skytale.s[t]=0; /* enche a matriz */

        if(argc!=4)  {
                printf("uso:entrada saida
                codificacao/decideficacao\n");
                exit();
        }

        if(toupper(*argv[3])=='C')  code(argv[1],argv[2]);
        else decode(argv[1],argv[2]);
}

code(input,output)
char *input,*output;
{
        int t,t2;
        FILE *fp1,*fp2;

        if((fp1=fopen(input,"r"))==0)  {
                printf("arquivo inacessivel para entrada\n");
                exit();
        }

        if((fp2=fopen(output,"w"))==0)  {
                printf("arquivo inacessivel para saida\n");
                exit();
```

```
        }
        for(t=0;t<100;++t)  {
                skytale.s[t]=getc(fp1);
                if(skytale.s[t]==EOF)  {
                        skytale.s[t]=0;
                        break;
                }
        }
        for(t=0;t<5;++t)
                for(t2=0;t2<20;++t2)
                        putc(skytale.s2[t2][t],fp2);

        fclose(fp1);  fclose(fp2);
}

decode(input,output)
char *input,*output;
{
        int t,t2;
        FILE *fp1,*fp2;

        if((fp1=fopen(input,"r"))==0)  {
                printf("arquivo inacessivel para entrada\n");
                exit();
        }



        if((fp2=fopen(output,"w"))==0)  {
                printf("arquivo inacessivel para saida\n");
                exit();
        }
        for(t=0;t<5;++t)
                for(t2=0;t2<20;++t2)
                        if((skytale.s2[t2][t]=getc(fp1))==EOF)
                        break;


        for(t=0;t<100;++t)
                putc(skytale.s[t],fp2);

        fclose(fp1); fclose(fp2);
}
```

Naturalmente, existem outros métodos para obter mensagens transpostas. Um método particularmente adaptável a computadores usa letras trocadas nas mensagens definidas por um algoritmo. Por exemplo, aqui está um programa que transpõe letras à frente de uma distância especificada pelo usuário, começando pela frente do vetor e alternando suas trocas com o fim dele:

```
#include "ctype.h"
#include "stdio.h"

main(argc,argv)
int argc;
char *argv[];
{
        int dist;

        if(argc!=5)  {
                printf("uso:entrada saida codificacao/
                decodificacao distancia\n");
                exit();
        }

        dist=atoi(argv[4]);
        if(toupper(*argv[3])=='C')  code(argv[1],argv[2],dist);
        else decode(argv[1],argv[2],dist);
}

code(input,output,dist)
char *input,*output;
int dist;
{
        char done,temp;
        int t;
        char s[256];
       FILE *fp1,*fp2;

        if((fp1=fopen(input,"r"))==0)  {
                printf("arquivo inacessivel para entrada\n");
                exit();
        }

        if((fp2=fopen(output,"w"))==0)  {
                printf("arquivo inacessivel para saida\n");
                exit();
        }

        done=0;
        do  {
                for(t=0;t((dist*2);++t)  {
                        s[t]=getc(fp1);
                        if(s[t]==EOF)  {
                                s[t]=0;
                                done=1;
                        }
                }
                for(t=0;t(dist;t++)  {
                        temp=s[t];
                        s[t]=s[t+dist];
```

```
                        s[t+dist]=temp;
                        t++;
                        temp=s[t];
                        s[t]=s[dist*2-t];
                        s[dist*2-t]=temp;
                }
                for(t=0;t<dist*2;t++)  putc(s[t],fp2);
        }  while(!done);
        fclose(fp1);  fclose(fp2);
}

decode(input,output,dist)


char *input,*output;
int dist;
{
        char done,temp;
        int t;
        char s[256];
        FILE *fp1,*fp2;

        if((fp1=fopen(input,"r"))==0)  {
                printf("arquivo inacessivel para entrada\n");
                exit();
        }

        if((fp2=fopen(output,"w"))==0)  {
                printf("arquivo inacessivel para saida\n"),
                exit();
        }

        done=0;
       do  {
                for(t=0;t<((dist*2);++t)  {
                        s[t]=getc(fp1);
                        if(s[t]==EOF)  {
                                s[t]=0;
                                done=1;
                        }
                }
                for(t=0;t<dist;t++)  {
                        t++;
                        temp=s[t];
                        s[t]=s[dist*2-t];
                        s[dist*2-t]=temp;
                        t--;
                        temp=s[t];
                        s[t]=s[t+dist];
                        s[t+dist]=temp;
                        t++;
```

```
            }
            for(t=0;t(dist*2;t++)  putc(s[t],fp2);
        }  while(!done);
        fclose(fp1); fclose(fp2);
}
```

Se você usar este método com a distância 10, a mensagem

encontre-me ao anoitecer

aparecerá como

etao anoi emcentro-n...e e...r c

Usando-se somente cifragem de transposição podem-se criar, acidentalmente, "dicas" no processo de transposição. No texto de exemplo que acabamos de mostrar, a palavra "emcentro-n" é sugestiva.

CIFRAGENS POR MANIPULAÇÃO DE BITS

O computador digital possibilita um novo método de codificação: manipulando bits que compõem os caracteres reais de um texto. Embora um purista diga que *manipulação de bits* (ou *alteração*, como algumas vezes é chamado) é, na realidade, só uma variação da cifragem por substituição, os conceitos, métodos, e opções diferem tão significativamente que deve ser considerado um método de cifragem com características próprias.

Cifragens por manipulação de bits, adaptam-se bem para uso em computadores porque empregam facilmente operações executadas pelo sistema. Textos cifrados também tendem a aparecer ininteligíveis, o que acrescenta segurança por fazer os dados parecerem-se com arquivos sem uso ou "crachados" e, desta forma, confusos para qualquer um que tente ter acesso ao arquivo.

Geralmente, cifragens por manipulação de bits são aplicadas somente para arquivos de computadores e não podem ser usadas para criar mensagens impressas porque a manipulação de bits tende a produzir caracteres que não podem ser impressos. Por esta

razão, você deve assumir que qualquer arquivo codificado por métodos de manipulação de bits ficará num arquivo de computador.

Cifragens por manipulação de bits convertem o texto pleno em texto cifrado através da alteração do padrão real dos bits de cada caractere através do uso de um ou mais dos seguintes operadores lógicos.

E
OU
NÃO
OU EXCLUSIVO
COMPLEMENTO DE 1

C é talvez a melhor linguagem para criar cifragens por manipulação de bits pois oferece os seguintes operadores bit a bit

Operador	Significado
|	OU
&	E
^	OU EXCLUSIVO
~	COMPLEMENTO DE 1

A mais simples e menos segura cifragem por manipulação de bits utiliza somente ~, o operador complemento de 1. (Lembre-se de que o operador ~ faz com que cada bit do byte seja invertido: 1 torna-se 0, e 0 torna-se 1.)Assim, um byte complementado duas vezes é o mesmo que o original. O programa de Cifragem por Complemento de 1, apresentado aqui, codifica qualquer arquivo texto.

```
#include "stdio.h"

main(argc,argv)
int argc;
char *argv[];
{
        if(argc!=4)  {
                printf("uso:entrada saida codificacao
                /decodificacao\n");
                exit();
        }
        if(toupper(*argv[3])=='C')  code(argv[1],argv[2]);
        else decode(argv[1],argv[2]);
}

code(input,output)
char *input,*output;
```

```
{
        int ch;
        FILE *fp1,*fp2;

        if((fp1=fopen(input,"r"))==0)  {
                printf("arquivo inacessivel para entrada\n");
                exit();

        }
        if((fp2=fopen(output,"w"))==0)  {
                printf("arquivo inacessivel para saida");
                exit();
        }

        do  {
                ch=getc(fp1);
                if(ch==EOF)  break;
                ch=~ch;
                if(ch=EOF)  ch++;
                putc(ch,fp2);
        }  while(1);
        fclose(fp1); fclose(fp2);
}

decode(input,output)
char *input,*output;
{
        int ch;
        FILE *fp1,*fp2;

        if((fp1=fopen(input,"r"))==0)  {
                printf("arquivo inacessivel para entrada\n");
                exit();
        }

        if((fp2=fopen(output,"w"))==0)  {
                printf("arquivo inacessivel para saida\n");
                exit();
        }


        do  {
                ch=getc(fp1);
                if(ch==EOF)  break;
                ch=~ch;
                if(ch==EOF)  ch--;
                putc(ch,fp2);
        }  while(1);

        fclose(fp1); fclose(fp2);
}
```

Observe que o caractere EOF tem tratamento especial para não confundir as rotinas de arquivos em disco. Se um EOF é criado pelo processo de encriptação, você deve alterá-lo para transformá-lo em outro caractere. Embora isto crie um duplo significado em alguns casos, normalmente não é possível criar um EOF por inversão de bits no arquivo texto; o código extra é só uma precaução. Assume-se também que EOF é –1, que é o caso para a maioria dos compiladores C; porém, se EOF não é –1, você deve consultar o seu manual do usuário e fazer as trocas necessárias. Se você acha que um duplo significado pode ocorrer, deve encontrar algum outro método de contornar este problema.

Não é possível mostrar com o que o texto cifrado da mensagem pareceria, porque a manipulação de bits usada aqui geralmente cria caracteres não imprimíveis. Experimente no seu computador e examine o arquivo – ele parecerá criptografado, com certeza!

Existem dois problemas com este esquema simples de codificação. Primeiro, o programa de encriptação não usa uma chave para decodificar; assim qualquer um com acesso ao programa pode decodificar um arquivo codificado. Segundo, e talvez o mais grave, este método seria facilmente percebido por qualquer programador experiente.

Um método melhor de codificação por manipulação de bits usa o operador XOR (ou exclusivo). O operador XOR possui a seguinte tabela verdade:

XOR	0	1
0	0	1
1	1	0

O resultado da operação XOR é VERDADE se e somente se um operando é VERDADEIRO e o outro é FALSO. Isto dá ao XOR a propriedade da unicidade: se você aplicar XOR a um byte com outro byte chamado *chave*, e então obtiver o resultado desta operação e aplicar XOR novamente com a chave, o resultado será o byte original, como visto aqui.

XOR	1	1	0	1	1	0	0	1	
	0	1	0	1	0	0	1	1	(chave)
	1	0	0	0	1	0	1	0	
XOR	1	0	0	0	1	0	1	0	
	0	1	0	1	0	0	1	1	(chave)
	1	1	0	1	1	0	0	1.	

iguais

Quando usado para codificar um arquivo, este processo soluciona dois problemas inerentes do método que utiliza o complemento de 1. Antes de mais nada, porque usa uma chave, o programa de encriptação sozinho não pode decodificar o arquivo; segundo, porque usar uma chave torna cada arquivo único, o que foi feito não é óbvio para alguém com formação única em ciência da computação.

A chave não precisa ter extensão de um só byte. Por exemplo, você poderia usar uma chave de vários caracteres e alternar os caracteres através do arquivo. Porém, uma chave de um único caractere é usada aqui para manter o programa claro.

```
#include "stdio.h"

main(argc,argv)
int argc;
char *argv[];
{
        if(argc!=5)  {
                printf("uso:entrada saida codificacao
                /decodificacao chave\n");
                exit();
        }
        if(toupper(*argv[3])=='C')
        code(argv[1],argv[2],*argv[4]);
        else decode(argv[1],argv[2],*argv[4]);
}

code(input,output,key)
```

```
char *input,*output;
char key;
{
        int ch;
        FILE *fp1,*fp2;

        if((fp1=fopen(input,"r"))==0)  {
                printf("arquivo inacessivel para entrada\n");
                exit();
        }
        if((fp2=fopen(output,"w"))==0)  {
                printf("arquivo inacessivel para saida");
                exit();
        }

        do  {
                ch=getc(fp1);
                if(ch==EOF)  break;
                ch=ch^key;
                if(ch=EOF)  ch++;
                putc(ch,fp2);
        }  while(1);
        fclose(fp1); fclose(fp2);
}

decode(input,output,key)
char *input,*output;
char key;
{
        int ch;
        FILE *fp1,*fp2;

        if((fp1=fopen(input,"r"))==0)  {
                printf("arquivo inacessivel para entrada\n");
                exit();
        }

        if((fp2=fopen(output,"w"))==0)  {
                printf("arquivo inacessivel para saida\n");
                exit();
        }

        do  {
                ch=getc(fp1);
                if(ch==EOF)  break;
                ch=ch^key;
                if(ch==EOF+1)  ch--;
                putc(ch,fp2);
        }  while(1);
        fclose(fp1); fclose(fp2);
}
```

COMPRESSÃO DE DADOS

As técnicas de compressão de dados essencialmente espremem uma quantidade de dados de informação em uma área menor. Compressão de dados é freqüentemente usada em sistemas de computação para aumentar a capacidade de armazenamento (por redução das necessidades de armazenamento do usuário) para diminuir o tempo de transferência (especialmente em linhas telefônicas), e para oferecer um certo nível de segurança. Embora existam muitos esquemas de compressão de dados disponíveis, este capítulo examinará somente dois deles. O primeiro é *compressão de bits*, onde mais de um caractere é armazenado em um único byte, e o segundo é *deleção de caractere*, onde os caracteres reais do arquivo são deletados.

OITO EM SETE

A maioria dos computadores modernos usa bytes com tamanho de potência de dois por causa da representação binária dos dados da máquina. As letras maiúsculas e minúsculas e a pontuação requerem somente por volta de 63 códigos diferentes utilizando somente 6 bits para representar um byte. (O byte de 6 bits poderia ter valores de 0 a 63.) Porém, a maioria dos computadores utiliza um byte de 8 bits; assim 25% dos bytes armazenados são gastos em simples arquivos de texto. Você poderia, assim, compactar quatro caracteres em 3 bytes se pudesse usar os dois *últimos* bits de cada byte. O único problema é a maneira como os códigos ASCII são organizados – existem mais de 63 códigos de caractere em ASCII, e as letras maiúsculas e minúsculas do alfabeto caem mais ou menos no meio. Isto significa que alguns destes caracteres requerem pelo menos 7 bits. É possível usar uma representação diferente de ASCII (o que raramente é feito), mas é geralmente aconselhável. A opção mais fácil é compactar 8 caracteres em 7 bytes, explorando o fato que nenhuma letra ou pontuação comum usa o oitavo bit do byte. Assim, você pode usar o oitavo bit de cada um dos 7 bytes para armazenar o oitavo caractere. Porém, você deve entender que muitos computadores – incluindo o IBM-PC – usam caracteres de 8 bits para representar caracteres especiais ou gráficos. Além disso, alguns processadores de texto usam o oitavo bit para enviar instruções ao processador de texto. Assim, o uso deste tipo de compactação de dados só funcionará em arquivos "estritamente" em ASCII que não usam o oitavo bit para nada.

Para visualizar como ela trabalha, considere os 8 caracteres representados por 8 bits seguintes:

byte 1	0 1 1 1	0 1 0 1
byte 2	0 1 1 1	1 1 0 1
byte 3	0 0 1 0	0 0 1 1
byte 4	0 1 0 1	0 1 1 0
byte 5	0 0 0 1	0 0 0 0
byte 6	0 1 1 0	1 1 0 1
byte 7	0 0 1 0	1 0 1 0
byte 8	0 1 1 1	1 0 0 1

Como você pode ver, o oitavo bit é sempre 0. Ele sempre é empregado em checagem de paridade, a menos que seja usado. A maneira mais fácil de comprimir 8 caracteres em 7 bytes é distribuir os 7 bits significativos do byte 1 nas 7 posições correspondentes aos oitavos bits não usados dos bytes de 2 a 8. Os 7 bytes restantes então aparecerão assim:

byte 1 — lido de cima para baixo

byte 2	1	1	1	1	1	1	0	1
byte 3	1	0	1	0	0	0	1	1
byte 4	1	1	0	1	0	1	1	0
byte 5	0	0	0	1	0	0	0	0
byte 6	1	1	1	0	1	1	0	1
byte 7	0	0	1	0	1	0	1	0
byte 8	1	1	1	1	1	0	0	1

Para reconstruir o byte 1, basta recuperá-lo novamente juntando os oitavos bits de cada um dos 7 bytes.

Esta técnica de compressão comprime qualquer texto de 1/8, ou 12,5%. Isto é uma melhora substancial. Por exemplo, se você fosse transmitir o código-fonte de seu programa favorito para um amigo através de uma ligação telefônica a longa distância,

ɔouparia 12,5% do custo da ligação. (Lembre-se de que o código-objeto, ou a versão executável do programa, necessita de todos os 8 bits.)

O programa seguinte comprime o arquivo de texto usando o método que acabamos de descrever:

```
#include "stdio.h"

main(argc,argv)
int argc;
char *argv[];
{
        if(argc!=4)  {
                printf("uso:entrada saida codificacao
                /decodificacao\n");
                exit();
        }
        if(toupper(*argv[3])=='C')  compress(argv[1],argv[2]);
        else decompress(argv[1],argv[2]);
}

compress(input,output)
char *input,*output;
{
        char ch,ch2,t,done;
        FILE *fp1,*fp2;

        if((fp1=fopen(input,"r"))==0)  {
                printf("arquivo inacessivel para entrada\n");
                exit();
        }
        if((fp2=fopen(output,"w"))==0)  {
                printf("arquivo inacessivel para saida\n");
                exit();
        }

        done=0;
        do  {
                ch=getc(fp1);
                if(ch==EOF)  break;
                ch=ch << 1;
                for(t=0;t<7;++t)  {
                        ch2=getc(fp1);
                        if(ch2==EOF)  {
                                ch2=0;
                                done=1;
                        }
                        ch2=ch2 & 127;
                        ch2=ch2 | ((ch<<t) & 128);
```

```
                        putc(ch2,fp2);
                }
        }  while(!done);
        fclose(fp1); fclose(fp2);
}

decompress(input,output)
char *input,*output;
{
        unsigned char ch,ch2,t,done;
        char s[7],temp;
        FILE *fp1,*fp2;

        if((fp1=fopen(input,"r"))==0)  {
                printf("arquivo inacessivel para entrada\n");
                exit();
        }

        if((fp2=fopen(output,"w"))==0)  {
                printf("arquivo inacessivel para saida\n");
                exit();
        }

        done=0;
        do  {
                ch=0;
                for(t=0;t<7;++t)  {
                        temp=getc(fp1);
                        if(temp==EOF)  break;
                        ch2=temp;
                        s[t]=ch2 & 127;
                        ch2=ch2 & 128;
                        ch2=ch2 >> t+1;
                        ch=ch | ch2;
                }
                putc(ch,fp2);
                for(t=0;t<7;++t)  putc(s[t],fp2);
        }  while(temp!=EOF);
        fclose(fp1); fclose(fp2);
}
```

O código deste programa é um pouco complexo porque vários bits devem ser deslocados. Tenha em mente que alguns compiladores C operam diferentemente com caracteres sinalizados do que com caracteres não sinalizados. (O compilador usado aqui – Aztec C para IBM-PC – exige que os caracteres não sinalizados sejam usados para deslocamento de bits, porém um caractere sinalizado deve ser usado para detectar EOF, necessitando, assim, do tipo de conversões vistas na rotina.)

A LINGUAGEM DO CARACTERE DE 16

Embora não se adapte à maioria das situações, um método interessante de compressão de dados deleta letras desnecessárias das palavras, abreviando, na verdade, a maioria das palavras. A compressão de dados é realizada porque os caracteres não utilizados não são armazenados. O uso de abreviações para poupar espaço é muito comum: por exemplo, "Sr." é normalmente usado no lugar de "Senhor". Em vez de usar abreviações comuns, o método apresentado nesta seção remove automaticamente certas letras da mensagem. Para fazer isto, um alfabeto mínimo será usado. Um *alfabeto mínimo* é aquele no qual muitas letras de uso freqüente são removidas, deixando somente aquelas necessárias para formar a maioria das palavras ou evitando ambigüidades. Assim, qualquer caractere que não esteja no alfabeto mínimo será extraído de qualquer palavra na qual aparece. O número exato de caracteres de um alfabeto mínimo é o "x" da questão. Porém, esta seção usará as 14 letras mais comuns, mais espaços e novas linhas.

Automatizar o processo de abreviação requer conhecimento das letras do alfabeto que são usadas com maior freqüência; assim, você pode criar um alfabeto mínimo. Teoricamente, você poderia contar as letras de cada palavra de um dicionário, pois diferentes editores usam diferentes freqüências. Deste modo, um diagrama de freqüências baseado somente nas palavras que fazem sentido na língua portuguesa não deve refletir o uso real de freqüência das letras. (E também levaria muito tempo para contar as letras!) Como uma alternativa, você pode calcular a freqüência das letras neste capítulo e usá-las como base para o nosso alfabeto mínimo. Para isto, você poderia utilizar este programa, o qual salta toda a pontuação exceto pontos, vírgulas e espaços.

```
#include "stdio.h"
#include "ctype.h"

main(argc,argv)
int argc;
char *argv[];
{
        FILE *fp1;
        int alpha[26],t;
        int space=0,period=0,comma=0;
        char ch;

        if(argc!=2)  {
                printf("Por favor especifique
                o arquivo do texto.\n");
                exit(0);
        }
```

```
        if((fp1=fopen(argv[1],"r"))==0)  {
                printf("arquivo inacessivel para entrada\n");
                exit();
        }

        for(t=0;t<26;t++) alpha[t]=0;
        do  {
                ch=getc(fp1);
                if(isalpha(ch))
                        alpha[toupper(ch)-'A']++;
                else switch(ch)  {
                        case ' ': space++;
                                  break;
                        case '.': period++;
                                  break;
                        case ',': comma++;
                                  break;
                }
        } while(ch!=EOF);

        for(t=0;t<26;++t)
                printf("%c: %d\n",'A'+t,alpha[t]);

        printf("period: %d\n",period);
        printf("space: %d\n",space);
        printf("comma: %d\n",comma);
        fclose(fp1);
}
```

Rodando o programa sobre o texto do Capítulo 1, você obtém as seguintes freqüências:

A	2292
B	168
C	966
D	1163
E	2237
F	233
G	317
H	102
I	1313
J	19
K	3
L	654
M	797

N	928
O	1738
P	722
Q	168
R	1488
S	976
T	1050
U	825
V	405
W	7
X	164
Y	50
Z	52
pontos	103
espaço	10871
vírgula	225

A freqüência das letras no Capítulo 1 compara-se bem com o português de maneira geral e fica um pouco fora só pelo uso muito repetitivo de palavras-chaves do C nos programas.

Para obter uma compressão de dados, você precisa cortar o alfabeto substancialmente, removendo letras usadas com menor freqüência. Embora existam muitas idéias diferentes sobre qual é exatamente o melhor alfabeto mínimo para se trabalhar, as 14 letras mais comuns e o espaço correspondem aproximadamente a 85% de todos os caracteres usados no Capítulo 1. O caractere nova linha também é necessário para manter as palavras, e sendo assim também precisa ser incluído. Desta forma, o alfabeto mínimo utilizado nesta seção constitui-se de 14 caracteres, espaço e nova linha.

A C D E I L M N O P R S T U <espaço> <nova linha>

Aqui está um programa que remove todos os caracteres a menos dos 16 selecionados.

```
#include "stdio.h"

main(argc,argv)
int argc;
char *argv[];
{
        if(argc!=3)  {
                printf("uso: entrada saida\n");
```

```
                exit();
        }
        comp2(argv[1],argv[2]);
}

comp2(input,output)
char *input,*output;
{
        char ch;
        FILE *fp1,*fp2;

        if((fp1=fopen(input,"r"))==0)  {
                printf("arquivo inacessivel para entrada\n");
                exit();
        }

        if((fp2=fopen(output,"w"))==0)  {
                printf("arquivo inacessivel para saida\n");
                exit();
       }

        do  {
                ch=getc(fp1);
                if(ch==EOF)  break;
                if(is_in(toupper(ch),"ACDEJILMNORSTU '\n'"))  {
                        if(ch=='\n')  putc('\r',fp2);
                        putc(ch,fp2);
                }
        } while(1);
        fclose(fp1); fclose(fp2);
}

is_in(ch,s)
char ch,*s;
{
        while(*s)  {
                if(ch==*s)  return 1;
                s++;
        }
        return 0;
}
```

Se você usar este programa na mensagem

À C&M S.A.

Esclarecemos que devido às alterações solicitadas, referentes à digitação do número do terminal através do teclado, podemos tornar incompatíveis as versões do interpretador.

Atenciosamente,

Engº Jose da Silva Neto

a mensagem comprimida aparece

A CM SA

Esclarecemos ue deido as alteracoes solicitadas reerente a diitacao do numero do terminal atraes do teclado odemos tornar incomatieis as ersoes do interretador.

Atenciosamente

En Jose da Silva Neto

Como você pode ver, a mensagem é bastante legível, embora alguma ambigüidade esteja presente. Ambigüidade é o principal problema deste método de compressão de dados. Porém, se você estiver familiarizado com o vocabulário do editor da mensagem, poderá provavelmente escolher um alfabeto mínimo melhor, que solucione em parte o problema da ambigüidade. Apesar do potencial de ambigüidade, foi salva uma quantidade razoável de espaço. A mensagem original tinha 243 bytes de comprimento e a mensagem compactada foi de 225 bytes de comprimento – um rendimento de aproximadamente 10%.

Se tanto a deleção de caracteres como a compressão de bits fossem aplicadas a uma mensagem, então aproximadamente 25% menos espaço de armazenamento seria necessário, o que pode ser significativo. Por exemplo, se você fosse o capitão de um submarino e quisesse enviar simplesmente sua posição, poderia comprimir a mensagem utilizando ambos os métodos; só assim seria a menor possível.

Tanto o método de compressão de dados por compressão de bits como por deleção de caracteres têm aplicação em encriptação. Compressão de bits não deixa de criptografar a informação, o que torna a decodificação mais difícil. Se utilizado antes da encriptação, o método de deleção de caracteres tem uma vantagem ótima: ele atrapalha a freqüência dos caracteres da linguagem-fonte.

QUEBRA DE CÓDIGOS

Nenhum capítulo sobre encriptação está completo sem um breve comentário sobre quebra de códigos. A arte de quebrar códigos é essencialmente a arte de tentativa e erro. Com o uso de computadores digitais, cifragens relativamente simples podem ser facilmente quebradas através de exaustivas tentativas e erros. Porém, os códigos mais

complexos ou não podem ser quebrados ou requerem técnicas e recursos normalmente não disponíveis. Para simplificar, esta seção enfoca a decodificação de códigos mais diretos.

Se você quiser quebrar uma mensagem que foi cifrada utilizando-se do método de substituição simples e com um só deslocamento no alfabeto, então só precisará tentar todos os 26 possíveis deslocamentos para ver se algum se encaixa. Um programa que faz isto é mostrado aqui.

```
#include "ctype.h"
#include "stdio.h"

main(argc,argv)
int argc;
char *argv[];
{
        if(argc!=2) {
                printf("uso: entrada\n");
                exit();
        }
        bc(argv[1]);
}

bc(input)
char *input;
{
        register int t,t2;
        char ch,s[1000],q[10];
        FILE *fp1;

        if((fp1=fopen(input,"r"))==0) {
                printf("arquivo inacessivel para entrada\n");
                exit();
        }

        for(t=0;t<1000;++t) {
                s[t]=getc(fp1);
                if(s[t]==EOF) break;
        }
        s[t]='\0';
        fclose(fp1);
        for(t=0;t<26;++t) {
                for(t2=0;s[t2];t2++) {
                        ch=s[t2];
                        if(isalpha(ch)) {
                                ch=tolower(ch)+t;
                                if(ch>'z') ch-=26;
                        }
                        printf("%c",ch);
```

```
                }
                printf("\n");
                printf("decodificou? (s/n): ");
                gets(q);
                if(*q=='s')  break;
        }
        printf("\noffset: %d",t);
}
```

Com uma pequena variação, você poderá usar o mesmo programa para quebrar cifragens que utilizam alfabeto randômico. Neste caso, substitua manualmente os alfabetos, como mostra este programa:

```
#include "ctype.h"
#include "stdio.h"
char sub[28];
char alphabet[28]="abcdefghijklmnopqrstuvwxyz ";

main(argc,argv)
int argc;
char *argv[];
{
        char s[80];

        if(argc!=2)  {
                printf("uso: entrada\n");
                exit();
        }

       do  {
                printf("Entre com uma letra do alfabeto:\n");
                gets(sub);
                bc2(argv[1]);
                printf("\nEsta certo?: (s/n) ");
                gets(s);
        }  while(tolower(*s)!='s');
}

bc2(input)
char *input;
{
        char ch;
        FILE *fp1,*fp2;

        if((fp1=fopen(input,"r"))==0)  {
                printf("arquivo inacessivel para entrada\n");
                exit();
```

```
        }

        do  {
                ch=getc(fp1);
                if(ch==EOF) break;
                if(isalpha(ch) || ch==' ')  {
                        putchar(alphabet[find(sub,ch)]);
                }
        }  while(1);
        fclose(fp1); fclose(fp2);
}

find(s,ch)
char *s;
char ch;
{
        register int t;
        for(t=0;t<28;t++) if(ch==s[t]) return t;
}
```

A menos de cifragens por substituição, cifragens por transposição e por manipulação de bits são dificilmente quebradas, utilizando o método de tentativas e erros. Se você tiver que quebrar estes tipos de códigos – boa sorte!

Ah, e por sinal – hsaovbno wlymljapvu pz haahpuhils, pa vjjbyz vusf hz hu hjjpklua.

Capítulo 8

GERAÇÃO DE NÚMEROS RANDÔMICOS E SIMULAÇÕES

Seqüências de números randômicos são usadas em várias situações de programação que vão desde simulações, que são as aplicações mais comuns, em jogos e outros softwares recreativos. A linguagem C não contém uma função intrínseca que gere números randômicos como outras linguagens de programação. Se bem que muitos compiladores C possuem alguma espécie de gerador de números randômicos em sua biblioteca-padrão, outros não. Além disso, a qualidade de muitos geradores de números randômicos não é suficiente para algumas situações de demanda, e algumas vezes é importante ter controle sobre a maneira como números randômicos são produzidos.

Neste capítulo você estudará como são escritas várias funções de geração de números randômicos e aprenderá como avaliá-las. Você também verá duas interessantes simulações que usam geradores de números randômicos desenvolvidas neste capítulo. A primeira é uma situação de um caixa de supermercado e a segunda é um gerenciador de carteira de investimentos.

GERADOR DE NÚMEROS RANDÔMICOS

Tecnicamente, o termo gerador de *números randômicos* é absurdo; números, por si só, são não-randômicos. Por exemplo, 100 é um número randômico? E 25? Claro que não. O que realmente significa "gerador de números randômicos" é alguma coisa que cria uma *seqüência* de números que parecem estar numa ordem aleatória. Isto nos leva a uma questão mais complexa. O que é uma seqüência aleatória de números? A única resposta correta é que uma seqüência aleatória de números é aquela na qual todos os elementos estão completamente desrelacionados. Esta definição nos leva a um paradoxo, de forma

que qualquer seqüência pode ser tanto randômica como não-randômica, dependendo da maneira como a seqüência é obtida. Por exemplo, a lista seguinte de números

1234567890

foi criada digitando-se as teclas superiores do teclado em ordem, assim a seqüência certamente não será considerada randômica. Mas e se isso tivesse acontecido ao se retirar exatamente esta seqüência de uma esfera cheia de bolas de ping-pong com números nelas escritos? Então ela *seria* uma seqüência randômica. Esta discussão mostra que a não-randomicidade de uma seqüência depende de *como ela foi gerada*, e não de qual é esta seqüência.

Saiba que estas seqüências de números gerados por um computador são *determinísticos:* cada número com exceção do primeiro depende do número anterior. Tecnicamente, isto significa que somente uma seqüência *quase randômica* pode ser criada por um computador. Porém, isto é suficiente para a maioria dos problemas, e para os propósitos deste livro, as seqüências serão simplesmente chamadas randômicas.

Geralmente, é melhor se os números em uma seqüência randômica são *uniformemente distribuídos*. (Não confunda isto com distribuição normal, ou curva do sino.) Em uma distribuição normal, todos os eventos são igualmente prováveis; assim, o gráfico da distribuição uniforme tende a ser uma linha chata ao invés de uma curva.

Antes da difusão dos computadores, sempre que números randômicos eram necessários, foram produzidos ou através de jogo de dados ou retirada de bolas numeradas em um jarro. Em 1955, a RAND Corporation publicou uma tabela de 1 milhão de dígitos randômicos obtidos com a ajuda de uma máquina parecida com um computador. Com o aparecimento da ciência da computação, embora muitos métodos fossem dirigidos para gerar números aleatórios, a maioria foi descartada.

Um método particularmente interessante que quase funcionou foi desenvolvido por John Von Neuman – o pai dos computadores modernos. Freqüentemente referenciado como *método meio quadrado,* ela eleva ao quadrado o número randômico anterior, e então extrai seus dígitos médios. Por exemplo, se você fosse criar números de três dígitos cujo valor tivesse sido 121, então você elevaria 121 ao quadrado e obteria 14641. Extraindo os três dígitos médios (do meio) você teria 464 como próximo número. O problema deste método é que ele tende a levar a um terno muito pequeno. Por esta razão o método não é usado atualmente. Hoje, a maneira mais comum de gerar números randômicos é usar a seguinte equação:

$$R_{n+1} = (aR_n+c) \bmod m$$

onde

$$R >= 0$$
$$a >= 0$$
$$c >= 0$$
$$m > \text{R, } a, \text{ e } c$$

Este método é muitas vezes referenciado como o *método da congruencialidade linear.* Observando a equação, você provavelmente pensa que geração de números randômicos é simples. Mas existe um porém – a eficiência desta equação depende fortemente dos valores de *a, c* e *m.* Escolher estes valores é algumas vezes mais uma arte do que uma ciência. Existem regras complexas que podem ajudá-lo a escolher estes valores, porém, esta discussão cobrirá somente algumas poucas e simples regras e experimentos.

O módulo *(m)* deve ser razoavelmente grande pois determina o intervalo dos números randômicos. A operação módulo fornece o resto da divisão que usa os mesmos operandos. Assim,

$$10 \% 4 = 2$$

porque 4 vai em 10 duas vezes e sobram 2. Desta forma, se o módulo é 12, então somente os números de 0 a 11 podem ser produzidos pela equação randômica; por outro lado, se o módulo é 21,425, os números de 0 até 21,424 podem ser produzidos. Lembre-se: um módulo pequeno não afeta realmente a randomicidade, só o intervalo. A escolha do multiplicador *a,* e o incremento, *c,* é muito pior. Normalmente, o múltiplo pode ser grande e o incremento bem pequeno. Muitos testes são necessários para confirmar que um bom gerador foi criado.

Como primeiro exemplo, aqui está um gerador comum de números randômicos. A equação mostrada em **RAN1()** foi usada como base para o gerador de números randômicos em um certo número de linguagens populares

```
float ran1()
 {
      static long int a=100001;
      a=(a*125)%2796203;
      return(float)a/2796203;
 }
```

Esta função tem três características importantes. Primeiro, o número randômico é inteiro – **long int** neste caso – mesmo que a função retorne **float.** Os inteiros são necessários para o método de congruencialidade linear, mas geradores de números randômicos, por convenção, espera-se que retornem um número entre 0 e 1, o que significa um ponto flutuante.

Segundo, a *semente,* ou valor inicial, é fornecida pelo hardware na função com o uso de *static long int a.* Este método fornece o valor da semente para a próxima chamada. Embora esta característica seja satisfatória para a maior parte das situações, é possível deixar o usuário especificar o valor inicial para tentar fazer a seqüência mais randômica. Se uma semente especificada pelo usuário é usada, a função é como segue:

```
float ran1(seed)
float seed;
{
      static long int a;
      static char once=1;

      if(once){
                a=seed*1000;
                /* pega o primeiro valor */
                once=0;
      }
      a=(a*125)%2796203;
      return(float) a/2796203;
}
```

Porém, para o restante deste capítulo, a semente será fornecida na função visando simplificar.

Terceiro, o número randômico é dividido pelo módulo primeiro antes de retornar. Assim, obtém-se um número entre 0 e 1. Se você estudar isto, verá que o valor de a anterior à linha de retorno deve estar entre 0 e 2796203. Assim, quando a é dividido por 2796203, um número igual ou maior do que 0 mas menor do que 1 é obtido.

Muitos geradores de números randômicos não são úteis porque produzem distribuições não-uniformes ou têm pequenas seqüências repetidas. Mesmo quando elas são imperceptíveis, estes problemas podem produzir resultados distorcidos se o mesmo

gerador de números randômicos é usado muitas vezes. A solução é criar vários geradores diferentes e usá-los individualmente ou juntá-los para obter números mais randômicos. Assim, os códigos de dois outros geradores de números randômicos são apresentados aqui. O gerador seguinte, chamado **RAN2()**, produz uma boa distribuição:

```
float ran2()
{
      static long int a=1;
      a=(a*32719+3)%32749;
      return (float) a/32749;
}
```

O gerador chamado **RAN3()** utiliza números bem pequenos

```
float ran3()
{
      static long int a=203;
      a=(a*10001+3)%1717;
      return (float) a/1717;
}
```

Cada um destes geradores de números randômicos produz uma boa seqüência de números randômicos. A questão permanece: Quão aleatórios são os números? Quão bom são esses geradores?

DETERMINANDO A QUALIDADE DE UM GERADOR

Você pode usar muitos testes para determinar a randomicidade de uma seqüência de números. Nenhum destes testes lhe mostrará se a seqüência é randômica, mas eles mostrarão se não o for. Os testes podem identificar uma seqüência não-randômica, mas só porque o teste não encontra um problema não quer dizer que a seqüência dada é certamente randômica. O teste, no entanto, dá uma garantia maior dos números do gerador de números randômicos que fornece uma seqüência. Para os propósitos deste capítulo, os testes ou são muito complicados ou exigem muito tempo para a maior parte dos geradores. Assim, você verá brevemente algumas das maneiras como uma seqüência pode ser testada e como ela pode falhar.

Para começar, aqui está a forma de encontrar o quão próxima está a distribuição de números na seqüência com aquilo que você esperaria ser uma distribuição randômica. Por exemplo, digamos que você está querendo gerar seqüências randômicas de dígitos de 0 a 9. Assim, a probabilidade de cada dígito ocorrer é 1/10, porque existem 10 possibilidades para cada número na seqüência, todas elas são igualmente possíveis. Assuma que a seqüência

9 1 8 2 4 6 3 7 5 8 2 9 0 4 2 4 7 8 6 2

foi gerada. Se você contar o número de vezes que cada dígito ocorre, o resultado é:

Dígito	Ocorrências
0	1
1	1
2	4
3	1
4	3
5	1
6	2
7	2
8	3
9	2

A pergunta que você deveria fazer a seguir é se esta distribuição é suficientemente parecida com a distribuição esperada.

Lembre-se: Se um gerador de números aleatórios é bom, ele gera seqüências aleatoriamente; em um estado verdadeiramente randômico, todas as seqüências são possíveis. Isso parece impor que qualquer seqüência gerada poderia qualificar-se como uma seqüência randômica válida. Mas como você pode dizer que uma seqüência é randômica? Na verdade, como poderia alguma seqüência de 10 dígitos ser não-randômica, se qualquer seqüência é possível? A resposta é que algumas seqüências têm *menos probabilidade* de serem randômicas do que outras. Você pode determinar a *probabilidade* de uma dada seqüência, usando o *teste do qui-quadrado*. O teste do qui-quadrado é basicamente subtrair o número de ocorrências esperadas de um número observado de ocorrências para todos os possíveis resultados para produzir um número, geralmente chamado de *V*. Você pode, então, olhar este número na tabela de qui-quadrado para encontrar a semelhança da seqüência com uma distribuição randômica. Uma pequena

tabela qui-quadrado é dada na Figura 8.1; você pode encontrar tabelas completas na maioria dos livros de estatística.

A fórmula para obter V é:

$$V = \sum_{1 \le i \le N} \frac{(O_i - E_i)^2}{E_i}$$

	p=99%	p=95%	p=75%	p=50%	p=25%	p=5%
n=5	0.5543	1.1455	2.675	4.351	6.626	11.07
n=10	2.558	3.940	6.737	9.342	12.55	18.31
n=15	5.229	7.261	11.04	14.34	18.25	25.00
n=20	8.260	10.85	15.45	19.34	23.83	31.41
n=30	14.95	18.49	24.48	29.34	34.80	43.77

Figura 8.1. Valores selecionados de qui-quadrado.

onde O_i é o número de ocorrências observadas, E_i é o número de ocorrências esperadas, e N é o número de elementos discretos. O valor para E_i é determinado através do produto da probabilidade com que estes elementos ocorrem pelo número de observações. Neste caso, porque espera-se que cada dígito ocorra 1/10 das vezes, se 20 amostras fossem levantadas, o valor para E será 2 para todos os dígitos. N é 10 porque são 10 elementos possíveis, os dígitos de 0 a 9. Assim

$$V = \frac{(1-2)^2}{2} + \frac{(1-2)^2}{2} + \frac{(4-2)^2}{2} + \frac{(1-2)^2}{2} + \frac{(3-2)^2}{2} + \frac{(1-2)^2}{2} +$$

$$\frac{(2-2)^2}{2} + \frac{(2-2)^2}{2} + \frac{(3-2)^2}{2} + \frac{(2-2)^2}{2} = 5$$

Para determinar a possibilidade desta seqüência não ser randômica, encontre a linha na Tabela 8.1 com o mesmo número de observações; neste caso, é 20. Então através da tabela você procura o número que é maior do que *V*. Neste caso, é a coluna 1. Isto significa que existem 99% de chance de a amostra de 20 elementos ter um *V* maior do que 8,260. Assim, existe somente 1% de probabilidade que a seqüência seja randômica. Ao passar o teste qui-quadrado, a probabilidade de *V* deve cair entre 25% e 75%. (Este intervalo deriva-se de cálculos que vão além do escopo deste livro.)

Você deve, no entanto, tirar esta conclusão com a seguinte questão: se todas as seqüências são possíveis, como pode essa seqüência ter somente 1% de chance de ser legítima? A resposta é que é só uma probabilidade – o teste qui-quadrado não é na realidade um teste determinístico, só uma referência. Na verdade, se você usar o teste qui-quadrado, deverá obter várias seqüências diferentes e a média dos resultados evitará que seja rejeitado um bom gerador de números randômicos. Qualquer seqüência simples pode ser rejeitada, mas médias de muitas seqüências juntas devem fornecer um bom teste.

Por outro lado, uma seqüência pode passar o teste qui-quadrado e ainda não ser randômica. Por exemplo,

1 3 5 7 9 2 4 6 8

passaria o teste qui-quadrado mas não parece muito randômica. Neste caso, uma *CORRIDA* foi gerada. Uma *CORRIDA* é simplesmente uma seqüência de números estritamente ascendente ou descendente que estão distribuídos em distâncias pares. Cada grupo de 4 dígitos está em ordem estritamente ascendente e assim (assuma que ela continuará) não seria uma seqüência randômica. Corridas podem ser separadas por dígitos "lixo", da seguinte forma: os dígitos que comprimem a *CORRIDA* podem ser intercalados por seqüências randômicas. É possível desenvolver testes para detectar estas situações, mas estão fora do propósito deste livro.

Outra característica que deve ser testada é o tamanho do *período;* isto é, quantos números podem ser gerados antes da seqüência começar a repetir-se – ou pior, degenerar-se em um ciclo curto. Todos os geradores de números randômicos baseados em computadores repetem eventualmente uma seqüência. Quanto maior o período, melhor o gerador. Mesmo que a freqüência dos números dentro do período esteja uniformemente distribuída, os números não constituem-se de uma série randômica, pois uma série verdadeiramente randômica não se repetirá constantemente. Geralmente, um período de muitos milhares de números é suficiente para a maioria das aplicações. (Novamente, um teste para isso pode ser feito, mas está fora do propósito deste livro.)

Vários outros testes podem ser aplicados para determinar a qualidade de um gerador de números aleatórios. Provavelmente existem mais códigos escritos para testar geradores de números randômicos do que desenvolvidos para geração. Aqui temos um outro teste que possibilita o teste de geradores de números randômicos "visualmente", usando um gráfico para mostrar como a seqüência é gerada.

O gráfico está baseado na idéia de freqüência de cada número. Porém, como um gerador de números randômicos pode produzir milhares de números diferentes, isto é impraticável. Em vez disto, você criará um gráfico agrupado pelo décimo dígito de cada número; por exemplo, como todos os números randômicos produzidos estão entre 0 e 1, o número 0,9265783 é agrupado sobre 9 e 0,34523445 é agrupado sobre 3. Isto significa que o gráfico do programa Exibidor de Gerador de Números Randômicos tem 10 linhas, e cada linha representa o número de vezes que um determinado número no grupo ocorreu.

O programa imprime também a média da seqüência, que pode ser usada para detectar influência de cada número. Como em outros programas gráficos neste capítulo, este programa roda somente no IBM-PC que estiver equipado com o adaptador de vídeo gráfico-colorido.

```
int freq1[10]={0,0,0,0,0,0,0,0,0,0};
int freq2[10]={0,0,0,0,0,0,0,0,0,0};
int freq3[10]={0,0,0,0,0,0,0,0,0,0};

main()
{
        int x,y;
        float ran1(),ran2(),ran3();
        float f,f2,f3,r,r2,r3;
        char s[80];

        mode(4);
        ground(0);
        pallete(1);
                color('R');
        f=0;f2=0;f3=0;
        scr_curs(0,6);
        printf("Comparacao de numeros randomicos");
        scr_curs(2,15);
        printf("Geradores");
        line(0,20,90,20);
        line(110,20,200,20);
        line(220,20,310,20);
        scr_curs(23,3);
        printf("ran1()        ran2()        ran3()");

        for(x=0;x<1000;++x)  {
```

```
                r=ran1();
                f+=r;
                y=r*10;
                freq1[y]++;

                r2=ran2();
                f2+=r2;
                y=r2*10;
                freq2[2]++;

                r3=ran3();
                f3+=r3;
                y=r3*10;
                freq3[y]++;

                display();
        }
        gets(s);
        mode(3);
        printf("Funcao 1 do numero randomico: %f\n",f/1000);
        printf("Funcao 2 do numero randomico: %f\n",f2/1000);
        printf("Funcao 3 do numero randomico: %f\n",f3/1000);
}

display()
{
        register int t;
        for(t=0;t<10;++t)  {
                line(t*10,20,t*10,freq1[t]+20);
                line(t*10+110,20,t*10+110,freq2[t]+20);
                line(t*10+220,20,t*10+220,freq3[t]+20);
        }
}

float ran1()
{
        static long int a=100001;

        a= (a*125) % 2796203;
        return (float) a/2796203;
}

float ran2()
{
        static long int a=1;

        a= (a * 32719+3) % 32749;
        return (float) a/32749;
}

float ran3()
```

```
{
        static long int a=203;

        a= (a *10001 +3) % 1717;
        return (float) a/1717;
}
```

Neste programa, as funções **RAN1( )**, **RAN2( )** e **RAN3( )** foram incluídas para fazer uma comparação lado a lado. Cada função gera 1000 números, e baseado no dígito da décima posição, o vetor de freqüência é atualizado. A função **display( )** plota todos os três vetores de freqüência na tela depois de cada número randômico ser gerado; assim você pode observar a evolução do gráfico. A Figura 8.2 mostra a saída de cada gerador de números randômicos ao final de 1000 números. A média é 0,496960 para **RAN1( )**, 0,490550 para **RAN2( )** e 0,512488 para **RAN3( )**. Estes são os resultados acessíveis.

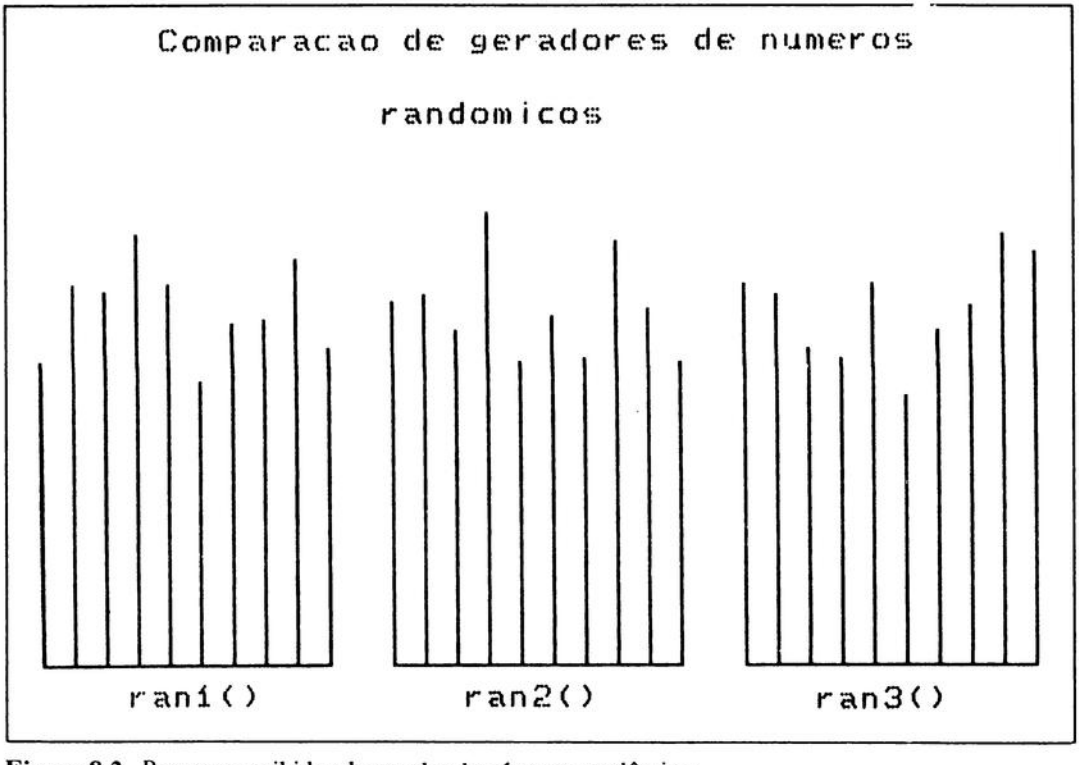


Figura 8.2. Programa exibidor de gerador de números randômicos.

Para utilizar o programa de exibição efetivamente, você deve ver a forma do gráfico e a maneira como ele cresce para checar, da melhor maneira, se ele está repetindo um ciclo. Este teste certamente não é conclusivo, mas ele fornece a você uma indicação da

maneira como o gerador produz seus números, e pode acelerar o processo de testes por permitir rejeitar rapidamente funções de pobres de geração. (Ele também é um ótimo programa para mostrar quando alguém lhe pede para mostrar seu computador!)

USANDO GERADORES MÚLTIPLOS

Uma técnica simples que melhora a randomicidade das seqüências produzidas pelos três geradores é combiná-los sobre um controle de uma função mestra. Esta função seleciona entre duas delas, baseada no resultado da terceira. Com este método você pode obter períodos bem grandes e diminuir o efeito de qualquer ciclo. A função chamada **random( )**, mostrada aqui, combina **RAN1( ), RAN2( )** e **RAN3( ).**

```
float random() /* geradores de seleção randômica */
{
      float f;
      f=ran3();
      if (f).5) return ran1();
      else return ran2();
}
```

O resultado de **RAN3( )** é usado para decidir se **RAN1( )** ou **RAN2( )** será o valor da função **random( ).** Experimente alterar a maneira como eles se combinam, trocando a constante no **if** para obter a distribuição exata que você deseja.

Aqui está um programa que exibe o gráfico de **random( )** e sua média.

```
int freq1[10]={0,0,0,0,0,0,0,0,0,0};
float random(),ran1(),ran2(),ran3();

main()
{
        int x,y;
        float f,f2,f2,r,r2,r3;
        char s[80];

        mode(4);
        ground(0);
        pallete(1);
        color('R');
        f=0;f2=0;f3=0;
        scr_curs(0,6);
        printf("A saida e obtida atraves de combinacao");
        scr_curs(2,5);
        printf("Gerador de tres numeros randomicos");
        for(x=0;x<1000;++x)  {
```

```
                r=random();
                f+=r;
                y=r*10;
                freq1[y]++;
                display();
        }
        gets(s);
        mode(3);
        printf("Funcao 1 do numero randomico: %f\n",f/1000);
}
display()
{
        register int t;
        for(t=0;t<10;++t)
                line(t*10+110,20,t*10+110,freq1[t]+20);
}

float random()
{
        float f;

        f=ran3();

        if(f).5) return ran1();
        else return ran2();
}

float ran1()
{
        static long int a=100001;

        a= (a*125) % 2796203;
        return (float) a/2796203;
}

float ran2()
{
        static long int a=1;

        a= (a * 32719+3) % 32749;
        return (float) a/32749;
}

float ran3()
{
        static long int a=203;


        a= (a * 10001 + 3) % 1717;
        return (float) a/1717;
}
```

A média de **random( )** é 0,49316.

A Figura 8.3 mostra o gráfico final depois que foram computados 1000 números.

SIMULAÇÕES

No restante deste capítulo, a aplicação de geradores de números randômicos será examinada em *simulações* no computador. Uma *simulação* é um modelo computadorizado de uma situação real. Qualquer coisa pode ser simulada, e o sucesso da simulação baseia-se principalmente no quanto o programador entende sobre o evento simulado. Como situações reais têm mil variáveis, muitas coisas são difíceis de serem efetivamente simuladas. Porém, existem vários eventos que podem muito bem ser simulados. Simulações são importantes por duas razões. Primeiro, elas permitem que você altere os parâmetros de uma situação para testar e observar possíveis resultados, mesmo que na vida real tais experiências possam ser muito caras ou perigosas. Por exemplo, a simulação de uma usina nuclear pode ser usada para testar os efeitos de certos tipos de falhas sem perigo. Segundo, simulações permitem criar situações que não podem ocorrer na vida real. Por exemplo, um psicólogo pode querer estudar os efeitos de gradualmente aumentar a inteligência de um rato para a de um humano a fim de descobrir em que ponto o rato percorre um labirinto mais rapidamente. Embora isto não possa ser feito na vida real, uma simulação pode providenciar uma apreciação da natureza, inteligência X instinto. Eis aqui o primeiro de dois exemplos de simulações que usam geradores de números randômicos.

SIMULANDO A FILA DO CAIXA

O primeiro exemplo simula uma fila no caixa de um supermercado. Assuma que a loja fica aberta durante 10 horas por dia, com horários de pico entre 12:00 e 13:00h, e das 17:00 às 18:00h. O intervalo das 12 às 13 é duas vezes mais movimentado que o normal e das 17 às 18 é três vezes mais movimentado. Enquanto esta simulação é executada, um gerador de números randômicos "cria" clientes, outro gerador determina quanto tempo levará para um cliente no caixa, e um terceiro gerador decide em qual das filas o cliente entrará. O objetivo da simulação é ajudar a gerência a encontrar um número ótimo de caixas necessárias para um dia típico, enquanto limita o número de pessoas na fila a qualquer hora do dia para no máximo 10, não ficando nenhum caixa parado.

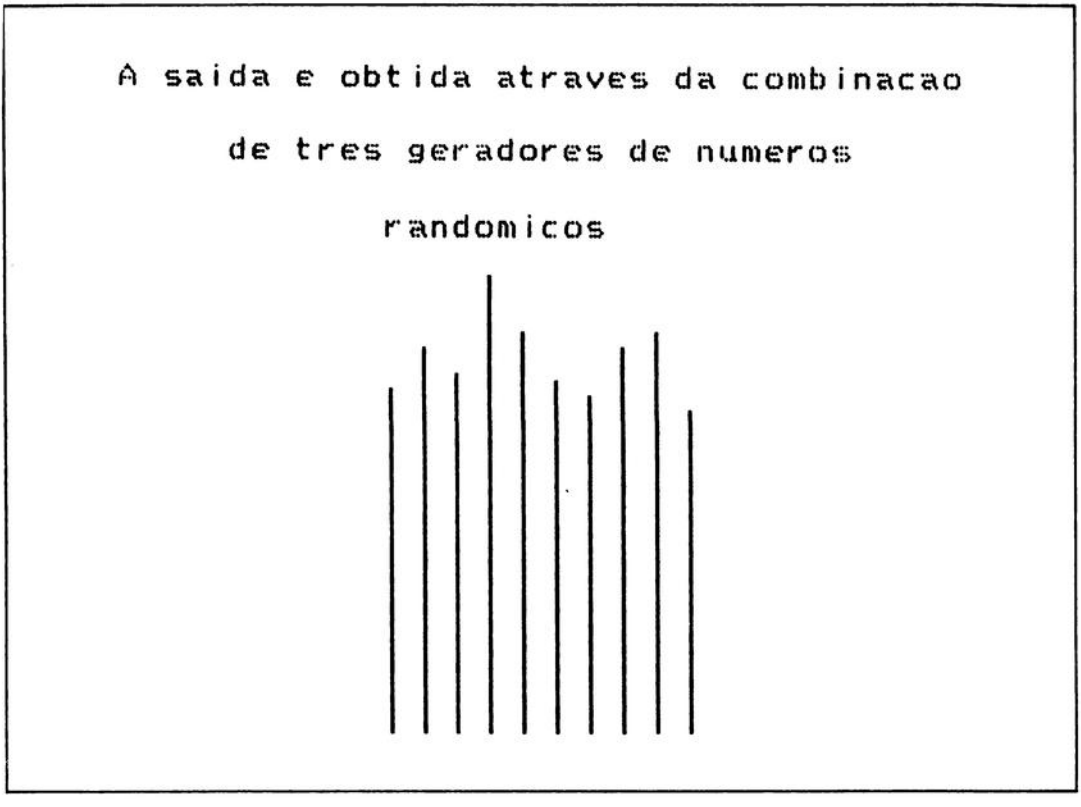


Figura 8.3. Gráfico final da função **random()**.

A chave para este tipo de simulação é criar processos múltiplos. Embora a linguagem C não suporte simultaneidade diretamente, você pode simular multiprocessamento pegando cada função que está dentro do laço do programa principal fazendo-a trabalhar e retornar – isto é, fatiando o tempo das funções. Por exemplo, a função que simula os caixas apenas checa uma parte de cada compra a cada instante que é chamada. Desta forma, cada função dentro do laço principal continua a ser executada. A função **main( )** do programa de caixa é vista aqui com seus dados globais.

```
float ran1(), ran2(), ran3();

char queues[10];
char open[10];
int cust;
int time=0;

main()
```

```
{
        int x,y;
        char s[80];

        mode(4);
        ground(0);
        palette(1);
        color('R');
        for(x=0;x<10;++x)  {
                queues[x]=0;
                open[x]=0;
        }
        scr_curs(24,20); printf("1          10");
        scr_curs(24,0); printf("Verifica linhas:");
        open[0]=1;
        do  {
                add_cust();
                add_queue();
                display();
                check_out();
                display();
                if(time>30 && time<50) add_cust();
                if(time>70 && time<80)  {
                        add_cust();
                        add_cust();
                }
                time++;
        }  while (!bdos(11,0,0) && time<100);
        gets(s);
        mode(3);
}
```

As funções **mode( ), ground( ), color( )** e **palette( )** são da biblioteca-padrão do Aztec C; são usadas para configurar a tela no modo gráfico 200 x 320 colorido com primeiro plano em vermelho e fundo preto. Procure configurar sua tela da mesma forma com as funções que seu compilador possui. Como nos capítulos anteriores, a função **scr-curs** move para uma posição específica X, Y.

O corpo do laço principal usado para executar a simulação é mostrado aqui.

```
add_cust();
add_queue();
display();
```

```
check_out();
display();
if(time>30 && time<50) add_cust();
if(time>70 && time<80)  {
        add_cust();
        add_cust();
}
time++;
```

A função **add-cust( )** usa tanto **RAN1( )** como **RAN3( )** para gerar o número de clientes que chegam às filas do caixa a cada requisição. A **add-queue( )** é usada para colocar os clientes em uma fila de caixa aberta, de acordo com os resultados de **RAN2( )**, e também abre uma nova fila se todas as caixas estiverem cheias. A **display( )** mostra uma representação gráfica da simulação. O **check-out( )** usa **RAN2( )** para fornecer a cada cliente sua conta, e cada chamada decrementa essa contagem de 1. Quando a contagem do cliente é 0, ele sai da fila do caixa.

A variável **time** altera a razão com que os clientes são criados no sentido de focalizar as horas de pico da loja. Na realidade, cada passagem através do laço é um décimo de uma hora.

As Figuras 8.4, 8.5 e 8.6 mostram o estado das filas de caixa quando **time** = 28, **time** = 60, e **time** = 88, correspondendo aos horários normal, ao final do primeiro pico, e ao fim do segundo pico respectivamente. Observe que no fim do segundo pico, o máximo de 6 caixas são necessárias. Isto significa que se a simulação foi feita corretamente, o supermercado não precisa operar as 4 caixas restantes.

Você pode controlar diretamente muitas variáveis do programa. Primeiro, pode alterar a maneira como os clientes chegam e o número de clientes que chegam. Você pode, também, trocar **add-cust( )** para retornar gradualmente mais ou menos clientes quando os horários de pico se aproximam e terminam. O programa assume que clientes escolherão randomicamente em qual fila entrarão. Embora esta maneira seja verdadeira para alguns clientes, outros obviamente escolherão a fila mais curta. Você pode compensar isto alterando a função **add-queue( )** para colocar os clientes na fila mais curta algumas vezes e colocar outros clientes randomicamente em outras vezes. A simulação não prevê acidentes ocasionais – tais como o caixa derrubar um vidro de massa de tomate – ou um cliente criador de casos em um caixa; em ambos os casos a fila pararia temporariamente.

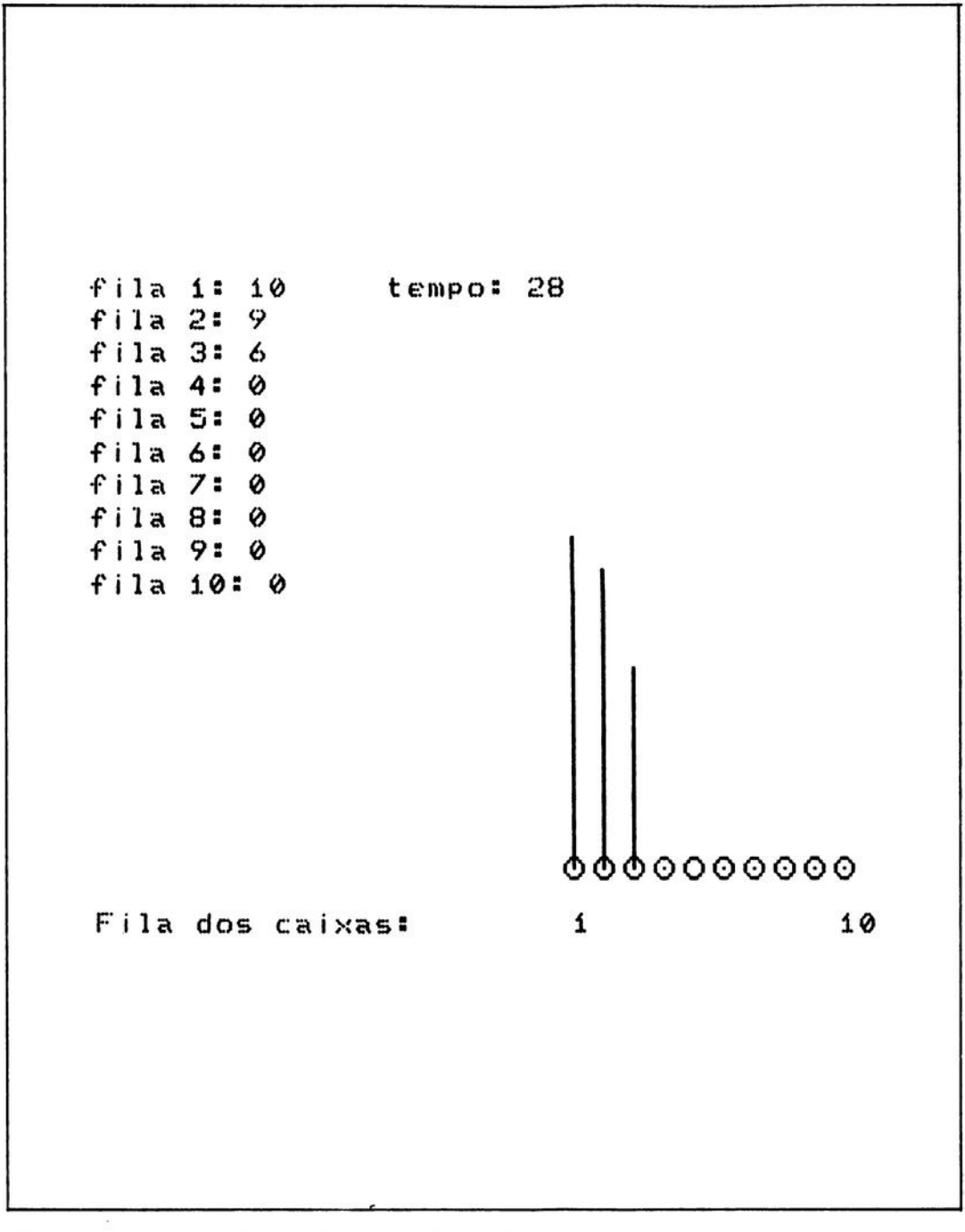


Figura 8.4. O estado da fila do caixa quando **time** = 28.

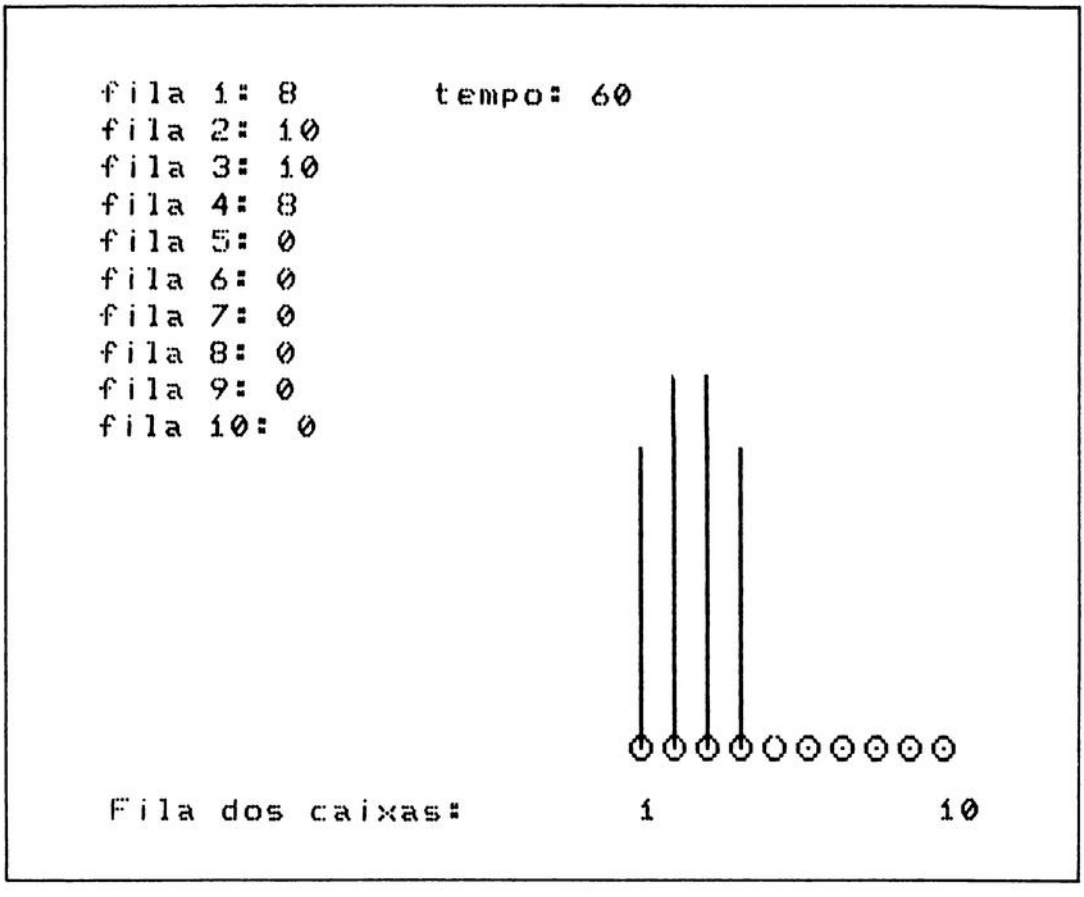


Figura 8.5. O estado da fila do caixa para **time** = 60.

O programa completo é visto aqui

```
float ran1(), ran2(), ran3();

char queues[10];
char open[10];
int cust;
int time=0;

main()
{
        int x,y;
        char s[80];

        mode(4);
```

```
        ground(0);
        palette(1);
        color('R');
        for(x=0;x<10;++x)  {
                queues[x]=0;
                open[x]=0;
        }
        scr_curs(24,20); printf("1           10");
        scr_curs(24,0); printf("Verifica linhas:");
        open[0]=1;
        do  {
                add_cust();
                add_queue();
                display();
                check_out();
                display();
                if(time>30 && time<50) add_cust();
                if(time>70 && time<80)  {
                        add_cust();
                        add_cust();
                }
                time++;
        }  while (!bdos(11,0,0) && time<100);
        gets(s);
        mode(3);
}

add_cust()
{
        float f,r;
        static char swap=0;

        if(swap) f=ran1();
        else f=ran3();
        swap=!swap;

        if(f<.5) return;
        else if(f<.6)  {
                cust++;
                return;
        }
        else if(f<.7)  {
                cust+=2;
                return;
        }
        else if(f<.8)  {
                cust+=3;
                return;
        }

        else cust+=4;
```

```
}

check_out()
{
        static char count[10]={0,0,0,0,0,0,0,0,0,0};
        register int t;

        for(t=0;t<10;++t)  {
                if(queues[t])  {
                        while(count[t]==0) count[t]=ran2()*5;
                        count[t]--;
                        if(count[t]==0) queues[t]--;
                }
                if(!queues[t]) open[t]=0;
        }
}

add_queue()
{
        register int t;
        int line;

        while(cust)  {
                if(allfull())
                        for(t=0;t<10;t++) if(!open[t])  {
                                open[t]=1;
                                break;
                        }
                line=ran3()*10;
                if(open[line] && queues[line]<10)  {
                        queues[line]++;
                        cust--;
                }
                if(t==10) return;
        }
}

allfull()
{
        register int t;

        for(t=0;t<10;t++) if(queues[t]<10 && open[t]) return 0;
        return 1;
}

display()
{
        register int t;

        scr_curs(0,15);
        printf("tempo: %d",time);
        for(t=0;t<10;++t)  {
```

```
                color(12);
                line((t*10)+160,20,(t*10)+160,120);
                color('R');
                circle((t*10)+160,20,3);
                line((t*10)+160,20,(t*10)+160,queues[t]*10+20);
                scr_curs(0+t,0); printf("fila %d: %d
                ",t+1,queues[t]);
        }
}

float ran1()
{

        static long int a=100001;

        a= (a*125) % 2796203;
        return (float) a/2796203;
}

float ran2()
{
        static long int a=1;

        a= (a * 32719+3) % 32749;
        return (float) a/32749;
}

float ran3()
{
        static long int a=203;

        a= (a *10001 + 3) % 1717;
        return (float) a/1717;
}
```

GERENCIADOR DE CARTEIRA DE ACESSO ALEATÓRIO

A arte de gerenciar uma carteira de ações está baseada em várias teorias e suposições sobre muitos fatores, alguns dos quais não são facilmente conhecidos a menos que você esteja integrado a eles. Existem estratégias de compra e venda baseadas em análises estatísticas de preços de ações e razões PE; existem correlações com o preço do ouro, o GNP, e mesmo as fases da lua. Para vingar-se, um cientista de computação utiliza o computador para simular o mercado livre – trocas de ações – sem nenhuma preocupação teórica.

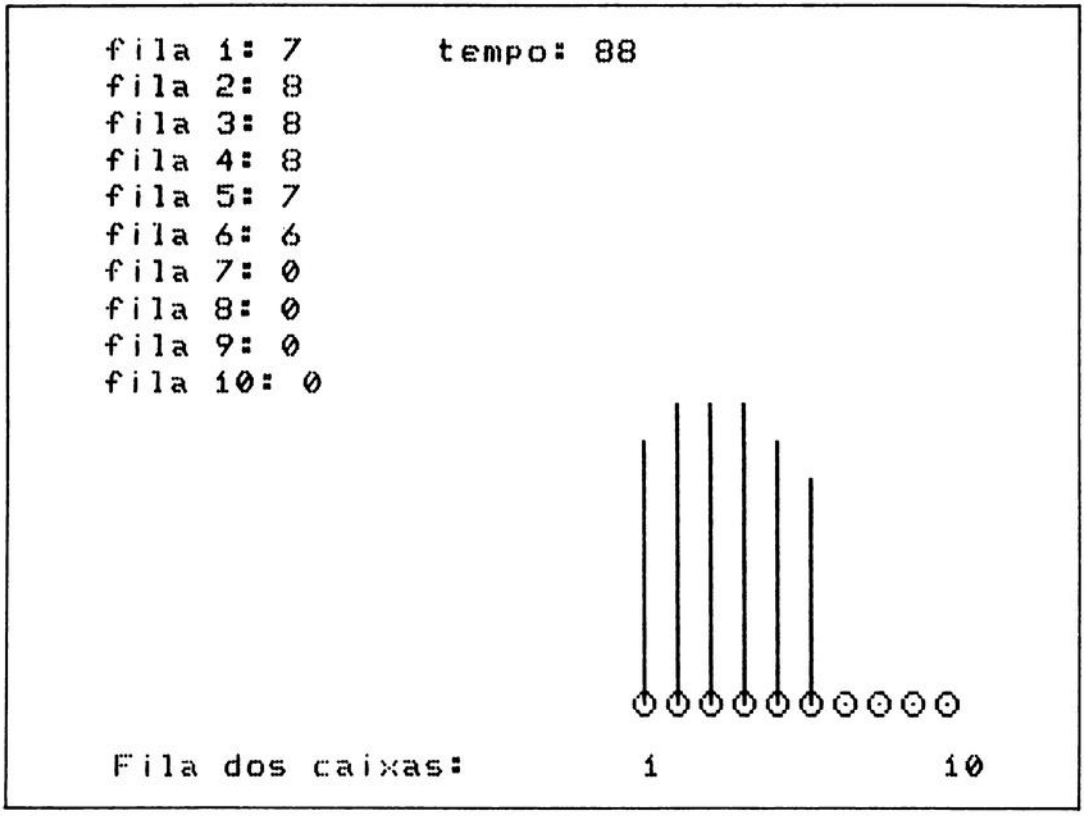


Figura 8.6. O estado da fila do caixa quando **time** = 88.

Você pode pensar que trocas de ações são simplesmente muito complicadas de simular, que contêm variáveis e muitas incógnitas; e que alternam-se muito rapidamente certas horas e muito vagarosamente em outras. Porém, o problema em si é a solução: pelo fato do mercado ser tão complexo, pode se pensar ser composto de *eventos de ocorrência aleatória*. Isto significa que você pode simular uma troca de ações como uma série desvinculada, ocorrências randômicas. Esta simulação está muito relacionada com o *método de acesso aleatório* de um gerenciador de carteira. O termo é derivado de um experimento clássico que envolve um bêbado desviando-se numa rua, aleatoriamente desviando-se de poste em poste. Com o método de acesso aleatório, você deixa a chance ser seu guia porque é tão bom quanto qualquer outro método.

Antes de continuar, preste atenção: o método de acesso aleatório é geralmente desacreditado por profissionais da área financeira; é apresentado aqui para sua apreciação, não para aplicações reais.

Para implementar o método de acesso aleatório, primeiro selecione 10 companhias da *Bolsa de Valores de São Paulo* por algum método de chances qualquer, assim como jogar dardos numa listagem de companhias e use os nomes das companhias que você acertou. Depois de selecionadas 10 companhias, coloque-as no programa de simulação de acesso aleatório e assim ele lhe dará o que pode fazer com elas.

Basicamente, o programa pode informar a você cinco coisas a fazer com as ações de cada companhia:

- Venda
- Compre
- Venda a termo
- Compre com margem
- Espere (não faça nada)

As operações de venda, compra e espera com ações são óbvias. Quando você *vende a termo,* vende ações que não possui, na esperança de que logo as compre mais barato e entregue-as à pessoa a quem vendeu. Vender a termo é uma forma de ganhar dinheiro quando o mercado está caindo. Quando você *compra com margem,* usa, por uma pequena remuneração, o dinheiro da corretagem para financiar parte do custo das ações que você adquiriu. A idéia por trás de *comprar com margem* é que você faz mais dinheiro do que faria se tivesse comprado uma pequena quantidade de ações à vista. Isto faz dinheiro somente em um mercado crescente.

O programa de Simulação por Acesso Aleatório é visto aqui. A chamada **bdos( )** confere o status do teclado e espera tecla selecionada. Isto lhe permite usar a seqüência produzida pelo gerador de números randômicos num ponto aleatório – na realidade, criando um valor inicial aleatório. Isto previne o programa de sempre produzir a mesma previsão.

```
char stock[10][30];     /* nome da firma */
float ran3();

main()  /* simulacao */
{
        register int t;
        char ch,s[80],*action();

        printf("Espere por um momento, e aperte uma tecla.\n");
        do  {
```

```
                ran3();
        }  while(!bdos(11,0,0));
        bdos(7,0,0);

        printf(" Entrar com um novo nome para estoque? ");
        gets(s);
        if(toupper(*s)=='S') enter();
        do  {
                for(t=0;t<10;++t)
                        printf("%30s: %s\n",stock[t],action());
                printf("\nnovamente? (s/n)");
                gets(s);
        }  while(toupper(*s)=='S' || *s==0);
}

enter()
{
        register int t;

        for(t=0;t<10;t++)  {
                printf("Entre com o nome da firma: ");
                gets(stock[t]);
        }
}

char *action()
{
        register int x;
        float f;

        f=ran3();
        x=f*10;

        switch(x)  {
                case 0: return "vender";
                case 1: return "comprar";
                case 3: return "vender poucas";
                case 4: return "comprar na margem";
                default: return "segurar";
        }
}

float ran1()
{
        static long int a=100001;

        a= (a*125) % 2796203;
        return (float) a/2706203;
}

float ran2()
```

```
{
        static long int a=1;


        a= (a * 32719+3) % 32749;
        return (float) a/32749;
}

float ran3()
{
        static long int a=203;

        a= (a * 10001 + 3) % 1717;
        return (float) a/1717;
}
```

O programa requer que você interprete as instruções da seguinte maneira:

Instrução	Interpretação
Compre	Compre o quanto puder de uma determinada ação sem se endividar.
Venda	Venda todas as suas ações, se acaso possuí-las. Então selecione uma nova companhia para reinvestir o seu dinheiro.
Compre a termo	Venda 100 cotas de uma companhia específica, mesmo que você possa comprá-las mais barato no futuro.
Venda com margem	Empreste dinheiro para comprar cotas de uma ação específica.
Espere	Não faça nada.

Por exemplo, se você fosse rodar esse programa usando nomes de companhias fictícias de Com1 até Com10, a previsão do primeiro dia seria como:

Com1:	venda
Com2:	compre
Com3:	compre a termo
Com4:	venda com margem
Com5:	espere

Com6:	espere
Com7:	espere
Com8:	compre
Com9:	espere
Com10:	venda com margem

O segundo dia de previsão será:

Com1:	espere
Com2:	espere
Com3:	venda
Com4:	venda com margem
Com5:	espere
Com6:	espere
Com7:	compre
Com8:	compre a termo
Com9:	espere
Com10:	venda

Se você preferir, pode rodar este programa semanalmente em vez de diariamente.

Experimente alterar o programa de alguma maneira. Por exemplo, você poderia alterar o programa para que ele lhe dê quantidades de ações para comprar ou vender, dependendo do investimento em dólares disponível. Novamente, lembre-se de que este programa é só para ilustrar e não é recomendado como uma maneira de fazer investimentos reais no mercado. Porém, é interessante criar uma carteira em uma folha de papel e acompanhar a sua performance.

Capítulo 9

COMPILAÇÃO DE EXPRESSÃO E AVALIAÇÃO

Como você escreve um programa que faz uma entrada de uma série de caracteres contendo uma expressão numérica, por exemplo 10–5*3, e retornar uma resposta, neste caso – 5? Se ainda existisse um "papa" entre programadores, então o programa poderia ser feito por aqueles poucos que sabem como fazer isso. Quase todos que usam o computador se mistificam pela forma como compiladores de linguagens de alto nível, programas de "planilha" e gerenciadores de banco de dados convertem expressões complexas, tais como 10*3–(4+c)/12, em expressões que o computador possa executar. Este processo de conversão é chamado *compilador de expressão*. Forma a espinha de todos os compiladores e interpretadores de linguagens, programas de planilha e qualquer outra coisa que converta estas expressões numéricas compreendidas por humanos de uma maneira que um computador possa usar. Poucos programadores sabem como escrever um compilador de expressão; este capítulo da programação é geralmente considerado como "fora dos limites", exceto por poucos iluminados.

Porém, este não é o caso. Um compilador de expressão é realmente muito direto e é parecido com outras tarefas de programação. Algumas vezes é mais fácil, porque trabalha estritamente com as regras da álgebra. Este capítulo desenvolve o que geralmente é conhecido como um *analisador recursivo descendente*, assim como todas as rotinas de suporte necessárias para permitir a você avaliar expressões numéricas complexas. Todas estas rotinas serão colocadas num arquivo que você poderá usar sempre que precisar. Depois que tiver dominado o compilador, você poderá alterá-lo e modificá-lo da maneira que melhor se adapte às suas necessidades – e junte-se aos "papas" você também.

EXPRESSÕES

Embora expressões possam ser feitas com todo tipo de formação, você estudará só um tipo: *expressões numéricas*. Para os propósitos deste capítulo, assuma que expressões numéricas podem ser feitas como a seguir:

- Números
- Operadores +, –, /, *, ∧, %, e =
- Parênteses
- Variáveis

O símbolo ∧ indica exponenciação, como em Basic, e o símbolo = representa o operador de asserção. Todos estes itens seguem as regras da álgebra com as quais você está familiarizado. Alguns exemplos de expressões são

```
10–8
(100–5)*14/6
a+b–c
10∧5
a=10–b
```

Assuma a seguinte prioridade para os operadores:

```
maior:  ∧
        * / %
        + –
menor:  =
```

Operadores de mesma prioridade avaliam-se da esquerda para a direita.

Por exemplo, neste capítulo, as seguintes afirmações serão feitas: todas as variáveis são letras únicas, o que significa que 26 variáveis – as letras de A a Z – são disponíveis. Todos os números são inteiros, embora você possa facilmente escrever rotinas que operem números em ponto flutuante. Finalmente, somente uma quantidade mínima de checagem de erros está incluída nas rotinas de forma a manter a lógica clara e ordenada.

Dê uma olhada na expressão

10–2*3

Esta expressão tem o valor 4. Embora você possa facilmente criar um programa que calcule esta expressão específica, deseja saber como criar um programa que forneça a resposta correta para qualquer expressão arbitrária deste tipo. Em primeiro lugar, você pode pensar em uma rotina como esta

```
a = pegue o primeiro operando
while (operandos presentes) {
        op = pegue operador
        b = pegue o segundo operando
        a = a op b
}
```

De acordo com esta rotina, você poderia pegar o primeiro operando, o operador e o segundo operando; executar a operação; executar a próxima operação; e assim por diante. Se você usar este método básico, a expressão 10–2*3 terá o valor 24 (que é 8*3) em vez da resposta correta que é 4, porque o procedimento omite a precedência dos operadores. Você não pode pegar os operandos e operadores na ordem da esquerda para a direita, porque a multiplicação deve ser feita antes da subtração. Um iniciante pode pensar que este problema pode ser facilmente resolvido – e algumas vezes, em casos muitos restritos, ele pode – mas o problema só piora quando parênteses, exponenciação, variáveis, chamadas de função, e coisas do gênero são acrescentadas.

Embora existam poucas maneiras de escrever funções que avaliem expressões deste tipo, você estudará a mais simples de escrever e também a mais comum. (Alguns outros métodos usados para escrever compiladores empregam tabelas complexas que quase sempre requerem outro programa para gerá-las. Estes métodos são chamados às vezes *compiladores dirigidos por tabelas*.) O método que você examinará é chamado compilador recursivo descendente, e você verá porque recebeu este nome, no decorrer deste capítulo.

DISSECANDO UMA EXPRESSÃO

Antes que você possa desenvolver um compilador para analisar expressões, deve recolher peças da expressão, facilmente. Por exemplo, dada a expressão

A*B–(W+10)

você deve ser capaz de pegar os operandos **A, B** e **W,** os parênteses, e os operadores *, +, e –. Em geral, você precisa de uma rotina que retorne cada item da expressão individualmente. A rotina também precisa ser capaz de desprezar espaços e tabulações, e deve saber quando chegou ao fim da expressão.

Formalmente, cada peça de uma expressão é chamada de *bastão.* Assim, a função que retorna o próximo bastão da expressão é chamada **get-token( ).** Um ponteiro caractere global é necessário para armazenar o ponteiro à expressão. Este ponteiro é chamado de **prog.** A variável **prog** é global porque precisa manter seu valor entre chamadas a **get-token( )** e deve permitir que outras funções o usem. Você também precisa saber que *tipo* de bastão está pegando. Para o compilador desenvolvido neste capítulo, você só precisa de três tipos: VARIÁVEL, NÚMERO e DELIMITADOR, onde DELIMITADOR é usado tanto para operador como para parênteses. Aqui está **get–token()** com suas globais necessárias, asserções **#define** e funções de suporte.

```
#define DELIMITER       1
#define VARIABLE        2
#define NUMBER          3

extern char *prog;
char token[80];
char tok_type;

get_token()
{
        register char *temp;

        tok_type=0;
        temp=token;

        while(iswhite(*prog)) ++prog;

        if(is_in(*prog,"+-*/^%()"))  {
                tok_type=DELIMITER;
                *temp++=*prog++;
        }
        else if(isalpha(*prog))
                serror(0);
        else if(isdigit(*prog))  {
                while(!isdelim(*prog)) *temp++=*prog++;
                tok_type=NUMBER;
        }

       *temp=0;
}
```

```
iswhite(c)
char c;
{
        if(c==' ' || c==9) return 1;
        return 0;
}

isdelim(c)
char c;
{
        if(is_in(c," +-/*^%()") || c==9 || c=='\r' || c==0)
                return 1;
        return 0;
}

is_in(ch,s)
char ch,*s;
{
        while(*s) if(*s++==ch) return 1;
        return 0;
}
```

O primeiro passo desta função é checar o terminador nulo, que indica o fim da expressão. Pelo fato da linguagem C usar seqüências de caracteres com terminação nula, se um nulo é encontrado, você sabe que a expressão acabou e um bastão nulo é retornado. Embora espaços sejam adicionados às expressões para uma melhor compreensão, estes espaços só confundem o compilador e você precisa desprezá-los.

Depois dos espaços terem sido desprezados, **prog** será apontado ou para um número, uma variável, um operador, ou um nulo, se seguidos espaços finalizam a expressão. Se o próximo caractere for um operador, ele será retornado como uma seqüência de caracteres na variável global **token,** e o tipo do DELIMITADOR é colocado em **tok–type.** Se o próximo caractere, porém, for uma letra, ele assumirá ser uma das variáveis e será retornado como uma seqüência de caracteres em **token; tok–type** sinalizará uma variável válida. Se o próximo caractere for um número, então um inteiro será retornado como uma série de caracteres em **token** com o tipo de NÚMERO. Finalmente, se o próximo caractere não for nenhum destes, você poderá assumir que o fim da expressão foi encontrado e este **token** é nulo.

Como foi dito anteriormente, para deixar o código limpo nesta função, várias checagens de erros foram omitidas e algumas premissas foram feitas. Por exemplo, qualquer caractere não reconhecido pode finalizar a expressão. Além disto, nesta versão, variáveis podem ter qualquer extensão, mas só a primeira letra é significativa. Porém, você pode completar estes e outros detalhes de acordo com a sua aplicação específica. Você

pode modificar ou aumentar **get–token** facilmente, de maneira a permitir variáveis maiores, números com pontos flutuantes, ou qualquer coisa que queira que seja desenvolvida por um bastão analisado.

Para entender como **get–token** funciona, estude o que ela retorna para cada um dos bastões da expressão A+100–(B*C)/2:

Bastão	**Tipo do Bastão**
A	variável
+	delimitador
100	número
–	delimitador
(	delimitador
B	variável
*	delimitador
C	variável
)	delimitador
/	delimitador
2	número
nulo	nulo

Não se esqueça de que **token( )** sempre contém uma série de caracteres nula, mesmo se esta expressão for simplesmente um caractere único.

ANÁLISE DE EXPRESSÃO

Lembre-se de que existem muitas maneiras de analisar e avaliar uma expressão. Para os propósitos deste capítulo, imagine uma expressão como *estrutura de dados recursivas* que são definidas em termos delas mesmas. Se, no momento, você restringir as expressões a usar somente +, –, *, /, e parênteses, poderá dizer que todas as expressões podem ser definidas pelas seguintes regras:

Expressão = > *[termo] [+termo] [–termo]*
termo = > *fator [*fator] [fator]*
fator = > *variável, número, ou (expressão)*

onde qualquer parte pode ser nula. Os colchetes significam opcional, e o => significa

"produz". Na verdade, as regras são chamadas de *regras de produção* da expressão. Assim você poderia ler a regra número 2 como "termo produz fator vezes fator, ou fator dividido por fator". A precedência dos operadores está implícita na maneira como a expressão é definida.

A expressão

10+5*8

tem dois termos: 10 e 5*8. Porém, tem três fatores: 10, 5, e 8. Estes fatores consistem-se de dois números e uma variável.

Este processo forma a base para um compilador recursivo descendente, que é basicamente um conjunto de rotinas mutuamente recursivas que trabalham de maneira encadeada. A cada passo determinado, o analisador compilador pode executar as operações específicas na seqüência algébrica correta. Para ver como este processo funciona, siga a análise da seguinte expressão

10/3–(100+56)

e execute as operações aritméticas a cada momento:

Passo 1. Pegue o primeiro termo: 10/3.

Passo 2. Pegue cada fator e divida os inteiros. Este valor é 3.

Passo 3. Pegue o segundo termo: (100+56).
Neste ponto você deve analisar a segunda expressão recursivamente.

Passo 4. Pegue cada fator e some. O resultado é 156.

Passo 5. Retorne da chamada recursiva e subtraia 156 de 3, que fornece uma resposta de –153.

Se você está um pouco confuso neste ponto, não se preocupe. Isto é um conceito complexo que precisa ser aplicado. Existem duas coisas para lembrar sobre esta visão recursiva das expressões: primeiro, a precedência dos operadores está *implícita* na maneira como as regras são definidas; segundo, este método de análise e avaliação de expressões é parecido com a maneira como você compilaria e avaliaria sem um computador.

UM COMPILADOR DE EXPRESSÕES SIMPLES

No restante deste capítulo dois compiladores são desenvolvidos. O primeiro compila e avalia só expressões constantes – isto é, expressões sem variáveis. Este é um compilador na sua forma mais simples. O segundo compilador inclui as 26 variáveis de A a Z.

Aqui está uma versão simples, completa, do compilador recursivo descendente para expressões inteiras.

```
#define DELIMITER       1
#define VARIABLE        2
#define NUMBER          3
extern char *prog;
char token[80];
char tok_type;

get_exp(result)
int *result;
{
        get_token();
        if(!*token)  {
                serror(2);
                return;
        }
        level2(result);
        return result;
}

level2(result)
int *result;
{
        register char op;
        int hold;

        level3(result);
        while((op = *token) == '+' || op == '-')  {
                get_token();
                level3(&hold);
                arith(op,result,&hold);
        }
}

level3(result)
int *result;
{
        register char op;
        int hold;
```

```
        level4(result);
        while((op = *token) == '*' || op == '/' || op == '%') {
                get_token();
                level4(&hold);
                arith(op,result,&hold);
        }
}

level4(result)
int *result;
{
        int hold;

        level5(result);
        if(*token== '^') {
                get_token();
                level4(&hold);
                arith('^',result,&hold);
        }
}

level5(result)
int *result;
{
       register char op;

        op = 0;
        if((tok_type == DELIMITER) &&
                * token == '+' || *token == '-') {
                op = *token;
                get_token();

        }
        level6(result);
        if(op)
                unary(op,result);
}

level6(result)
int *result;
{
        if((*token == '(') && (tok_type == DELIMITER)) {
                get_token();
                level2(result);
                if(*token != ')')
                        serror(1);
                get_token();
        }
        else
                primitive(result);
}
```

```
primitive(result)
int *result;
{
        register int i;

        if(tok_type==NUMBER)  {
                *result=atoi(token);
                return get_token();
        }
        serror(0);
}


arith(o,r,h)
char o;
int *r,*h;
{
        register int t,ex;

        switch(o)  {
                case '-':
                        *r=*r-*h;
                        break;
                case '+':
                        *r=*r+*h;
                        break;
                case '*':
                        *r=*r * *h;
                        break;
                case '/':
                       *r=(*r)/(*h);
                        break;
                case '%':
                        t=(*r) / (*h);
                        *r=*r - (t* (*h));
                        break;
                case '^':
                        ex=*r;
                        if(*h==0)  {
                                *r=1;
                                break;
                        }
                        for(t=*h-1;t>0;--t)  *r=(*r) * ex;
                        break;
        }
}


unary(o,r)
char o;
int *r;
{
```

```
        if(o=='-') *r=-(*r);
}

putback()
{
        char *t;

        t=token;
        for(;*t;t++) prog--;
}

serror(error)
int error;
{
        static char *e[]=  {
                "erro de sintaxe",
                "falta parenteses",
                "nao tem expressao"
        };
        printf("%s\n",e[error]);
}

get_token()
{
        register char *temp;
        tok_type=0;
        temp=token;

        while(iswhite(*prog)) ++prog;

        if(is_in(*prog,"+-*/^%()"))  {
                tok_type=DELIMITER;
                *temp++=*prog++;
        }
       else if(isalpha(*prog))
                serror(0);

        else if(isdigit(*prog))  {
                while(!isdelim(*prog)) *temp++=*prog++;
                tok_type=NUMBER;
        }

        *temp=0;
}

iswhite(c)
char c;
{
        if(c==' ' || c==9) return 1;
        return 0;
}
```

```
isdelim(c)
char c;
{
        if(is_in(c," +-/*^%()") || c==9 || c=='\r' || c==0)
                return 1;
        return 0;
}

is_in(ch,s)
char ch,*s;
{
        while(*s) if(*s++==ch) return 1;
        return 0;
}
```

O compilador, como visto, pode aceitar os operadores +, –, *, /, e %, assim como exponenciação (^), o menos unário, e parênteses. Contém seis níveis e a função **primitive( )**, que devolve o valor de um número inteiro. Além disso, contém também as rotinas **arith( )** e **unary( )** para executar as várias operações aritméticas, assim como **get-token( )**. Como foi discutido anteriormente, as duas globais **token** e **tok-type** retornam da expressão o bastão seguinte a seu tipo. O **externprog** é um ponteiro para a expressão, que é considerada carregada por outra parte do programa.

A função **main( )** que demonstra o uso do compilador é apresentada aqui:

```
char *prog;     /* guarda expressao a ser compilada */
main()
{
        char *malloc();
        int answer;
        char *p;

       p=malloc(100);
        if(!p)  {
                printf("nao alocou memoria\n");
                exit();
}

        do  {
                prog=p;
                printf("entre com a expressao: ");
                gets(prog);

                get_exp(&answer);

                printf("A resposta e: %d\n",answer);
        }  while(*p);
}
```

Para entender exatamente como o compilador avalia uma expressão, trabalhe sobre a seguinte expressão, que você pode assumir estar contida em **prog**.

10–3*2

Quando **get-exp( )** (a rotina de entrada do compilador) é chamada, ela pega o primeiro bastão, e se este bastão é nulo, imprime a mensagem **nenhuma expressão presente** e retorna. Se o bastão está presente, então **level2( )** é chamada. (A **level1( )** será momentaneamente acrescentada ao compilador quando o operador de asserção for adicionado, mas não é necessário aqui.)

Agora o bastão contém o número 10. A função **level2( )** chama **level3( )**, e **level3( )** chama **level4( )**, que por sua vez chama **level5( )**. A função **level5( )** observa se o bastão é um unário + ou –; neste caso não é, assim **level6( )** ou chama recursivamente **level2( )**, no caso de uma expressão com parênteses, ou chama **primitive( )** para encontrar o valor inteiro.

Finalmente, quando **primitive()** é executada e **result** contém o inteiro 10, **get–token** obtém outro bastão, e a função passa a retornar ao início da seqüência. O bastão é agora o operador – e as funções retornam ao **level2( )**.

O próximo passo é muito importante. Porque o bastão é –, é salvo, **get–token** obtém novo bastão 3, e a descendente da seqüência começa novamente. Mais uma vez, **primitive( )** é chamada, o inteiro 3 é devolvido em **result** e o bastão * é lido. Neste ponto a primeira operação aritmética ocorre com a multiplicação de 2 por 3. O resultado é, então, devolvido para **level2( )** e a subtração é executada fornecendo a resposta 4. Embora o processo possa parecer complicado a princípio, você deve trabalhar em outros exemplos para verificar, por você mesmo, que sempre funciona corretamente.

Você poderia usar este compilador como uma calculadora, como ilustração de uma amostra de programa gerenciador. Você pode também usar em uma base de dados ou uma aplicação simples de planilha. Antes que pudesse ser usado em uma linguagem ou numa calculadora sofisticada, o compilador teria de monitorar variáveis, que é o assunto da próxima seção.

ACRESCENTANDO VARIÁVEIS AO COMPILADOR

Todas as linguagens de programação e muitas calculadoras e planilhas usam variáveis para armazenar valores para uso posterior. O compilador simples da seção anterior deve ser expandido para incluir variáveis antes que você possa usá-lo para estes propósitos. Primeiro você precisa das variáveis propriamente ditas. O compilador só

reconhecerá as variáveis de A até Z, embora você possa expandi-lo se você quiser. Cada variável usa somente uma posição do vetor de 26 elementos. Assim, você deve acrescentar o seguinte:

```
int vars[26]= {          /* 26 user variables, A-Z */
        0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
        };
```

Como você pode ver, as variáveis são inicializadas com zero como uma cortesia para o usuário.

Você também precisa de uma rotina para ver o valor de uma dada variável. Como você está usando letras de A a Z como nome das variáveis, pode facilmente indexar o vetor *vars* baseado no seu nome. Aqui está a função **find-var( )**.

```
find_var(s)
char *s;
{
     if(!isalpha(*s)) {
          serror(1);
          return 0;
     }
     return vars[toupper(*token)-'a'];
}
```

Como descrito, a função aceita variáveis de nomes longos, mas só a primeira letra é significativa. Você pode modificar esta característica para aceitar o tamanho que você precisa.

Você pode também modificar a função **primitive( )** para tratar tanto números e variáveis como primitivas, como visto aqui.

```
primitive(result)
     int *result;
{
     register int i;
     switch(tok_type) {
     case VARIABLE:
```

```
                *result=find_var(token);
                return get_token();
        case NUMBER:
                *result=atoi(token);
                return get_token();
        default:
                serror(0);
        }
}
```

Tecnicamente, é tudo que você precisa para um compilador usar variáveis corretamente; porém, não existe nenhuma maneira destas variáveis conterem valores. Freqüentemente você pode assumir variáveis fora do compilador, mas, assim, como é possível tratar o = como um operador de asserção, existem muitas maneiras de incluí-lo no compilador. Um método é acrescentar o **level1( )** ao compilador.

```
level1(result)
int *result;
{
        int hold;
        int slot,ttok_type;
        char temp_token[80];

        if(tok_type==VARIABLE) {
                strcpy(temp_token,token);
                ttok_type=tok_type;

                slot=toupper(*token)-'A';
                get_token();
                if(*token != '=') {
                        putback();
                        strcpy(token,temp_token);
                        tok_type=ttok_type;
                }
                else {
                        get_token();
                        level2(result);
                        vars[slot]=*result;
                        return;
                }
        }
        level2(result);
}
```

Como você pode ver, deve observar a frente para saber se a asserção está sendo feita realmente; nesta situação você precisa salvar o estado do compilador e assim poderá ser restaurada se não for a asserção.

Aqui está o compilador completo aumentado.

```
#define DELIMITER       1
#define VARIABLE        2
#define NUMBER          3

extern char *prog;
char token[80];
char tok_type;

int vars[26]=  {          /* 26 variaveis para uso, A-Z */
        0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
        };

get_exp(result)
int *result;
{
        get_token();
        if(!*token)  {
                serror(2);
                return;
        }
        level1(result);
        return result;
}

level1(result)
int *result;
{
        int hold;
        int slot,ttok_type;
        char temp_token[80];

        if(tok_type==VARIABLE)  {
                strcpy(temp_token,token);
                ttok_type=tok_type;
                slot=toupper(*token)-'A';
                get_token();
                if(*token != '=')  {
                        putback();
                        strcpy(token,temp_token);
                        tok_type=ttok_type;
                }
                else  {
                        get_token();
```

```
                        level2(result);
                        vars[slot]=*result;
                        return;
                }
        }
        level2(result);
}

level2(result)
int *result;
{
        register char op;
        int hold;

        level3(result);
        while((op = *token) == '+' || op == '-')  {
                get_token();
                level3(&hold);
                arith(op,result,&hold);
        }
}

level3(result)
int *result;
{
        register char op;
        int hold;

        level4(result);
        while((op = *token) == '*' || op == '/' || op == '%')  {
                get_token();
                level4(&hold);
                arith(op,result,&hold);
        }
}

level4(result)
int *result;
{
        int hold;

        level5(result);
        if(*token== '^')  {
                get_token();
                level4(&hold);
                arith('^',result,&hold);
        }
}

level5(result)
int *result;
{
```

```
        register char op;

        op = 0;
        if((tok_type == DELIMITER) &&
        * token == '+' || *token == '-') {
                op = *token;
                get_token();
        }
       level6(result);
        if(op)
                unary(op,result);
}

level6(result)
int *result;
{
        if((*token == '(') && (tok_type == DELIMITER))  {
                get_token();
                level1(result);
                if(*token != ')')
                        serror(1);
                get_token();
        }
        else
                primitive(result);
}

primitive(result)
int *result;
{

        register int i;

        switch(tok_type)  {
        case VARIABLE:
                *result=find_var(token);
                return get_token();
        case NUMBER:
                *result=atoi(token);
                return get_token();
        default:
                serror(0);


        }
}

arith(o,r,h)
char o;
int *r,*h;
{
        register int t,ex;
```

```
        switch(o)  {
                case '-':
                        *r=*r-*h;
                        break;
                case '+':
                        *r=*r+*h;
                        break;
                case '*':
                        *r=*r * *h;
                        break;
                case '/':
                        *r=(*r)/(*h);
                        break;
               case '%':
                        t=(*r) / (*h);
                        *r=*r - (t* (*h));
                        break;
                case '^':
                        ex=*r;
                        if(*h==0)  {
                                *r=1;
                                break;
                        }
                        for(t=*h-1;t>0;--t)  *r=(*r) * ex;
                        break;
        }
}

unary(o,r)
char o;
int *r;
{
        if(o=='-') *r=-(*r);
}

putback()
{
        char *t;

        t=token;
        for(;*t;t++) prog--;
}

find_var(s)
char *s;
{
        if(!isalpha(*s))  {
                serror(1);
                return 0;
        }
}
```

```
serror(error)
int error;
{
        static char *e[]=  {
                "erro de sintaxe",
                "falta parenteses",
                "nao tem expressao"
        };
        printf("%s\n",e[error]);
}

get_token()
{
        register char *temp;

        tok_type=0;
       temp=token;

        while(iswhite(*prog)) ++prog;

        if(is_in(*prog,"+-*/^%=()"))  {
                tok_type=DELIMITER;
                *temp++=*prog++;
        }
        else if(isalpha(*prog))  {
                while(!isdelim(*prog))  *temp++=*prog++;
                tok_type=VARIABLE;
        }
        else if(isdigit(*prog))  {
                while(!isdelim(*prog)) *temp++=*prog++;
                tok_type=NUMBER;
        }

        *temp=0;
}

iswhite(c)
char c;
{
        if(c==' ' || c==9) return 1;
        return 0;
}


isdelim(c)
char c;
{
        if(is_in(c," +-/*^%=()") || c==9 || c=='\r' || c==0)
                return 1;
        return 0;
}
```

```
is_in(ch,s)
char ch,*s;
{
        while(*s) if(*s++==ch) return 1;
        return 0;
}
```

Para ver como esta versão do compilador funciona, você pode usar a mesma função **main( )** que usou para o compilador simplificado. Com o compilador maior você pode agora inserir expressões tais como

A=10/4
A–B
C=A(F–21)*

CHECAGEM DE SINTAXE EM UM COMPILADOR RECURSIVO DESCENDENTE

Numa compilação de expressão, o *erro de sintaxe* é uma situação na qual a expressão inserida não se encaixa nas regras restritas necessárias do compilador. Normalmente são causados por erros humanos – mais comumente, por erros de digitação. Por exemplo, as expressões seguintes não serão compiladas corretamente pelo compilador deste capítulo:

10**8
(10–5)*9)
/8

A primeira expressão tem dois operadores numa linha, a segunda tem um parênteses a mais e a última tem um sinal de divisão iniciando uma expressão. Nenhuma destas condições são permitidas por compiladores. É preciso proteger-se contra erros de sintaxe porque podem levar o compilador a fornecer resultados errôneos.

Como você estudou o código para os compiladores, provavelmente observou a função **serror( )**, que é chamada em certas situações. Diferente de muitos outros compiladores, o método recursivo descendente faz uma checagem fácil de sintaxe porque, para a maior parte, erros de sintaxe ocorrem tanto em **primitive( ), find-var( )**, ou **level6( )**, onde parênteses são checados. A checagem de sintaxe como agora se apresenta

tem só um problema: o compilador completo não aborta num erro de sintaxe. Isto pode causar múltiplas mensagens de erro ao ser zerado.

A melhor maneira de implementar **serror( )** é tê-la executando uma rotina **reset( )**. Muitos compiladores vêem com um par de chamadas de funções associadas, por exemplo **set-exit( )** e **reset( )**. Estas duas funções permitem ao programa desviar para uma função *diferente*. Assim, em **serror( )**, você executaria **reset( )** a algum ponto seguro do seu programa fora do compilador.

Se seu compilador não tem este tipo de par de rotinas ou se você tentar escrever um código portátil, sua única outra opção será acrescentar uma variável global que é checada a cada nível. A variável seria inicialmente FALSO, e qualquer chamada a **serror( )** transformaria esta variável em VERDADE, fazendo o compilador abortar uma função por mês.

Se você deixar o código da maneira que está, tudo o que acontecerá é que múltiplas mensagens de erros de sintaxe vão ser apresentadas. Isto poderia ser evitado em algumas situações, mas bem-vindo em outras porque vários erros serão pegos. Geralmente, porém, você irá querer melhorar a checagem de sintaxe antes de usá-la em programas comerciais.

Capítulo **10**

CONVERTENDO PASCAL E BASIC PARA C

Muitos programadores dedicam seu tempo a converter programas de uma linguagem para outra. Isto é chamado de *tradução*. Dependendo dos métodos que você utilizar para traduzir e o quanto você conhece das linguagens-fonte e destino, este processo pode ser fácil ou difícil. Este capítulo apresenta alguns tópicos e técnicas que o ajudarão a converter programas de Pascal e Basic para C.

Por que alguém iria querer traduzir um programa escrito numa linguagem em outra? Uma razão é a *manutenciabilidade:* um programa escrito numa linguagem não-estruturada como Basic é difícil de manter e aumentar. Outra razão é a *velocidade* e a *eficiência:* C como linguagem é muito eficiente, e algumas tarefas exigentes têm sido traduzidas para C com performance melhor. Uma terceira razão é *praticidade:* um usuário pode ver um programa útil feito em uma certa linguagem mas pode possuir ou usar um computador com uma linguagem diferente. Você provavelmente encontrará outras razões para traduzir um programa para linguagem C.

Pascal e Basic foram escolhidos entre uma centena de linguagens porque são linguagens populares entre os usuários de microcomputadores e porque representam fins opostos do espectro de linguagem de programação. Pascal é uma linguagem estruturada muito parecida com C, ao passo que Basic é uma linguagem não estruturada e não tem virtualmente nenhuma semelhança com C. Embora este capítulo não possa cobrir cada linguagem em todos seus detalhes, ele examinará muitos dos problemas mais importantes que você irá encontrar. Assumiu-se que você está familiarizado tanto com Pascal como com Basic.

CONVERTENDO PASCAL PARA C

Pascal e C têm muitas similaridades, especialmente no controle de suas estruturas e no uso de suas rotinas únicas com variáveis locais. Isto torna possível fazer muitas *traduções de uma para a outra:* você pode, na maior parte das vezes, simplesmente substituir a função ou palavra-chave equivalente em C. Com uma tradução de uma para outra você poderá usar o computador, assisti-lo no processo de tradução. Um programa simples de tradução será desenvolvido mais tarde neste capítulo.

Embora Pascal e C sejam similares, existem três diferenças entre elas. A primeira é que Pascal é mais restrito e em alguns pontos mais limitado que C. Por exemplo, Pascal padrão não somente torna mais difícil escrever programas de sistema (pois endereços de memória não podem ser diretamente carregados em ponteiros, como em C), mas também não executará conversões de tipos para você. A segunda e mais importante é que é formalmente em bloco estruturado, enquanto C não. O termo *bloco estruturado* refere-se a uma habilidade da linguagem em criar logicamente unidades de código conectadas que podem ser conjuntamente referenciadas. O termo também significa que procedimentos podem ter procedimentos integrados a eles, conhecido somente pelo procedimento de saída. Embora C seja normalmente chamado de linguagem em bloco estruturado porque permite a criação fácil de blocos de código, não permite que funções sejam definidas dentro de outras funções. Por exemplo, o seguinte código em Pascal é válido:

```
procedure A;
var
     x:integer;
     procedure B;
     begin
          writeLn('dentro do procedure B');
     end;

begin
     writeLn('dentro do procedure A');
     B;
end;
```

Neste caso, **procedure B** está definido dentro do **procedure A.** Isto quer dizer que **procedure B** é conhecido somente pelo **procedure A.** Fora do **procedure A,** outro **procedure B** pode ser definido sem conflito. O mesmo código traduzido para C, no entanto, precisaria ter duas funções:

```
A()
{
      printf("começando A\n");
      B();
}

B()
{
      printf("dentro da função B\n");
}
```

Além disso, você teria de certificar-se de que não existe nenhuma outra função chamada B em algum outro lugar do programa.

A segunda diferença entre Pascal e C é que todas as variáveis, funções e procedimentos do Pascal precisam ser declarados antes de serem usados. No Pascal padrão, isto significa que referências posteriores não são permitidas sem declarações **forward.** Em C, variáveis precisam ser declaradas antes de serem usadas, mas referências a funções posteriores não são restritivas – na verdade elas são bastante comuns.

A terceira diferença é que Pascal padrão não suporta compilações separadas, por outro lado compilação separada é aconselhável em C.

UMA COMPARAÇÃO ENTRE PASCAL E C

A Figura 10.1 compara operadores e palavras-chaves de Pascal com operadores e palavras-chaves de C. Como você pode ver, muitas palavras-chaves de Pascal não têm equivalentes em C porque Pascal usa palavras-chaves em lugares onde C usa operadores para executar os mesmos passos. Algumas vezes, Pascal é simplesmente "mais falador" do que C.

Pascal	C
and	&&
array	
begin	{
case	switch

Figura 10.1. Comparação de palavras-chaves e operadores Pascal e C.

Pascal	**C**
const	#define
div	/ (using integers)
do	
downto	
else	else
end	}
file	
forward	extern (on occasion)
for	for
function	
goto	goto
if	if
in	
label	
mod	%
nil	(sometimes \0)
not	!
of	
or	||
packed	
procedure	
program	
record	struct
repeat	do
set	
then	
type	
to	
until	while (as in do/while)
var	
while	while
with	

Figura 10.1. Comparação de palavras-chaves e operadores Pascal e C (continuação).

Além das palavras-chaves, Pascal tem muitos *identificadores-padrões* que podem ser usados diretamente em um programa. Estes identificadores podem ser funções (como **writeLn**) ou variáveis globais (como **MaxInt**) que armazenam informação sobre o estado do sistema. Pascal também usa identificadores-padrões para especificar tipos de dados

como **real, inteiro, Boolean** e **caractere.** A Figura 10.2 mostra vários identificadores-padrões do Pascal e seus equivalentes em C. Além destes, vistos na figura, muitas das funções intrínsecas do Pascal têm equivalentes em C que são encontrados na biblioteca-padrão; porém, elas podem variar de compilador para compilador.

Pascal	C
Boolean	char or integer
byte	char
char	char
EOF	EOF (in stdio library)
false	0
flush	flush( ) (in stdio library)
integer	integer
read	scanf( ) and others
real	float
true	any nonzero value
write	printf( )

Figura 10.2. Identificadores-padrões do Pascal e seus equivalentes em C.

Pascal também difere de C em seus operadores. A Figura 10.3 mostra os operadores do Pascal e seus equivalentes em C.

Pascal	C	Meaning
+	+	Addition
−	−	Subtraction
*	*	Multiplication
/	/	Division
div	/	Integer division
mod	%	Modulus
^		Exponentiation
:=	=	Assignment
=	==	Equals as a condition
<	<	Less than
>	>	Greater than
>=	>=	Greater than or equal to
<=	<=	Less than or equal to
<>	!=	Not equal

Figura 10.3. Operadores do Pascal e seus equivalentes em C.

CONVERTENDO "LOOPS" EM PASCAL EM "LOOPS" EM C

Loops (laços) de controle de programas são fundamentais para a maioria dos programas, em vista disto você precisa comparar laços do Pascal com os do C. Pascal tem três laços intrínsecos: **for, while** e **repeat–until.** C tem um laço correspondente para cada um.

for em Pascal tem a forma geral

for *valor inicial* **to** *valor final* **do statement;**

for em Pascal é muito mais limitado do que **for** em C porque ele não permite incrementos além de 1 (ou –1 se **downto** for usado), e porque a condição do loop está rigidamente amarrada ao mecanismo contador, ao contrário de C que tem uma configuração mais flexível. No entanto, estas diferenças não afetam o processo de tradução do Pascal para C, porque **for** em Pascal pode ser visto simplesmente como um subconjunto em **C.** Por exemplo, a declaração em Pascal.

```
for x:=10 to 100 do writeln(x);
```

pode ser traduzida para C como

```
for(x=10;x<=100;++x) printf("%d\n",x);
```

while em Pascal e **while** em C são virtualmente iguais. Porém, **repeat-until** em Pascal e **while** em C exigem que você use palavras-chaves diferentes e "inverta" a condição de teste. Isto ocorre porque **repeat–until** do Pascal faz com que loop corra até que *(until)* algo torne-se verdadeiro, por outro lado o **do–while** em C gira enquanto *(while)* a condição do laço *é* verdadeira. Uma tradução dos tipos de laços é vista a seguir

Pascal	**C**
while x<5 do	while(x<5)
begin	{
writeln(x);	printf("%d \n",x);
read(x);	x=getnum( );
end;	}
repeat	do {

```
    read(x);                    x=getnum( );
    writeln(x);                 printf("%d \n",x);
until x>5;                    } while(x<=5);
```

Observe a tradução do **repeat–until** para o **do–while;** você precisa inverter o sentido da condição de teste.

UM EXEMPLO DE TRADUÇÃO

Para sentir o gostinho do processo de tradução, acompanhe a conversão de um programa em Pascal para C. A seguir temos um programa simples em Pascal.

```
program test (input,output);
var qwerty: real;

procedure tom (x: integer);
begin
        writeln(x*2);
end;

function ken (w: real): real;
begin
        ken:=w/3.1415;
        qwerty:=23.34
end;

begin
        qwerty:=0;
        writeln(qwerty);
        writeln('Oi, pessoal');
        tom(25);
        writeln(ken(10));
        writeln(qwerty:2:4);
end.
```

Este programa em Pascal possui uma função e um procedimento declarados. Como

procedimentos e funções são a mesma coisa em C, você não precisa se preocupar com a diferença, exceto em retornar o valor corretamente. Assim, **procedure tom** torna-se

```
tom(x)
int x;
{
      printf("%d\n"), x*2);
}
```

e **function ken** fica

```
float ken(w)
float w;
{
      qwerty=23.34;
      return w/3.1415;
}
```

O fato de **ken( )** retornar uma **float**, você precisa explicitamente declarar isto, colocando a declaração de tipo **float** na frente do nome **ken.**

A seguir, o código de **program** (que começa com o primeiro **begin** que não está dentro de outra função ou procedimento) deve ser convertido em uma função **main( ).** E torna-se

```
main()
{
        qwerty=0;
        printf("%f",qwerty);
        printf("Oi, pessoal\n");
        tom(25);
        printf("%f\n",ken(10));
        printf("%2.4f\n",qwerty);
}
```

Para finalizar, você precisa declarar a variável global **qwerty** como uma **float.** Depois de fazer isto e juntar as peças, a tradução para C do programa em Pascal ficará assim

```
float qwerty;

main()
{
        qwerty=0;
        printf("%f",qwerty);
        printf("Oi, pessoal\n");
        tom(25);
        printf("%f\n",ken(10));
        printf("%2.4f\n",qwerty);
}

tom(x)
int x;
{
        printf("%d", x*2);
}

float ken(w)
float w;
{
        qwerty=23.34;
        return w/3.1415;
}
```

USANDO O COMPUTADOR PARA AJUDAR A CONVERTER PASCAL EM C

É possível fazer um programa para computador que aceita código-fonte em uma linguagem e tem saídas em outra. A melhor maneira de fazer isto é implementar um compilador de linguagem real para o código-fonte – mas em vez de gerar código, ela terá saídas na linguagem de destino. Você pode, ocasionalmente, encontrar propagandas para tais produtos em lojas de computadores, e seus altos preços refletem a complexidade da tarefa.

Um método mais modesto é construir um programa simples para auxiliar no seu esforço, executando algumas das tarefas simples de tradução. Este "auxílio computacional" pode fazer trabalhos de conversão muito fáceis.

Um tradutor auxiliar computacional aceita como entrada um programa na linguagem-fonte e executa todas as conversões de um para outro na linguagem de destino,

automaticamente, deixando as conversões mais pesadas para você. Por exemplo, para atribuir a **count** o valor 10 em Pascal, você faria

```
count:=10;
```

Em C, a declaração é a mesma, exceto que não existe ponto e vírgula. Assim, o programa auxiliar (de assistência) computacional pode trocar a declaração de asserção := em Pascal para = em C. Porém, as maneiras como programas em Pascal e C acessam arquivos em disco são diferentes, e não existe nenhuma maneira direta de realizar cada conversão automaticamente. O tradutor deixa estes tipos de conversões para você fazer.

Primeiro o tradutor precisa de uma função que retorna um bastão a cada instante do programa em Pascal. A função **get-token( )**, desenvolvida no Capítulo 9, pode ser modificada para este uso como é apresentado abaixo

```
get_token()
{
        register char *temp;

        tok_type=0; tok=0;
        temp=token;

        if(*prog=='\r') {
                *temp++='\r';
                *temp++='\n';
                *temp='\0';
                prog+=2;
                return;
        }

        if(*prog=='\0') {
                *temp='\0';
                return;
        }
        while(iswhite(*prog))  ++prog;

        if(*prog=='=') {
                prog++;
                strcpy(token,"==");
                return;
        }

        if(*prog==':') {
                prog++;
                if(*prog=='=')
                {
                        *temp++ = '=';
```

```
                    prog++;
                }
                else *temp++=':';

                *temp='\0';
                return;
        }

        if(is_in(*prog,"'"))  {
                *temp++="'";  prog++;
                while(!is_in(*prog,"'"))  *temp++=*prog++;
                *temp="'"; temp++; *temp='\0'; prog++;
                return;
        }

        if(is_in(*prog,"+-*;.,/^%()"))  {
                tok_type=OP;
                *temp=*prog;
               prog++;
                if(*temp=='.')  *temp=' ';
                temp++;
                *temp=0;
                return;
        }

        if(isalpha(*prog))  {
                while(isalpha(*prog)) *temp++=*prog++;
                *temp='\0';
                translate(token);
                return;
        }

        if(isdigit(*prog))  {
                while(!isdelim(*prog))  *temp++=*prog++;
                tok_type=NUMBER;
                *temp=0;
                return;
        }
        prog++; /* caracter desconhecido */
}
```

Em **get-token( )**, a asserção em Pascal := é convertida para C como =, e = é convertido em seu equivalente em C, ==. Enquanto isto é suficiente para um programa de tradução simples, um programa de tradução mais completo teria provavelmente feito esta conversão em uma rotina operadora de conversão maior.

A segunda rotina importante traduz palavras-chaves em Pascal e algumas

funções em suas correlatas em C. Novamente, a função **traslate( )** mostrada aqui não é a melhor maneira de codificar tal rotina, mas é suficiente para os propósitos do programa.

```
translate(s)
char *s;
{
        if(!strcmp(s,"and")) strcpy(s,"&&");
        else if (!strcmp(s,"begin")) strcpy(s,"{");
        else if (!strcmp(s,"case")) strcpy(s,"switch");
        else if (!strcmp(s,"div")) strcpy(s,"/");
        else if (!strcmp(s,"do")) strcpy(s,"do");
        else if (!strcmp(s,"else")) strcpy(s,"else");
        else if (!strcmp(s,"end")) strcpy(s,"}");
        else if (!strcmp(s,"forward")) strcpy(s,"extern");
        else if (!strcmp(s,"for")) strcpy(s,"for");
        else if (!strcmp(s,"function")) strcpy(s,"\n");
        else if (!strcmp(s,"goto")) strcpy(s,"goto");
        else if (!strcmp(s,"if")) strcpy(s,"if");
        else if (!strcmp(s,"mod")) strcpy(s,"%");
        else if (!strcmp(s,"nil")) strcpy(s,"'\0'");
        else if (!strcmp(s,"not")) strcpy(s,"!");
        else if (!strcmp(s,"procedure")) strcpy(s,"\n");
        else if (!strcmp(s,"record")) strcpy(s,"struct");
        else if (!strcmp(s,"repeat")) strcpy(s,"do");
       else if (!strcmp(s,"until")) strcpy(s,"while");
        else if (!strcmp(s,"while")) strcpy(s,"while");
        else if (!strcmp(s,"writeln")) strcpy(s,"printf");
        else if (!strcmp(s,"write")) strcpy(s,"printf");
        else if (!strcmp(s,"real")) strcpy(s,"float");
        else if (!strcmp(s,"integer")) strcpy(s,"int");
        else if (!strcmp(s,"char")) strcpy(s,"char");
}
```

Uma versão melhorada desta função usaria uma busca binária em um vetor que armazena palavras-chaves do Pascal, eliminando desta maneira o número de asserções **strcmp( )** que pudessem tornar esta rotina muito lenta se a lista fosse expandida. Algumas palavras (tais como **program**) não têm equivalente em C, e, neste caso, são substituídas por uma nova linha. Uma série de caracteres nula não é usada, porque está reservada para indicar o fim do arquivo.

Aqui está o programa de tradução completo

```
#include "stdio.h"
#include "ctype.h"

#define OP      1
#define KEYWORD 2
#define VAR     3
#define NUMBER  4

char token[80];
int tok_type;
int tok;

char s[10000];
char *prog;

main(argc,argv)
int argc;
char *argv[];
{
        FILE *fp1,*fp2;
        char *p;
        prog = s;

        if((fp1=fopen(argv[1],"r"))==0)  {
                printf("arquivo inacessivel para entrada\n");
                exit(0);
        }

        if((fp2=fopen(argv[2],"w"))==0)  {
                printf("arquivo inacessivel para saida\n");
                exit(0);
        }

        while((*prog=getc(fp1))!=EOF)
                prog++;

        *prog='\0';
        prog=s;
        for(;;)  {
                get_token();
                if(!*token)  break;
                p=token;
                while(*p)  putc(*p++,fp2);
                putc(' ',fp2);
        }
        fclose(fp1);fclose(fp2);
}

get_token()
{
        register char *temp;
```

```
tok_type=0; tok=0;
temp=token;

if(*prog=='\r') {
        *temp++='\r';
        *temp++='\n';
        *temp='\0';
        prog+=2;
        return;
}

if(*prog=='\0') {
        *temp='\0';
        return;
}
while(iswhite(*prog))  ++prog;

if(*prog=='=') {
        prog++;
        strcpy(token,"==");
        return;
}

if(*prog==':') {
        prog++;
        if(*prog=='=')
        {
                *temp++ = '=';
                prog++;
        }
        else *temp++=':';

        *temp='\0';
        return;
}

if(is_in(*prog,"'")) {
        *temp++="'"; prog++;
        while(!is_in(*prog,"'")) *temp++=*prog++;
        *temp="'"; temp++; *temp='\0'; prog++;
        return;
}

if(is_in(*prog,"+-*;.,/^%()")) {
        tok_type=OP;
        *temp=*prog;
        prog++;
        if(*temp=='.') *temp=' ';
        temp++;
        *temp=0;
        return;
```

```
        }

        if(isalpha(*prog))  {
                while(isalpha(*prog)) *temp++=*prog++;
                *temp='\0';
                translate(token);
                return;
        }

        if(isdigit(*prog))  {
                while(!isdelim(*prog))  *temp++=*prog++;
                tok_type=NUMBER;
                *temp=0;
                return;
        }
        prog++;
}

iswhite(c)
char c;
{
        if(c==' ' || c==9) return 1;

        return 0;
}

isdelim(c)
char c;
{
        if(is_in(c," +-/*^%()") || c==9 || c=='\r' || c==0)
                return 1;
        return 0;
}

is_in(ch,s)
char ch,*s;
{
        while(*s) if(*s++==ch) return 1;
        return 0;
}

translate(s)
char *s;
{
        if(!strcmp(s,"and")) strcpy(s,"&&");
        else if (!strcmp(s,"begin")) strcpy(s,"{");
        else if (!strcmp(s,"case")) strcpy(s,"switch");
        else if (!strcmp(s,"div")) strcpy(s,"/");
        else if (!strcmp(s,"do")) strcpy(s,"do");
        else if (!strcmp(s,"else")) strcpy(s,"else");
        else if (!strcmp(s,"end")) strcpy(s,"}");
```

```
        else if (!strcmp(s,"forward")) strcpy(s,"extern");
        else if (!strcmp(s,"for")) strcpy(s,"for");
       else if (!strcmp(s,"function")) strcpy(s,"\n");
        else if (!strcmp(s,"goto")) strcpy(s,"goto");
        else if (!strcmp(s,"if")) strcpy(s,"if");
        else if (!strcmp(s,"mod")) strcpy(s,"%");
        else if (!strcmp(s,"nil")) strcpy(s,"'\0'");
        else if (!strcmp(s,"not")) strcpy(s,"!");
        else if (!strcmp(s,"procedure")) strcpy(s,"\n");
        else if (!strcmp(s,"record")) strcpy(s,"struct");
        else if (!strcmp(s,"repeat")) strcpy(s,"do");
        else if (!strcmp(s,"until")) strcpy(s,"while");
        else if (!strcmp(s,"while")) strcpy(s,"while");
        else if (!strcmp(s,"writeln")) strcpy(s,"printf");
        else if (!strcmp(s,"write")) strcpy(s,"printf");
        else if (!strcmp(s,"real")) strcpy(s,"float");
        else if (!strcmp(s,"integer")) strcpy(s,"int");
        else if (!strcmp(s,"char")) strcpy(s,"char");
}
```

Na verdade, o programa de Computação Assistencial em Pascal para C lê o código-fonte inteiro em Pascal, segura o bastão por um tempo nesta atividade, executa todas as que pode e escreve numa versão em C. Exceto para algumas trocas de operadores, a função-padrão **strcmp( )** deleta um bastão na tabela de traduções, e **strcpy( )** converte isto em um bastão próprio em C. Para ver como este programa simples faz para traduzir do Pascal para C mais facilmente, rode este programa em Pascal pelo programa tradutor.

```
program test (input,output);
procedure tom (x: integer);
begin
        writeln(x*2);
end;

function ken (w: real): real;
begin
        if w=100 writeln('w is 100 inside ken');
        ken:= w/3.1415;
end;

begin

        writeln('Oi, pessoal');
        tom(25);
        writeln(ken(10));
end.
```

O pseudocódigo em C é

```
program test ( input , output ) ;

tom ( x : int ) ;
{
printf ( x * 2 ) ;
} ;


ken ( w : float ) : float ;
{
if w == 100 printf ( "w is 100 inside ken" ) ;
ken = w / 3.1415;
} ;

{
printf ( "Oi, pessoal" ) ;
tom ( 25 ) ;
printf ( ken ( 10 ) ) ;
}
```

Como você pode ver, este não é um código em C, mas você poupou muita digitação. Tudo que você precisa fazer é editar uma linha por vez para corrigir as diferenças.

CONVERTENDO DE BASIC PARA C

A tarefa de converter Basic para C é muito mais difícil do que a de converter Pascal para C. Basic não é uma linguagem estruturada, e possui pouca similaridade com C, o que significa dizer que não só não tem um conjunto completo de controle de estruturas mas, mais do que isso, não tem sub-rotinas separadas com variáveis locais. A tarefa de tradução apresenta muitos detalhes. Geralmente requer conhecimento profundo tanto de Basic como de C, e conhecimento do programa, porque na verdade você terá que reescrever o programa em C e usar a versão em Basic como referência. Por causa da complexidade da tarefa, esta seção verá algumas das mais incômodas traduções e oferecerá sugestões.

CONVERTENDO LAÇOS EM BASIC EM LAÇOS EM C

O laço **for/next** é a única forma de controle de laços na maioria das versões em

Basic. A forma geral dos laços **for/next** em Basic e do laço **for** em C é geralmente a mesma; existe iniciação, condição de teste e incremento. O laço **for** do C é muito mais sofisticado e flexível do que o **for/next** do Basic, mas quando traduzir de Basic para C, isto não terá importância. Por exemplo, o laço **for/next** de Basic

```
10      for x=1 to 100
20      print x
30      next
```

traduz-se em C como

```
for(x=1; x<=100; ++x) printf("%d\n",x);
```

Como você pode ver, a conversão é essencialmente uma substituição de um pelo outro. O truque, na verdade, para converter o laço **for/next** é ter certeza de que o laço de controle de variável não é modificado dentro do laço. Por exemplo, em

```
10      for count=10 to 0 step -1
20      input A
30      print A*count
40      if A=100 then count=10
50      next
```

a asserção **if/then** na linha 40 poderia fazer o laço sair antes. Para traduzir isto corretamente para código em C, você deve permitir para esta contingência da forma:

```
for (count=10; count)0; --count)
        a=getnum();
        printf("%d\n", a*count);
        if(a==100) break;
}
```

Alguns tipos de Basic têm um laço **while/wend** disponível. Neste caso, você usaria um **while** em C e sua tradução seria direta. Se o Basic que você está usando não tem o laço **while/wend** ou se você não quis usá-lo, seu trabalho será maior porque você precisa reconhecer um *laço construído*, usando asserções **goto.** Este também será o caso se

um tipo de laço **do/while** for necessário em Basic. Estes tipos de tradução tornam-se um pesadelo porque você precisa entender realmente como o código trabalha no sentido de reconhecer o laço e traduzi-lo para um controle de estrutura de laço interno em C.

Depois de encontrar um laço, existe uma maneira fácil de dizer se o laço construído em Basic deve ser traduzido para o **while** ou **do/while** em C. Lembre-se de que um laço **do/while** *sempre é executado ao menos uma vez* porque a condição do laço é checada no fim do laço, ao passo que o laço **while** pode ou não executar porque sua checagem de condição é testada no início. Assim, você precisa olhar cuidadosamente para cada laço construído e determinar onde o teste do laço se aplica. Por exemplo, o código em Basic

```
100   s=s+1
200   q=s/3.1415
300   print q;
400   if s<100 then goto 100
```

é realmente um laço **do/while** disfarçado porque sempre executará pelo menos uma vez. Depois da linha 100 ser executada, as linhas de 200 a 400 serão executadas. Se S é menor do que 100, o programa voltará à linha 100. Em C, isso poderia ser

```
do {
      s++;
      q=s/3.1415;
      printf("%f",q);
} while (s<100);
```

No exemplo seguinte em Basic, o teste do laço é executado no início do laço, e assim exige o uso do laço **while:**

```
10    a=1
20    if a>100 then goto 80
30    print a
40    input b
50    a=a+b
60    goto 20
70    print "done"
```

O equivalente em C é

```
a=1;
while(a<=100) {
      printf("%d\n",a);
      b=getnum();
      a=a+b;
}
printf("done");
```

Evite colocar qualquer iniciação dentro do laço acidentalmente. Neste exemplo, a asserção **a = 1** tem de estar fora do laço porque é uma condição de iniciação e não pertence ao laço propriamente dito.

CONVERTENDO A ASSERÇÃO IF/THEN/ELSE

A maioria dos Basic's tem uma única linha de asserção **IF/THEN/ELSE.** Isto implica que quando um bloco deve ser executado baseado no resultado de um **IF,** de um **GOTO** ou **GOSUB.** Você precisa reconhecer esta situação, porque irá querer estruturar o código em asserções **if/else** próprias de C quando você traduz. Como um exemplo, considere o seguinte fragmento de código em Basic:

```
120 IF T <100 THEM GOTO 500
130 Y=W
140 T=10
150 INPUT A$

.
500 REM RESUME DISK READS
```

Para encontrar um bloco **IF** em um programa em Basic, a condição do **IF** precisa ser atribuída na negativa: não pode ser a condição que você quer entrar em um bloco **IF,** mas preferivelmente a condição que cause um salto em torno dele. Este é um dos piores problemas em Basic. Usar rotinas **GOSUB** como fim de um **IF** ou de um **ELSE** facilita o problema um pouco, mas não totalmente. Se o fragmento de código Basic fosse traduzido diretamente para C, ficaria assim:

```
if (t<500);
else {
        y=w;
        t=10;
        gets(a);
}
/* resume disk reads */
```

Você pode agora ver um problema, o final do **if** é realmente uma asserção vazia. A única maneira de resolver é reescrever a condição **if** e assim tornar-se verdadeira, o bloco de código é pedido. O fragmento de código então torna-se

```
if(t>=500) {
        y=w;
        t=10;
        gets(a);
}
/* resume disk reads */
```

Agora, o código, escrito em C, faz sentido.

As diferenças entre a maneira de usar o **IF/THEN** de Basic e a maneira de usar **if** em C demonstra que a linguagem de programação normalmente escolhe o método de resolver problemas. A maioria das pessoas acha a forma positiva do **if** mais natural de usar do que a forma negativa.

CRIANDO FUNÇÕES EM C DE PROGRAMAS EM BASIC

Um dos motivos da dificuldade de traduzir Basic em C é que C não suporta sub-rotinas com variáveis locais. Isto significa que uma tradução literal de um programa em Basic para o C produziria uma função **main( )** e só algumas poucas funções. Uma tradução melhor criaria um programa em C com uma função **main( )** e muitas outras funções. Fazer isto requer conhecimento do programa e um bom olho para ler o código. Porém, aqui estão algumas regras para orientá-lo.

Primeiro, transforme todas as sub-rotinas **GOSUB** em funções. Observe também funções parecidas nas quais só variáveis foram trocadas, e coloque-as em uma função com parâmetros. Por exemplo, esse código em Basic tem duas sub-rotinas – uma na linha 100 e outra na linha 200:

```
10    A=10
20    B=20
30    GOSUB100
40    PRINT A,B
50    C=20
60    D=30
70    GOSUB200
80    PRINT C,D
90    END
100   A=A*B
110   B=A/B
120   RETURN
200   C=C*D
210   D=C/D
220   RETURN
```

Ambas sub-rotinas fazem exatamente a mesma coisa, exceto que elas operam conjuntos separados de variáveis. A tradução deste programa para C tem só uma função que usa parâmetros para evitar haver duas funções dedicadas:

```
main()
{
        int a,b,c,d;

        a=10; b=20;
        f1(&a,&b);
        printf("%d %d\n",a,b);
        c=20; d=30;
        f1(&c,&d);
        printf("%d %d\n",c,d);
}

f1(x,y)
int *x,*y;
{
        *x= *x * (*y);
        *y= *x / *y;
}
```

Esta tradução para C aproxima mais o significado do código do leitor do que a versão Basic faz, o que implica que existem realmente duas funções separadas envolvidas.

A segunda regra é para fazer todos os códigos repetidos em uma função. Em um programa em Basic, as mesmas poucas linhas de código podem ser repetidas. Um

programador freqüentemente faz isto para tornar o código um pouco mais rápido. Porque geralmente uma linguagem complicada, usando função, em oposição a usar código de linha, tem um efeito menor; a melhor documentação pesa mais do que qualquer ganho na velocidade.

LIBERTANDO-SE DE VARIÁVEIS GLOBAIS

Em Basic, todas as variáveis são globais: elas são conhecidas através do programa e podem ser modificadas em qualquer ponto do programa. No processo de tradução, tente converter todas as variáveis globais possíveis em variáveis locais, porque isto torna o programa mais recuperável e livre de "bugs". Quanto mais variáveis globais existem, mais provável que efeitos colaterais ocorram.

Algumas vezes é difícil saber quando mudar uma variável local para uma função. As escolhas mais fáceis são aquelas que controlam contadores em pequenos pedaços de código. Por exemplo, no código

```
10   FOR X=1 TO 10
20   PRINT X
30   NEXT
```

X é usado somente para controlar o laço **FOR/NEXT** e pode, assim, ser feito com variáveis locais da função.

Outro tipo de variável que é candidata a tornar-se local é uma variável temporária. Uma variável temporária armazena um resultado intermediário no cálculo. Variáveis temporárias são difíceis de reconhecer. Por exemplo, a variável C12, vista aqui, armazena um resultado temporário no cálculo

```
 10   INPUT      A,B
 20   GOSUB      100
 30   PRINT      C12
 40   END
100   C12=A*B
110   C12=C12/0.142
120   RETURN
```

O mesmo código em C, com C12 como variável local, seria

```
#include "stdio.h"

main()
{
        float a,b,f1();

        scanf("%f%f",&a,&b);
        printf("%f",f1(a,b));
}

float f1(a,b)
float a,b;
{
        float c12;

        c12=a*b;
        printf("n1=%f",c12);
        c12/=0.142;
        printf("n2=%f",c12);
        return c12;
}
```

Lembre-se de que é sempre melhor ter o menor número de variáveis globais possível, assim é importante encontrar bons candidatos para variáveis locais.

CONCLUSÕES FINAIS SOBRE TRADUÇÃO

Embora traduzir programas possa ser a mais tediosa de todas as tarefas de programação, é também uma das mais comuns. Um bom método é entender a maneira como o programa que você está traduzindo trabalha, e aprender a usá-lo. Uma vez que você sabe como o programa funciona, ele fica mais fácil de ser recodificado; você sabe se sua versão está funcionando corretamente. Além disso, quando você conhece o programa que está traduzindo, o trabalho torna-se mais interessante porque não é só um simples processo de substituição de símbolos.

O próximo capítulo inclui um caso especial de tradução. Nesta situação, você traduzirá um programa em C que escreveu com um compilador, em um programa que compilará e rodará com um compilador C diferente. Embora isto pareça fácil, é freqüentemente a mais difícil tarefa da tradução.

Capítulo **11**

EFICIÊNCIA, PORTABILIDADE E DEPURAÇÃO

A habilidade em escrever programas que utilizam eficientemente os recursos do sistema, livres de erros, e que podem ser facilmente transportáveis para outros computadores é a característica de um programador profissional. É, também, esta habilidade que transforma a ciência de computação na "arte da ciência de computação", porque são poucas as técnicas disponíveis para assegurar o sucesso. Este capítulo apresenta alguns dos métodos através dos quais eficiência, depuração de programas e portabilidade podem ser alcançadas.

EFICIÊNCIA

Quando relacionado a um programa de computação, o termo eficiência refere-se à velocidade de execução de programas, ao uso que faz dos recursos do sistema, ou a ambos. Os recursos do sistema incluem RAM, espaço em disco, papel da impressora e basicamente qualquer coisa que possa ser alocada ou usada. Um programa ser ou não eficiente é um julgamento subjetivo – depende da situação. Considere um programa que usa 47K de RAM de código executável, 2 megabytes de espaço em disco, e que tem um tempo médio de execução de 7 minutos. Se for um programa de ordenação rodando em um Apple II, então o programa provavelmente não é muito eficiente. No entanto, se for um programa de previsão de tempo rodando em um computador Cray, então o programa provavelmente é muito eficiente.

Outra consideração a fazer quando se fala em eficiência é que otimizar um determinado aspecto de um programa irá freqüentemente degenerar outro. Por exemplo, fazer um programa tornar-se mais rápido geralmente significa torná-lo maior, como quando você usa instruções seqüenciais em vez de chamadas de funções para acelerar as seqüências de chamada. Além disso, tornar mais eficiente o uso do espaço em disco

através da compactação de dados tornará o acesso a disco invariavelmente mais lento. Estes e outros tipos de tentativas podem ser frustrantes – especialmente para o usuário final, que não consegue ver como uma coisa pode afetar a outra.

Sabendo destes problemas, você deve se perguntar como então pode-se discutir a eficiência de programas. Existem realmente alguns procedimentos de programação que sempre são eficientes – ou pelo menos são mais eficientes do que outros. Existem, também, algumas poucas técnicas que tornam os programas, além de mais rápidos, menores.

OS OPERADORES INCREMENTO E DECREMENTO

Discussões sobre a eficiência do uso da linguagem C quase sempre começam por considerar os operadores incremento e decremento. Lembre-se de que o operador incremento + + incrementa seu argumento de um, e o operador decremento – – decrementa seu argumento de um. O operador incremento essencialmente reproduz este tipo de assertiva:

```
x=x+1;
```

e o operador decremento reproduz a assertiva do tipo:

```
x=x-1;
```

Além da vantagem óbvia de reduzir o número de teclas digitadas, os operadores incremento e decremento contêm uma outra grande vantagem: eles são executados mais rapidamente e necessitam de menos RAM do que suas correspondentes declarações para a maioria dos compiladores C. Isto é devido à maneira como o código-objeto é gerado pelo compilador. Por exemplo, se você utilizar uma simples e imaginária linguagem assembly próxima da maioria das linguagens assembly encontradas para microprocessadores, a declaração

```
x=x+1;
```

gerará a seqüência de código

```
move        A,x     ; carrega o valor de x da memória no acumulador
move        B,1     ; coloca 1 no registrador B
add         B       ; soma B ao acumulador
store       x       ; armazena o novo valor em x
```

No entanto, se você usar o operador incremento, o código seguinte produzirá:

```
move A,x    ; armazena o valor de x da memória no acumulador
incr A      ; incrementa (o que é x) de 1
store x     ; armazena o novo valor de x
```

Aqui, uma instrução inteira foi eliminada, o que significa que o código será executado mais rapidamente e será menor.

Alguns compiladores C reconhecem automaticamente expressões tais como x=x+1 e para produzir um código-objeto melhor fornecerão uma saída com código como se tivesse sido escrito x++. Este processo é chamado de *otimização*. No entanto, você não deve contar com isto muito freqüentemente, e se tiver que transportar seu código para um computador novo usando um compilador diferente, você deve usar os operadores incremento e decremento explicitamente.

PONTEIROS VERSUS VETORES INDEXADOS

Outra técnica que produz tanto códigos mais rápidos como menores é substituir ponteiros aritméticos por vetores indexados. Para entender porque isto pode fazer diferença, dê uma olhada nos seguintes fragmentos de código, que fazem a mesma coisa.

Aritmética de Ponteiros

```
p=vetor;
for(;;) {
    a=*(p++);

}
```

Indexação de Vetores

```
for(;;) {
    a=vetor[t++];

}
```

Com o método de ponteiros, após **p** ter sido carregado com o endereço de **vetor,** por exemplo em um registrador indexador (assim como o SI do processador 8086), só um incremento precisa ser executado a cada vez que o laço é executado. Porém, a versão do vetor indexado força o programa a calcular o vetor indexado baseado no valor de **t** a cada

passagem através do laço. A disparidade entre aritmética de ponteiros e indexação de vetores cresce à medida que índices múltiplos são usados: aritmética de ponteiros pode usar adição simples, ao passo que cada índice requer sua própria seqüência de instruções.

No entanto, como precaução, você poderia querer usar vetores indexados quando o índice fosse derivado de uma fórmula complexa e quando o uso da aritmética de ponteiros pudesse deixar confuso o sentido do programa. É normalmente melhor degenerar a performance um pouco do que sacrificar a compreensão.

USO DE FUNÇÕES

Lembre-se sempre de que o uso de funções independentes com variáveis locais forma a base da programação estruturada. Funções são os blocos construídos de programas em C, e elas são um dos aspectos mais fortes de C. Não deixe que nada descrito aqui nesta seção seja mal-entendido. Agora que você foi avisado, precisa saber de alguns aspectos das funções C e seus efeitos na velocidade e tamanho do seu código.

O primeiro e mais importante: C é uma *linguagem orientada pela pilha:* todas as variáveis locais e parâmetros para função usam a pilha para armazenamento temporário. Quando uma função é chamada, o endereço de retorno da rotina de chamada é colocado na pilha. Isto permite que a sub-rotina retorne à posição de onde foi chamada. Quando a função retorna, esse endereço – assim como todos os parâmetros e variáveis locais – precisam ser retirados da pilha. O processo de colocar esta informação na pilha é geralmente referenciado como uma *seqüência de chamada,* e o processo de retirar a informação da pilha é chamado de *seqüência de retirada.* Estas seqüências levam tempo – e às vezes um tempo significativo.

Para entender como uma chamada de função pode retardar seu programa, veja os exemplos dos dois códigos apresentados aqui.

Versão 1

```
for(x=1;x<100;++x) {
        t=compute(x);
}

float compute(q)
int q;
{
        float t;

        t=abs(sin(q)/100/3.1416);
        return t;
}
```

Versão 2

```
for(x=1;x<100;++x) {
        t=abs(sin(q)/100/3.1416);
}
```

Embora cada laço execute a mesma função, a versão 2 é muito mais rápida porque o cabeçalho das seqüências de chamada e retorno foram eliminados usando-se código em linha. Para entender de que maneira boa parte do tempo é eliminado, estude o seguinte pseudocódigo em assembly, que mostra as seqüências de chamada e retorno para a função **compute()**. O código real usado depende de como o compilador está implementado e que processo está sendo usado, mas geralmente segue o mesmo padrão deste exemplo.

```
; seqüência de chamada

move A, x                    ; coloca o valor de x no acumulador
push A                       ;
call compute                 ; a instrução de chamada coloca o end. de retorno na pilha
      .
      .
; seqüência de retorno
; o valor retornado da função deve ser colocado
; em um registrador – nós usaremos B

move B, stack–1              ; pega variável temporária t
return                       ; retorna a rotina de chamada

; rotina de chamada então faz o seguinte

pop A                        ; limpa parâmetros usados na chamada
```

Usar a função **compute( )** dentro do laço faz com que as seqüências de chamada e retorno sejam executadas cem vezes. Se você quer escrever códigos realmente rápidos, então usar **compute( )** dentro do laço não é o meio correto.

Neste momento, você pode pensar que deveria escrever um programa que tivesse somente algumas funções maiores e assim rodaria mais rápido. Na maioria dos casos, no entanto, a pequena diferença de tempo não será significativa, e a perda de estrutura será considerável. Mas existe outro problema. Trocar funções que são usadas por muitas rotinas com código em linha tornará seu programa muito grande, porque o mesmo código será duplicado várias vezes. Tenha em mente que sub-rotinas foram inventadas principalmente como uma maneira de fazer uso mais eficiente da memória. Uma regra geral de tornar o seu programa mais rápido significa fazê-lo maior, enquanto torná-lo menor significa fazê-lo mais lento.

Finalmente, só faz sentido usar código em linha em vez de chamada de função,

quando velocidade é de absoluta prioridade. De outra forma, o uso livre de funções é definitivamente recomendado.

BUSCANDO OS LIMITES

Em alguns círculos, C tem ganho a reputação de ser uma linguagem criptografada e de difícil leitura. Esta reputação é devido inteiramente a programadores puristas que sempre procuram escrever programas muito eficientes. Em função de C permitir expressões muito complexas para serem escritas em uma linha, que podem às vezes tornar o programa um pouco mais rápido, alguns programas em C são difíceis de decifrar. Um programa intensamente otimizado é algumas vezes necessário, mas na maioria dos casos o ganho é pequeno e reduz-se consideravelmente a manutenciabilidade. Em todos os casos, você deve ter uma boa razão quando diminuir a clareza do seu código.

PROGRAMAS PORTÁTEIS

É comum que um programa escrito para uma máquina seja transportado para outro computador com um processador ou sistema operacional, ou ambos diferentes. Este processo é chamado *portabilidade* e pode ser ou muito mais fácil ou extremamente difícil, dependendo da maneira como o programa foi escrito originalmente. Um programa é portátil se ele foi facilmente transportado. Um programa não é muito portátil se contiver numerosos itens *dependentes da máquina* – fragmentos de código que trabalharão somente com um sistema operacional e processador específico. C foi desenvolvido para permitir portabilidade de código, mas ainda assim requer cuidados, atenção aos detalhes, e freqüentemente sacrifício da eficiência máxima para realmente alcançar um código portátil. Nesta seção você examinará algumas áreas com problemas específicos e conhecerá algumas soluções.

USANDO # DEFINE

Talvez a maneira mais simples de fazer programas portáteis é tornar *todos* os sistemas ou processador dependentes de um "número mágico" em uma diretiva de

substituição macro **#define.** Estes "números mágicos" incluem tamanho dos buffers para acesso a disco, comandos especiais para tela e teclado, informação sobre alocação de memória, e qualquer outra coisa que tenha ao menos uma chance de ser trocada quando o programa for transportado. Se você fizer um número mágico em diretivas **#define,** estas "definições" não só deixarão claro o número mágico para a pessoa que está fazendo o transporte, mas também simplificarão a edição; seus valores precisam ser trocados somente uma vez em vez de alterar todo o programa.

Por exemplo, aqui estão duas funções que usam **read( )** e **write( )** para acessar informação em um arquivo em disco.

```
f1();
{
      write(fd,buf,128)
}

f2()
{
      read(fd,buf,128)
}
```

O problema é que o número **128** é difícil de codificar tanto em **read( )** como em **write( )**. Isto pode ser aceito por um sistema operacional mas não por outro. A melhor maneira de codificar é vista aqui

```
# define buf_size 128

f1()
{
      write(fd,buf,buf_size);
}

f2()
{
      read(fd,buf,buf_size);
}
```

Neste caso, só o **#define** será trocado e todas as referências a **buf–size** serão automaticamente corrigidas. Esta versão não é somente mais fácil de alterar, mas também evita muitos erros de edição. Lembre-se de que provavelmente haverá muitas referências

a **buf-size** em um programa real, assim o ganho em portabilidade é freqüentemente significativo.

DEPENDÊNCIAS DO SISTEMA OPERACIONAL

Virtualmente todos os programas comerciais contêm códigos que são específicos a um sistema operacional. Por exemplo, um projeto de planilha deve fazer uso da memória de vídeo do IBM-PC para permitir trocas rápidas de telas, ou um pacote gráfico deve usar comandos gráficos especiais que são somente aplicados a um sistema operacional. Algumas dependências do sistema operacional são necessárias para a rapidez dos programas comercialmente viáveis. Porém, não existe razão em fixar códigos além do necessário.

Como foi sugerido anteriormente, funções de arquivo em disco podem algumas vezes conter dependências de máquina implícitas. As funções **real( )** e **write( )** encontradas na biblioteca-padrão, por exemplo, podem funcionar com vários tamanhos de buffers, mas um sistema operacional deve requerer um múltiplo par de alguns números para operar mais eficientemente. Assim, um tamanho de buffer de 128 deve estar bom para o CP/M 2.2 mas não deve ser aceito para o MS-DOS. Neste caso, o tamanho do buffer deve ser discutido anteriormente.

Quando você precisa usar chamadas ao sistema para acessar o sistema operacional é melhor fazê-las todas em uma função-mestre e assim você só precisa trocar esta função para acomodar um novo sistema operacional e pode deixar o resto do código intacto. Por exemplo, se chamadas ao sistema forem necessárias para limpar a tela e o fim de linha, e para localizar o cursor em uma coordenada X, Y, então você deve criar uma função-mestre como **op-sys-call( )**, vista aqui:

```
op_sys_call(op,x,y)
char op
int x,y
{
     switch (op) {
          case1:
                    clear_screen();
                    break;
          case2:
                    clear_eol();
                    break;
```

```
                case3:
                        goto xy(x,y);
                        break;
                }
        }
```

Só o código que trata as funções reais precisa ser alterado, deixando a interface comum intacta.

DEPURANDO

Plageando Thomas Edson, programar é 10% de inspiração e 90% de depuração. Bons programadores são normalmente bons depuradores. Embora você, provavelmente, tenha grandes talentos de depuração, deve atentar para certos tipos de "bugs" que podem ocorrer facilmente quando está usando C.

ERROS DE ORDEM DE PROCESSAMENTO

Quando os operadores incremento ou decremento são usados em programas escritos em C, a ordem na qual as operações são executadas faz diferença no caso destes operadores precederem ou seguirem a variável. Por exemplo, os dois comandos

Versão 1	**Versão 2**
y=10;	y=10;
x=y++;	x=++y;

não são iguais. O primeiro atribui o valor 10 a **x** e então incrementa **y**. O segundo incrementa **y** para 11 e então atribui o valor 11 a **x**. Assim, na versão 1, **x** contém 10; na versão 2, **x** contém 11. A regra é que os operadores incremento e decremento ocorrem antes de outras operações se precederem o operando; de outra forma, eles ocorrem depois. Um erro de ordem de processamento ocorre quando trocas são feitas a um comando já existente. Por exemplo, você deve inserir o comando

```
x = *p++;
```

que atribui o valor apontado por **p** a **x** e então incrementa o ponteiro **p**.

Porém, imagine que mais tarde você decida que **x** precisa realmente do valor apontado por **p** vezes o valor apontado por **p.** Para fazer isso, você deve reescrever o comando para

```
x = *p++ * (*p);
```

No entanto, isto não funciona, porque **p** já foi incrementado. A solução apropriada é escrever

```
x = *p* (*p++);
```

Erros como este podem se tornar difíceis de encontrar. Pode haver chances, tais como em laços ou rotinas que não rodem corretamente, que faltem uma para fechar. Se você tiver qualquer dúvida sobre um comando, recodifique-o para ter mais certeza.

PROBLEMAS COM PONTEIROS

Um erro comum em programas em C é o uso impróprio de ponteiros. Problemas com ponteiros recaem em duas categorias gerais. O não-entendimento do sentido e dos operadores de ponteiros, e o uso acidental de ponteiros inválidos. Para solucionar o primeiro tipo de problema, você precisa entender a linguagem C; para solucionar o segundo, precisa verificar sempre a validade do ponteiro antes de usá-lo.

O programa seguinte ilustra um erro típico de ponteiro que os programadores de C cometem:

```
main        /* este programa esta errado */
{
     char *p;
     char *alloc();
     *p=alloc(100); /* esta linha esta errada */
     gets(p);
     printf(p);
}
```

Este programa parecerá "crachado", provavelmente estragando o sistema operacional. Ele não funcionará porque o endereço retornado por **alloc( )** *não* foi atribuído

a **p,** mas ao endereço de memória apontado por **p,** que é completamente desconhecido neste caso. Para corrigir este programa, você deve substituir

```
p=alloc(100); /* esta linha está correta*/
```

na linha incorreta.

O programa possui um outro erro mais sutil: não existe uma checagem em tempo de execução do endereço retornado por **alloc( )**. Lembre-se, se a memória está cheia, **alloc( )** retornará 0, que nunca é um ponteiro válido em C. O problema ocasionado por este tipo de "bug" é difícil de encontrar porque ocorre raramente, só quando um pedido de alocação falha. Prevenir é a melhor maneira de trabalhar com isto. Aqui está uma versão correta do programa, que inclui uma checagem de validade de ponteiro

```
main()     /* esse programa agora esta correto */
{
     char *p;
     char *alloc();
     p=alloc(100);  /* isso esta correto */

     if (p==0) {
          printf("fora da memória\n");
          exit(0);
     }
     gets(p);
     printf(p);
}
```

Ponteiros "selvagens" são extremamente difíceis de acompanhar. Se você estiver fazendo atribuições a um ponteiro variável que não contém um endereço de ponteiro válido, seu programa deve aparecer correto para a função algumas vezes, mas inválido em outras. Estatisticamente, quanto menor o seu programa, mais chance ele tem de rodar corretamente, mesmo com um ponteiro perdido, pois pouca memória é usada. Conforme seu programa cresce, falhas tornam-se mais comuns, mas ao tentar depurar você pensará sobre os acréscimos recentes ou trocas em seu programa, não em erros de ponteiros. Assim, você provavelmente irá procurar o "bug" no lugar errado.

Uma indicação de um problema de ponteiro é que os erros parecerão não-convencionais. Seu programa poderá funcionar corretamente uma vez e errado outra. Algumas vezes outras variáveis conterão "lixo" por uma razão inexplicável. Se estes

problemas ocorrerem, cheque seus ponteiros. Como regra de procedimento, você deve checar todos os ponteiros quando os "bugs" começarem a aparecer.

Embora ponteiros possam ser problemáticos, eles são, também, um dos aspectos mais úteis e poderosos da linguagem C, e valem a pena apesar de todos os problemas que possam causar. Faça um esforço no início para aprender a usá-los corretamente.

Um ponto final a lembrar sobre ponteiros é que você precisa iniciá-los antes deles serem usados. Isto parece simples o bastante, mas muitos bons programadores de C ainda caem nesta armadilha, ocasionalmente. Por exemplo, o seguinte fragmento de código será um desastre porque você não sabe para onde **x** está apontando:

```
int *x;

*x=100;
```

Assumir um valor a uma porção desconhecida provavelmente destruirá alguma coisa de valor – talvez outro código ou dado do seu programa.

REDEFININDO FUNÇÕES

Você pode – mas não deve – chamar suas funções pelos mesmos nomes na biblioteca-padrão C. A maioria dos compiladores usará sua função em vez de uma da biblioteca, causando problemas diretos ou indiretos.

Aqui está um exemplo de um problema direto causado por redefinir uma função de biblioteca.

```
main()
{
        FILE *fp;
        char big[1000];
        init_array(big);
        if((fp=open("name","r"))==-1)  {
                printf("arquivo inacessivel\n");
                exit(0);
        }

.
.
.
```

```
init_array(p)
char *p;
{
        register int t;
        for(t=0;t<1000;t++,p++)  {
                *p=t;
                if(t<100) open(p);
}
open(p)
char *p;
{
        *p='0';
}
```

Este programa não rodará ou fará coisas esquisitas. A função-padrão **open()** foi redefinida no programa para atribuir o caractere 0 a certos elementos do vetor. Não existe nada a fazer com **open( )** usada no mesmo programa para abrir um arquivo em disco.

Uma versão ainda pior do problema de redefinição aparece quando uma função-padrão da biblioteca é redefinida, mas a função-padrão não é usada diretamente no programa – é usada indiretamente por outra função-padrão. Considere o programa seguinte

```
char text[1000];
main()
{
     int x;
     scanf ("%d",&x);
     .
     .
     .
}

getc(p) /* retorna caractere do vetor */
{
     return text[p];
}
.
.
.
```

Este programa não trabalhará com a maioria dos compiladores porque **scanf( )**, uma função-padrão do C, provavelmente chamará **getc( )**, uma função-padrão do C que foi

redefinida no programa. O problema pode ser difícil de encontrar, porque você não tem nenhuma indicação de ter criado um efeito colateral. Irá simplesmente parecer que **scanf( )** não funciona corretamente.

A única maneira de evitar tais problemas é nunca dar a uma função que você tenha escrito o mesmo nome de uma função da biblioteca-padrão. Se você tem dúvidas, acrescente suas iniciais ao início do nome, como em **cs-getc( )** em vez de **getc( )**.

BIZARROS ERROS DE SINTAXE

Ocasionalmente, você verá um erro de sintaxe que não conseguirá entender ou mesmo reconhecer como erro. Mesmo o compilador C algumas vezes possui um "bug" que provoca o registro de falsos erros. A única solução é refazer o seu código. Outros erros não-usuais simplesmente requerem uma busca mais detalhada para serem encontrados.

Um erro particularmente estranho ocorrerá quando você tentar compilar este código:

```
main()
{
      char  *p,  *myfunc();  /*  myfunc()  retorna um  ponteiro  a
                             caracter */
      .
      .
      .
}
myfunc()
{
      .
      .
      .
}
```

A maioria dos compiladores apresentará uma mensagem de erro do tipo *function redefined* (função redefinida) e apontará para **myfunc( )**. Como pode ser? Não existem duas funções **myfunc( )**. A resposta é que você declarou que **myfunc( )** retorna um ponteiro a caractere dentro de **main( )**. Esta declaração provoca um entrada na tabela de símbolos que é feita com esta informação. Quando o compilador encontra **myfunc( )** mais

tarde dentro do programa, não existe nenhuma indicação que **myfunc( )** retornará qualquer outra coisa além de um inteiro, o tipo-padrão. Assim, você "redefiniu" a função. O programa correto seria como segue.

```
main()
{
      char  *p,  *myfunc();  /*  myfunc()  retorna  um  ponteiro  a
                             caractere */
      .
      .
      .
}
char *myfunc()
{
      .
      .
      .
}
```

O código a seguir gerará outro erro de sintaxe que é dificilmente compreendido:

```
main()/* este programa tem um erro de sintaxe */
{
      func1();
}

func1();
{
      printf("esta é func1 \n");
}
```

O erro aqui é o ponto e vírgula depois da declaração de **func1( )**. O compilador verá a função como uma declaração fora de outra função, o que caracteriza um erro; porém, os compiladores registrarão o erro de diferentes maneiras. Muitos compiladores apresentarão a mensagem de erro como *bad declaration sintax* (declaração com sintaxe errada), apontando para a primeira chave aberta após **func1( )**. Pelo fato de você estar acostumado a ver pontos e vírgulas depois de declarações, encontrará dificuldade para enxergar de onde provém esta mensagem de erro.

ERROS DO TIPO "UM FORA"

Agora você já sabe que todas as indexações em C começam com 0. Um erro comum envolve o uso do laço **for** para acessar elementos de um vetor. Considere o seguinte programa, o qual inicia um vetor de 100 inteiros:

```
main()          /* este programa não funcionará */
{
        int x, num[100];
        for(x=1;x<=100; ++x) num[x]=x;
}
```

O laço **for** neste programa está errado por duas razões. Primeiro, ele não inicia **num[0],** o primeiro elemento do vetor **num. num[99]** é o último elemento do vetor e o laço vai até 100. A forma correta de escrever este programa é

```
main()          /* este está certo */
{
        int x, num[100];
        for(x=0;x<=100; ++x) num[x]=x;
}
```

Lembre-se: um vetor de 100 tem elementos de 0 a 99.

ERROS DE LIMITE

A linguagem C e muitas funções da biblioteca-padrão têm pouca ou nenhuma checagem de limites. Por exemplo, é possível escrever por cima de vetores, arquivos em disco, e (através de declarações de ponteiros) variáveis. Estas coisas normalmente não

ocorrem, mas quando elas acontecem, pode ser muito difícil determinar o sintoma que causam.

Por exemplo, o programa seguinte lê uma série de caracteres do teclado e os exibe na tela.

```
main()

{

      int  var1;
      char s[10];
      int  var2;
      var1=10; var2=10;
      get_string(s);
      printf("%s %d %s",s,var1,var2);

}

get_string(string)

char *string;

{
      register  int t;
      printf("entre com 20 caracteres\n");
      for(t=0;t<20; ++t) {
            *s++=getchar();

      }
}
```

Neste caso não existem erros de código. No entanto, um erro indireto aparece quando **get-string( )** é chamada com **s.** A variável **s** está declarada com 10 caracteres de comprimento, mas **get-string( )** lerá 20 caracteres, provocando uma escrita por cima de **s.**

O problema real é que ao mesmo tempo que **s** exibe todos os 20 caracteres corretamente, tanto **var1** como **var2** não possuirão valores corretos. Todos os compiladores C precisam alocar uma região de memória – normalmente uma região da pilha – para variáveis locais. As variáveis **var1** e **var2,** e **s,** serão alocadas como mostra a Figura 11.1.

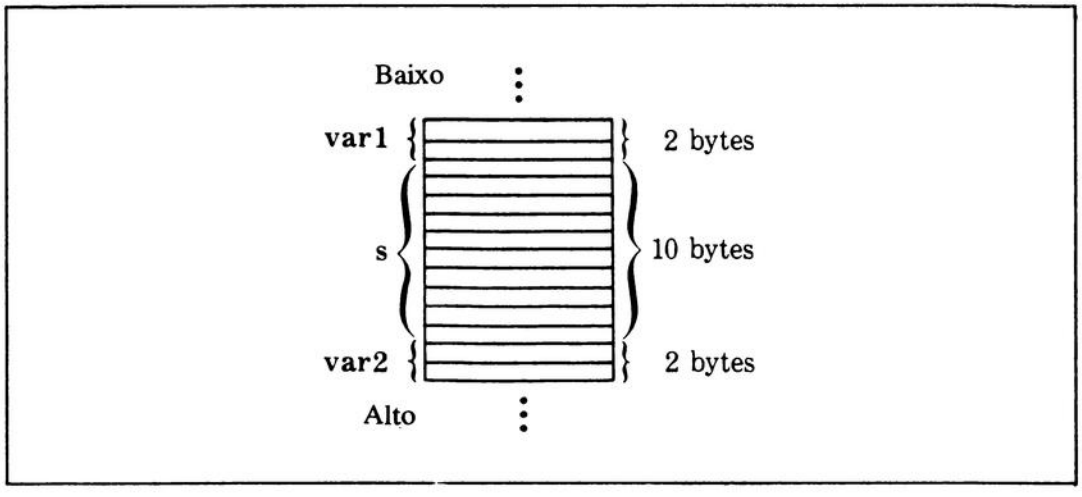


Figura 11.1. As variáveis **var1, var2** s na memória.

Seu compilador C pode inverter a ordem de **var1** e **var2,** mas elas ainda irão projetar-se sobre **s.** Quando se escreve sobre **s,** a informação adicional é colocada na área reservada para **var2,** destruindo qualquer conteúdo anterior. Assim, em vez de imprimir o número 10 para ambas as variáveis inteiras, o programa exibirá alguma coisa a mais para aquela variável destruída pela reescrita de **s.** Isto fará com que você veja o problema de maneira errada.

OMISSÃO DE DECLARAÇÃO DE FUNÇÃO

Sempre que uma função retornar um tipo válido diferente de um inteiro, ela precisará ser declarada para fazer isto dentro de qualquer outra função que a utilize. Por exemplo, este programa multiplica dois números em ponto flutuante.

```
main()    /* isto está errado */
{
     float     x,y;
     scanf("%f%f",&x,&y);
     printf("%f",mul(x,y));
}

float mul(a,b)
```

```
flaot a,b;
{

      return a*b;

}
```

Embora **main( )** espere um valor inteiro de **mul( ), mul( )** retorna um número em ponto flutuante. Você obterá respostas sem significado, porque **main( )** só copiará dois bytes dos oito necessários para um **float.**

Para corrigir este programa, declare **mul( )** em **main( )** como a seguir:

```
main()     /*    este está correto */

{

      float x,y,mul();
      scanf("%f%f",&x,&y);
      printf("%f",mul(x,y));

}

float mul(a,b)
float a,b
{

      return a*b;

}
```

Acrescentar **mul( )** à lista de declaração **float( )** diz para **main( )** que ela deve esperar um valor em ponto flutuante ser retornado de **mul( ).**

ERROS DE ARGUMENTOS DE CHAMADA

Você deve se certificar ao ligar o tipo de argumento que uma função espera com o tipo que a ela é passado. Por exemplo, lembre-se de que **scanf( )** espera receber o **endereço** do seu argumento, e não seu valor. Isto significa que você deve chamar **scanf( )** com argumentos que utilizam o operador **&.** O código seguinte está errado:

```
int  x;
char string[10];
scanf("%d%s",x,string);
```

Este código está correto:

```
scanf("%d%s",&x,string);
```

Lembre-se: seqüência de caracteres já passam seus endereços para funções, assim você não deve usar o operador **&**, neste caso.

Outro erro comum é esquecer que funções em C não podem modificar seus argumentos. Se for necessário modificar um argumento para uma função, você deve passar o endereço do argumento para a função e usar referências de ponteiros para acessar o argumento.

Se os parâmetros formais da função são do tipo **float**, então você precisa passar variáveis com ponto flutuante para a função. Por exemplo, o programa seguinte não funcionará corretamente:

```
main()    /* este programa esta errado */
{
     int  x,y;
     scanf("%d%d",x,y);
     printf("%d",div(x,y));
}


float div(a,b)
float a,b;
{
     return a/b;
}
```

Você não pode usar uma função com ponto flutuante como **div( )** para retornar um valor inteiro, e não pode esperar que **div( )** opere corretamente – ela espera números em ponto flutuante, não inteiros. Você deve lembrar-se de que uma atribuição pode sempre ser usada para trocar um tipo por outro, se necessário.

TEORIA GERAL DE DEPURAÇÃO

Cada um tem um método diferente de programar e depurar. Porém, algumas técnicas têm provado ser melhores do que outras. No caso da depuração, testes incrementais são considerados os de menor custo e o mais eficiente método, embora possa parecer que retarda o processo de desenvolvimento.

Testar por partes é simplesmente o processo de sempre possuir código funcionando. Assim que for possível obter uma parte do seu programa funcionando, você deverá fazê-lo, testando esta parte completamente. Na medida em que aumente seu programa, continue a testar as novas partes, bem como a maneira como elas se ligam ao código já depurado. Desta maneira você pode certificar-se que qualquer possibilidade de "bugs" está concentrada em uma pequena área de código.

A teoria dos testes incrementais está normalmente baseada na probabilidade e áreas. Como você sabe, *área* é uma dimensão quadrada. Cada vez que é acrescentado comprimento, você dobra a área. Assim, conforme seu programa cresce, existe uma área *n*-quadrada na qual você deve procurar erros. No processo de depuração, você, como programador, prefere a menor área possível para trabalhar com estes erros. Através de testes incrementais você pode subtrair a área já testada da área total, reduzindo, desta forma, a região que pode conter os erros.

PENSAMENTOS FINAIS

No decorrer deste livro, muitos algoritmos e técnicas foram discutidos, alguns em detalhes. Lembre-se de que a ciência da computação é tanto teórica como empírica. Embora seja fácil ver por que um algoritmo é melhor que outro, é difícil dizer o que faz um programa ter sucesso. Quando ele contém depuração, eficiência e portabilidade, experimentação, fornecerá muitas vezes informação mais facilmente do que poderiam fornecer teorias.

Programar é tanto uma ciência como uma arte. É uma ciência porque você precisa conhecer lógica e entender como e por que os algoritmos funcionam; é uma arte

porque você cria uma entidade completa que é o programa. Como um programador, você possui um dos melhores trabalhos da terra – você caminha na linha entre a arte e a ciência, obtenha o melhor de ambos.

ÍNDICE ANALÍTICO

NÃO PERCA SEU TEMPO DIGITANDO!

Como você provavelmente já sabe, copiar os programas do livro pode tornar-se um problema. Você pode facilmente cometer erros ou acidentalmente omitir uma linha. O processo consome tempo e "time is money!"

Por essas razões, nós, da Bix Informática Ltda.-ME, estamos oferecendo um disquete compatível com IBM-PC, contendo todos os programas-fontes deste livro.

O preço do disquete é de apenas 3.0 OTN's e você poderá obtê-lo, enviando o cupom abaixo, com cheque nominal, à

bix Informática Ltda. – ME
a/c
Cláudio Gaiger Silveira
Rua Urimonduba, 66/23
Itaim-Bibi, São Paulo
CEP 04530

ou entrando em contato através do telefone (011) 280-0194.

corte aqui e envie este cupom e pagamento em envelope selado

SIM. Envie-me o número de disquetes do C Avançado assinalado.

☐ 1 disquete

_____ disquetes

Por favor preencha os seguintes dados:

_____
Nome completo

_____
Firma Telefone

_____
Endereço para correspondência

_____
Cidade/Estado CEP

bix Informática Ltda.-ME

Por favor, para efeito de pesquisa, assinale o item apropriado.

☐ Engenheiro
- ☐ Elétrico
- ☐ Mecânico
- ☐ Outro

☐ Matemático
☐ Arquiteto
☐ Estudante
☐ Outro _____

_____
Assinatura

O pagamento deve acompanhar o pedido. Aguarde de 2 (duas) a 6 (seis) semanas para o recebimento.

Impresso na Prol editora gráfica ltda.
03043 Rua Martim Burchard, 246
Brás - São Paulo - SP
Fone: (011) 270-4388 (PABX)
com filmes fornecidos pelo Editor.

Composição e Arte-Final:
J A G Composição Editorial e Artes Gráficas Ltda.
Praça F. Roosevelt, 208 - 8º andar
Tel. (011) 255-5694 - São Paulo

OUTROS LIVROS NA ÁREA

LINGUAGENS

Ayres – Aplicações Estatísticas em Basic

Botelho – Basic Prático – Intermediário e Avançado

Carvalho – Assembler para o TK 90X e TK95

Carvalho – Basic para o TK 90X e TK95

Fox/Fox – Iniciação ao Basic

Gaiger – Linguagem C – Guia do Operador

Gottfried – Programação com Basic

Hehl – Fortran IV

Hehl – Linguagem de Programação Estruturada: Fortran 77

Hester – Open Access – Guia do Usuário

Hogan – Forth – Guia do Usuário

Maldonado – Introdução ao Fortran 77 para Microcomputadores

Medidata – MUMPS – Guia do Usuário

Mottola – Linguagem de Programação Assembly para Apple II - 6502

Newcomer – Cobol Estruturado

Peckham – Manual de Basic para o Apple II

Renzetti – Turbo Pascal – Guia do Operador

Schildt – Linguagem C – Guia do Usuário

Siragusa – Basic Estruturado

Wood – Turbo Pascal – Guia do Usuário

0-07-450400-2