OSBORNE/ McGRAW-HILL

LINGUAGEM C

GUIA DO USUÁRIO

McGraw-Hill

Herbert Schildt

LINGUAGEM C

GUIA DO USUÁRIO

LINGUAGEM C
GUIA DO USUÁRIO

Tradução
Lars Gustav Erik Unonius
Engenheiro Eletrônico

Revisão Técnica
João Hajime Takeda
Analista de Sistemas
Laboratório de Sistemas Integráveis da
Escola Politécnica da USP

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
C Made Easy

Editor: Milton Mira de Assumpção Filho
Coordenadora de Revisão: Daisy Pereira Daniel

Capa: Layout: Cyro Giordano
Arte final: Ademir Aparecido Alves

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

Schildt, Herbert.
S36L Linguagem C : guia do usuário / Herbert Schildt ; tradução Lars Gustav Erik Unonius ; revisão técnica João Hajime Takeda. -- São Paulo : McGraw-Hill, 1986.

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

86-1040 CDD-001.6424

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

A minha esposa Sheryl,
cuja ajuda tornou este livro
possível

NOTA DO REVISOR TÉCNICO

A maioria dos programas presentes neste livro foi revisada, tendo em vista sua adequação aos compiladores disponíveis ao revisor.

Tanto exemplos como respostas foram analisados, e muitos compilados, e corrigidos pelo revisor, procurando dispor ao leitor o mais útil e correto programa, dentro das limitações apresentadas pelo programa do autor.

Os programas foram compilados no compilador

"Microsoft versão 3.00" e
"Microsoft versão 1.04 – Lattice".

Os programas são compatíveis com as funções de biblioteca apresentadas em ambiente UNIX.

A maior parte das alterações introduzidas nos programas destina-se a melhorar a apresentação final no monitor.

A alteração mais significativa reside na introdução da função 'fflush( )' junto às funções de impressão, para que a frase que o programa visa imprimir seja escrita no momento e local corretos.

Muitos compiladores não fazem um esvaziamento automático quando das seguintes instruções, por exemplo:

```
printf("...");
ch= getchar( );
```

Normalmente, para que as séries de caracteres sejam impressas no monitor, faz-se necessário um delimitador (usualmente CR). Se este delimitador não estiver presente, a frase não será escrita. Para forçar a escrita dispomos da função 'fflush( )'.

O compilador Microsoft versão 3.00 faz este esvaziamento automaticamente. Para o compilador versão 1.04, faz-se necessária a introdução da função.

Outra situação em que a função 'fflush( )' se faz útil é quando queremos evitar a entrada de "lixo" pelo teclado. Isto pode ser obtido, fazendo com que todos os caracteres, que estejam esperando para ser lidos pelo programa, sejam descartados. Este é o exato efeito do seguinte comando

```
fflush(stdin);
```

Em muitos pontos foram tomadas providências, mas há ainda vários por introduzir. Fica a critério do leitor melhor adaptar os programas ao compilador e ao formato mais agradável.

SUMÁRIO

INTRODUÇÃO

A finalidade deste livro é ensinar-lhe a linguagem de programação C. Já que programação aprende-se melhor fazendo, recomendamos que você tenha acesso a um compilador C. Existem vários à disposição, para a maioria dos microcomputadores, incluindo alguns excelentes para o IBM PC e seus compatíveis. Os exemplos apresentados neste livro serão corretamente compilados, sem erro, virtualmente em qualquer compilador C. É melhor verificar primeiro o seu manual de usuário, pois poderão ocorrer pequenas variações entre compiladores diferentes.

Este livro assume que você tem algum conhecimento de programação. Você deve conhecer os conceitos gerais de variáveis, declarações de atribuição e elos. Não se preocupe, sua experiência em programação não precisa ser muito extensa.

Pelo fato do BASIC geralmente acompanhar o computador, provavelmente ele lhe é familiar. BASIC tornou-se uma linguagem conhecida para exemplos de programação por ser muito divulgado. Para ajudá-lo a melhor compreender certos aspectos da linguagem C, serão utilizados exemplos de BASIC e C lado a lado, principalmente no início do livro. Algumas vezes, a visão de uma declaração em uma linguagem que se está aprendendo vale muito mais que páginas de explicação. Entretanto, não é necessário o conhecimento de BASIC. Mesmo que você não conheça BASIC, este livro será excelente para aprender a linguagem de programação C.

Caso você ainda não tenha comprado um compilador, é recomendável que adquira um que seja *compatível* com o sistema UNIX, porque a biblioteca de funções que o acompanha é similar à descrita neste livro.

Nos primeiros exemplos, os dados serão colocados nos programas, utilizando a função **getchar( )**, encontrada na maioria dos compiladores, ou **getnum( )**, desenvolvida no texto como uma maneira simples de introduzir números decimais.

Os exemplos deste livro foram compilados e processados utilizando-se o compilador Aztec C para o IBM PC. Os exemplos serão ainda compilados e processados utilizando-se o compilador SuperSoft C, com a exceção de que os exemplos de ponto flutuante poderão necessitar de uma ligeira alteração para acomodar diferenças na implementação com SuperSoft. De maneira geral, qualquer compilador, compatível com UNIX versão 7, poderá compilar e processar os programas deste livro.

CAPÍTULO 1

APRESENTANDO C

C, com freqüência, é denominada uma linguagem de computador de nível médio. *Nível médio* não tem uma conotação negativa: não significa que C é menos poderosa, mais difícil de utilizar ou menos desenvolvida que uma linguagem de alto nível como BASIC ou Pascal; nem significa que C seja similar a uma linguagem de baixo nível, como a linguagem assembly, que não passa de uma representação simbólica do código de máquina, passível de ser lida pelo computador. C é uma linguagem de nível médio porque combina elementos de uma linguagem de alto nível com a funcionalidade do assembly. A Tabela 1.1 mostra os níveis de diversas linguagens de computador, inclusive C.

Uma linguagem de nível médio fornece aos programadores um conjunto mínimo de declarações de controle e manipulação de dados, que eles poderão utilizar para definir construções de alto nível. Em contraste, a linguagem de alto nível é projetada procurando fornecer aos programadores tudo o que eles desejam que seja parte da linguagem. Uma linguagem de baixo nível obriga os programadores a definir explicitamente todas as funções do programa, porque nada é parte da linguagem. Assim, nenhuma das abordagens é melhor do que outra; cada uma tem sua aplicação específica. Pensamos, com freqüência, nas linguagens de nível médio como linguagens de construção, porque o programador primeiro cria as rotinas que vão executar todas as funções necessárias ao programa, para em seguida juntá-las.

C permite – na verdade, precisa – que o programador defina as rotinas para execução dos comandos de alto nível. Essas rotinas são denominadas *funções* e são muito importantes na linguagem C. Facilmente você pode montar uma biblioteca de funções C, para que executem tarefas processadas pelo seu programa. Desta maneira você consegue personalizar C para que atenda a suas necessidades.

Como uma linguagem de nível médio, C manipula os bits, os bytes e os endereços com que o computador funciona. Ao contrário do BASIC, uma linguagem de alto nível que pode operar diretamente em uma série de caracteres, executando diversas funções série, C pode

Tabela 1.1 Níveis das linguagens de computação

Alto Nível	Nível médio	Baixo Nível
Ada	C	Assembler
BASIC	FORTH	
COBOL		
FORTRAN		
Pascal		

operar diretamente nos caracteres. Em BASIC existem declarações intrínsecas para ler e escrever arquivos de discos. Em C, esses procedimentos são executados por funções que não são propriamente parte da linguagem C, mas são fornecidos na biblioteca padrão. Essas funções são rotinas especiais escritas em C e que executam essas operações. Por exemplo, a declaração PRINT do BASIC não tem equivalência direta em C. Existe, entretanto, uma função denominada **printf( )** na biblioteca de funções padronizadas do Compilador C, fornecida pelo fabricante.

C tem suas vantagens. Tem poucas declarações para serem lembradas – apenas 28 palavras chaves. (A versão BASIC do IBM PC tem 159.) Isto significa que os compiladores C podem ser escritos com uma certa facilidade, de modo que normalmente exista um disponível para sua máquina. Uma vez que C opera com os mesmos tipos de dados que o computador, a saída em código do compilador C é eficiente e rápida. Para a maioria das tarefas, C pode ser utilizado em lugar do assembly.

O código C é bastante portátil. *Portabilidade* significa que o software escrito para um tipo de computador pode ser adaptado para outro tipo. Por exemplo, caso um programa escrito para um Apple II possa ser facilmente transferido para um IBM PC, este programa é portátil. A portabilidade é importante quando você planeja utilizar um computador novo com um processador diferente. A maioria dos programas aplicativos somente precisa ser recompilada com um compilador C escrito para o novo processador. Isto permite economizar centenas de horas e dinheiro.

UTILIZAÇÃO DE C

C foi utilizado pela primeira vez para programação de sistemas. *Programação de sistema* refere-se a uma classe de programas que, ou são parte, ou operam em conjunto com o sistema operacional do computador. Programas de sistemas fazem com que o computador seja capaz de executar trabalho útil. Seguem alguns exemplos de programas de sistema que freqüentemente são escritos em C:

- Sistemas operacionais
- Assembler
- Gerenciador de impressão
- Programas
- Interpretadores para comunicação
- Compiladores
- Editores de texto
- Operadores de redes
- Bancos de dados
- Utilitários

Existem muitas razões por que C é utilizado para programação de sistemas. Os programas de sistema, com freqüência, devem ser processados com rapidez. Os programas compilados pelos compiladores C podem ser processados quase que com a mesma rapidez que aqueles escritos em assembly. No passado, grande parte do software de sistema era escrito em linguagem assembly, porque nenhuma das linguagens de computação disponíveis conseguiam criar programas com suficiente rapidez de processamento. Escrever em linguagem assembly é um trabalho árduo e cansativo. Uma vez que a codificação em C é obtida com maior rapidez do que a codificação em assembly, a utilização de C reduz em muito os custos.

Outro motivo pelo qual C é utilizado para programação de sistemas deve-se ao fato de ser uma linguagem de programador. Os programadores profissionais sentem-se atraídos pelo C porque não apresenta restrições e manipula com facilidade bits, bytes e endereços. O programador de sistemas necessita do controle direto de C sobre as funções de E/S e gerenciamento da memória. C permite ainda que o programa reflita a personalidade do programador.

Pelo fato de os programadores gostarem de programar em C, esta tem sido utilizada nos últimos anos como uma linguagem de programação de uso geral. C é lido com facilidade. Uma vez estando familiarizado com C, você consegue seguir o fluxo preciso e a lógica de um programa, e verificar facilmente a operação geral das sub-rotinas. A listagem do programa C é clara; em comparação, a linguagem do tipo BASIC parece confusa e embaralhada. Talvez a melhor explicação de C tornar-se uma linguagem de uso geral se deva ao fato da facilidade e tranqüilidade que oferece ao usuário.

C COMO UMA LINGUAGEM ESTRUTURADA

C é uma linguagem estruturada, assim como o são Ada e Pascal. BASIC, COBOL e FORTRAN são linguagens não-estruturadas. A principal característica de uma linguagem estruturada é a utilização de blocos. Um *bloco* é um conjunto de instruções que estão ligadas logicamente. Por exemplo, imagine uma instrução IF que, sendo bem-sucedida, executará cinco instruções discretas. Se estas instruções puderem ser agrupadas e facilmente referenciadas, formarão um bloco.

Uma linguagem estruturada fornece uma variedade de possibilidades de programação. Ela sustenta o conceito de sub-rotinas com variáveis locais. Uma *variável local* nada mais é do que uma variável conhecida apenas pela sub-rotina em que está definida. Uma linguagem estruturada sustenta também diversas construções em elo, como **while**, **do-while** e **for**. (A utilização do **goto** é proibida ou desencorajada e não é a forma comum de controle do programa como em BASIC ou FORTRAN.) Uma linguagem estruturada permite que se utilize sub-rotinas compiladas separadamente sem que pertençam ao programa propriamente dito. Significa que você pode criar uma biblioteca de sub-rotinas, formada por funções úteis e testadas, que podem ser acessadas por qualquer programa que você escreva. Uma linguagem estruturada permite que você insira instruções e não exige o conceito restrito de campo como em FORTRAN.

As linguagens estruturadas são mais modernas, ao passo que as não-estruturadas são mais antigas. De fato, uma das características de uma linguagem de computação antiga é ela não ser estruturada. Em virtude de sua clareza, as linguagens estruturadas não são apenas mais fáceis de serem utilizadas em uma programação como também mais fáceis de serem mantidas.

Mesmo que você consiga pensar em uma linguagem não-estruturada que satisfaça as exigências de uma linguagem estruturada (como BASIC avançado), a linguagem estruturada está baseada na compartimentalização de funções e dados: isto é, a redução de cada tarefa em sua própria sub-rotina ou bloco de código. Ao aprender a linguagem de programação C, a diferença entre uma linguagem estruturada e uma linguagem não-estruturada ficará bastante clara.

INTERPRETADORES VERSUS COMPILADORES

Os termos *interpretador* e *compilador* se referem à maneira como o programa é executado. Em tese, qualquer linguagem de programação pode ser compilada ou interpretada, mas algumas linguagens normalmente são executadas de um ou outro modo. Porém, a maneira como um programa é executado não é definida pela linguagem em que é escrita. Os interpretadores e compiladores são apenas programas sofisticados que operam sobre o código-fonte de seu programa. *Código-fonte* é o texto do programa que você escreve.

O interpretador lê o código-fonte de seu programa uma linha por vez e executa as instruções contidas nessa linha. O compilador lê o programa inteiro e converte-o em *código-objeto*, que consiste em uma tradução do código-fonte em uma forma que pode ser diretamente executada pelo computador. Código-objeto é conhecido também como *código binário* e *código de máquina.* Uma vez que o programa está compilado, uma linha de código-fonte não apresenta mais significado para a execução de seu programa.

Por exemplo, BASIC normalmente é interpretado e C geralmente é compilado. O interpretador deve estar presente sempre que o programa é executado. Em BASIC o interpretador deve ser executado primeiro, o programa carregado, para em seguida ser digitado **RUN**, cada vez que desejarmos utilizar o programa. O compilador, por outro lado, converte o programa em código-objeto que pode ser executado diretamente pelo computador. Já que o compilador traduz o programa uma única vez, em C, a única coisa necessária é executar diretamente o programa, normalmente digitando seu nome, depois de compilado.

Programas compilados são processados com muito maior rapidez do que os interpretados. Porém, o processo de compilação em si leva muito mais tempo. Isto, porém, é compensado pela economia de tempo feita durante a utilização do programa. A única situação em que isto não é verdade ocorre quando o programa é muito curto – digamos, menos do que 50 linhas – e não utiliza nenhum elo.

Além da vantagem com a velocidade, os compiladores protegem o código-fonte contra roubo e alterações não autorizadas. O código compilado não tem semelhança com o código-fonte, e é esta a razão pela qual os compiladores são utilizados, quase que exclusivamente, pelas empresas de software comerciais.

Dois termos que você vai ver com freqüência neste livro e no manual do compilador C são *tempo de compilação* e *tempo de processamento.* Tempo de compilação refere-se aos eventos que ocorrem durante o processo de compilação. Tempo de processamento refere-se aos eventos que ocorrem enquanto o *programa* está sendo executado. Infelizmente, você os verá utilizados com freqüência em associação com a palavra *error* (erro), como em *compile-time errors* e *run-time errors.*

CAPÍTULO 2

UMA VISÃO GERAL DE C

Antes de compreender qualquer informação específica sobre C, devemos ver como se parece um programa C, em comparação ao equivalente em BASIC. Este capítulo vai nos mostrar os fundamentos de C; os capítulos seguintes vão explicar, em detalhes, todos os aspectos da linguagem de programação C.

A Figura 2.1 apresenta o primeiro programa normalmente escrito pelos principiantes em C ou em BASIC. Este programa apenas imprime a palavra **HELLO** na tela do computador, seguido por uma combinação retorno-de-carro/avanço-de-linha.

```
main()
{
printf("HELLO\n");
}
```

```
10 PRINT "HELLO"
20 END
```

Figura 2.1 Versão em C e em BASIC de um programa que imprime **HELLO**

FUNÇÕES EM C

A linguagem C baseia-se no conceito de blocos construtivos. Os blocos construtivos são denominados *funções*. Um programa C é uma coletânea de uma ou mais funções. Para escrever um programa, primeiro você deve criar as funções para depois juntá-las.

A função é uma sub-rotina, formada por uma ou mais declarações C, que executa uma ou mais tarefas. Em uma codificação C, bem escrita, cada função executa apenas uma tarefa.

Cada função tem um nome e uma lista de argumentos que vai receber. De modo geral, a função pode receber o nome que você desejar, com exceção de **main**, reservado para a função que inicia a execução do programa.

Este livro utiliza uma convenção, que já se tornou um padrão quando se escreve a respeito de C, ao denominar uma função, esta terá parênteses após seu nome. Por exemplo, se o nome de uma função é **max**, será escrito como **max( )**. Esta anotação ajudará a distinguir nomes de variáveis de nomes de funções.

No programa HELLO da Figura 2.1 tanto **main( )** como **printf( )** são funções. Como foi afirmado anteriormente, **main( )** é a primeira função executada quando se inicia o processamento de seu programa. A função **printf( )**, que não é propriamente parte da linguagem C, é uma sub-rotina escrita em C. Esse tipo de sub-rotina é normalmente desenvolvida pelo projetista do compilador e é parte da biblioteca padrão de C. A função **printf( )** faz com que seu argumento seja impresso na tela do computador. No programa HELLO, o argumento é a série dentro do parênteses, **"HELLO\n"**. O símbolo **\n** é utilizado pelo C para indicar uma nova linha, isto é, uma combinação retorno-de-carro/avanço-de-linha.

A FORMA GERAL DAS FUNÇÕES C

O programa HELLO apresenta a forma geral de uma função C. O programa inicia com **main( )**. Em seguida, uma chave de abertura significa o início da função, seguida por qualquer declaração que faça parte da mesma. Neste programa, a única declaração é **printf( )**. A chave de fechamento sinaliza o fim da função. Aqui indica também o fim do programa. A forma geral de uma função é

nome-da-função (lista de argumentos)
declaração da lista de argumentos
{ *chave de abertura indicando início da função*
.
. *corpo da função*
.
} *chave de fechamento encerrando a função*

Como vemos, a primeira coisa que uma função necessita é um nome. Dentro dos parênteses que seguem o nome da função está a *lista de argumentos*. Na linha imediatamente inferior está a *declaração da lista de argumentos*, informando ao compilador o tipo de variável que deve esperar. Em seguida, as chaves envolvem o corpo da função. O *corpo da função* é formado pelas declarações C que definem o que a função faz. C tem uma declaração **return** explícita, que obriga o retorno de uma função. Como nenhum **return** é encontrado, a função pára automaticamente a execução e retorna, ao encontrar a chave final. Isso difere da combinação, em BASIC, **GOSUB-RETURN** porque o BASIC requer o **RETURN** para saber quando retornar de uma sub-rotina.

A FUNÇÃO main( )

A função **main( )** é especial por ser a primeira a ser chamada quando seu programa é executado. Significa o início de seu programa. Ao contrário de um programa BASIC, que inicia na linha de menor número ou no "topo" do programa, um programa C é iniciado com uma chamada à função **main( )**. A função **main( )** pode estar localizada em qualquer parte de seu programa, apesar de ser geralmente a primeira função, por uma questão de clareza. É *obrigatória* a existência de **main( )** em algum ponto do programa, para que o compilador C consiga determinar onde iniciar a execução.

A função **main( )** é idêntica a qualquer outra função C, com exceção da chave de fechamento de **main( )**, que significa o fim do programa. Quando esta chave é encontrada, o programa retorna ao sistema operacional.

Pode haver apenas um **main( )** no programa. Se houvesse mais do que um, seu programa não saberia onde iniciar a execução. A maioria dos compiladores detecta este tipo de erro, mesmo antes de iniciar o estágio de execução.

ARGUMENTOS DE FUNÇÃO

No programa HELLO, a função **printf( )** tem um argumento: a série que será impressa na tela do computador. As funções em C poderão ter de zero a diversos argumentos. (O limite superior é determinado pelo compilador que você está utilizando.) *Argumento* é um valor passado para a função. Quando se define uma função, as variáveis que recebem valores de argumentos também devem ser declaradas. São chamadas *parâmetros formais* da função. Por exemplo, a função abaixo vai fornecer o produto dos dois argumentos inteiros. A declaração **return** fornece o produto para a rotina chamadora.

```
/*
** funcao mul
*/
mul(x,y)
int x,y;  /* aqui x e y sao declaradas como variaveis
             inteiras */
{
return(x * y); /* retorna o produto dos argumentos */
}
```

Toda vez que **mul( )** for chamado multiplicará os valores de **x** e **y**. Lembre-se, porém, são apenas as variáveis operacionais da função que recebem os valores que você estipula ao chamar a função.

A Figura 2.2 apresenta um programa curto que utiliza a função **mul( )**. Este programa vai imprimir dois números na tela: **2** e **2340**. As variáveis **x**, **y**, **j** e **k** não são modificadas pela chamada à função **mul( )**. De fato, **x** e **y** de **main( )** não têm relação com **x** e **y** em **mul( )**.

```
main()
{
int x,y,j,k,p;

x = 1;
y = 2;
p = mul(x,y);
printf("%d\n",p); /* imprime p em decimal com printf */

j = 234;
k = 10;
p = mul(k,j);
printf("%d\n",p);
}

/*
** funcao mul
*/
mul(x,y)
int x,y; /* aqui x e y sao declaradas como variaveis
            inteiras */
{
return(x * y); /* retorna o produto dos argumentos */
}
```

Figura 2.2 Um programa utilizando a função **mul( )**

A função **mul( )** vai multiplicar os valores de **x** e **y** bem como os de **j** e **k**. Ao chamar uma função, os argumentos podem ser constantes, como no programa HELLO, ou variáveis como no exemplo **mul( )**. Ao criar a função **mul( )** com dois argumentos, você declarou que as variáveis do argumento são **x** e **y**. Estes são os parâmetros formais da função; contêm a informação que você insere ao chamar a função. C copia o valor da constante ou da variável utilizada como argumento da função, na variável que a função declarou em sua lista de argumentos. Ao contrário de algumas outras linguagens de computação, C não coloca nenhuma informação de volta nos argumentos de função.

```
main()
{
printf("%d\n",123);
}
```

```
10 PRINT 123
20 END
```

Figura 2.3 A versão C e a BASIC de um programa que imprime 123

Nas funções C, os argumentos são sempre separados por vírgulas. Neste livro, o termo *lista de argumentos* refere-se aos argumentos separados por vírgula. A lista de argumentos de **mult( )** é **x, y**.

A FUNÇÃO printf( )

Todos os programas deste livro, utilizados como exemplos, e que resultam em saída para o console, utilizarão a função **printf( )**. É uma função de saída para console de propósitos múltiplos, fornecido com o compilador C. No programa HELLO você já viu como imprimir uma série de caracteres na tela do computador. O programa da Figura 2.3 vai imprimir o número 123 na tela. O equivalente em BASIC é fornecido para comparação.

Neste programa, **printf( )** é chamado com dois argumentos. O primeiro argumento, "**%d**", informa **printf( )** como tratar o segundo argumento, **123**. O sinal percentual informa **printf( )** que o caractere seguinte é um comando de controle de formatação, indicando como os dados devem ser impressos. O **d** significa que os dados que seguem – neste caso o número 123 – devem ser apresentados como um número decimal.

A forma geral de **printf( )** é

printf(*"série de controle"***,***lista de argumentos***)**

Na função **printf( )**, a série de controle contém os comandos de formatação que informam a **printf( )** como apresentar os demais argumentos na tela e, ainda, quantos outros argumentos existem. Lembre-se de que cada argumento, da lista de argumentos, é separado por uma vírgula. **printf( )** permite uma variedade de comandos de formatação, como é indicado na Tabela 2.1.

Tabela 2.1 Códigos de controle de formatação de **printf( )**

Código printf( )	Formato
%c	Caractere simples
%d	Decimal
%e	Notação Científica
%f	Ponto flutuante
%g	Utiliza %e ou %f, dependendo de qual for mais curto
%o	Octal
%s	Série de caracteres
%u	Decimal sem sinal
%x	Hexadecimal

A série de controle poderá conter um ou ambos, entre dois tipos de dados: caracteres para serem impressos ou instruções para que sejam apresentados os argumentos subseqüentes

da lista de argumentos. Comandos de formatação poderão estar inseridos em qualquer ponto da primeira série de caracteres. Quando chamamos **printf( )**, a série de controle é varrida por **printf( )**. Todos os caracteres normais são simplesmente impressos na tela. Quando encontrar um comando de formatação, **printf( )** vai lembrar de utilizá-lo quando imprimir o argumento apropriado. Comandos de formatação e argumentos são combinados da esquerda para a direita. A quantidade de comandos de formatação existentes na série de controle informa quantos argumentos subseqüentes **printf( )** deve esperar encontrar.

Se quisermos imprimir o caractere % encontrado na série de controle, devemos utilizar dois sinais percentuais em seguida, isto é, %%.

Tabela 2.2 Exemplos da função **printf( )**

Série de controle e lista de argumentos	**Apresentação resultante**
("%s %d", "isto é uma *string*", 100);	isto é uma *string* 100
("isto é uma *string* %d",100);	isto é uma *string* 100
("o número %d é decimal, %f é flutuante",10,110.789);	o número 10 é decimal, 110.789 é flutuante
("%c%s%d–%x", 'a',"número em decimal e hexa: ",10,10);	a número em decimal e hexa: 10–A
("%s", "HELLO\n");	HELLO

Existem duas coisas a ser lembradas a respeito da utilização de **printf( )** para apresentar dados de série e caracteres. Primeiro, caracteres individuais que utilizam o comando de formatação %c devem ser envolvidos por apóstrofe; por exemplo, **'c'**. Segundo, série de caracteres que utilizam comando de formatação %s são envolvidos por aspas; por exemplo, **"Isto é uma string"**. A Tabela 2.2 mostra a função **printf( )** em ação.

É *obrigatória* a existência, na série de controle, do mesmo número de argumentos e comandos de formatação. Os comandos de formatação podem incluir, ainda, modificadores que definem casas decimais e, no caso de séries, espaços em branco antes ou depois da série. Esses e outros detalhes a respeito de **printf( )** serão explicados no Capítulo 5.

VARIÁVEIS EM C

Apesar de grande parte do Capítulo 3 discutir as variáveis e sua utilização, existem algumas coisas a respeito delas que você deveria saber agora. Os nomes das variáveis em C podem ter um ou vários caracteres de comprimento. O compilador C que você está utilizando determina o comprimento máximo; entretanto, por uma questão de segurança, assuma que terão ao menos seis caracteres. Os nomes das variáveis podem começar com qualquer letra do alfabeto ou o caractere de sublinhar. Em seguida, poderá haver uma letra, um número ou caractere de sublinhar.

O caractere de sublinhar pode ser utilizado para ressaltar a leitura de um nome de variável, como em **first_name**. Ao contrário da maioria dos dialetos de BASIC, letras maiúsculas ou minúsculas são diferentes quando utilizados em uma variável; isto é, para C, **contar** e **CONTAR** são nomes diferentes. Alguns exemplos de nomes de variáveis aceitáveis são **first, last, Addr1, top_of_file, name23, _temp, t, s23e3**, e assim por diante.

Não é permitida a utilização das palavras-chaves de C como nomes de variáveis. *Palavras-chaves* são aquelas que formam a linguagem de programação C. Suas variáveis não devem ainda receber os mesmos nomes de outras funções C. Apesar de suas rotinas terem precedência sobre outras rotinas e variáveis de sistema, dar às variáveis o mesmo nome que uma função C padronizada poderá criar confusão para você e alguns compiladores. Além destas duas restrições, a prática da boa programação estabelece que deverão ser utilizados nomes para as variáveis que reflitam o seu significado ou a sua utilização.

Em C, como em todas as linguagens de programação, existem diversos tipos de variáveis. Por exemplo, todos os tipos de BASIC têm variável série e de ponto flutuante, e alguns têm variáveis inteiras. Como é mostrado na Tabela 2.3, C apresenta sete tipos de dados inerentes, que apresentam uma palavra-chave C equivalente. Além destes, poderá ser criado um grupo de variáveis denominado *estrutura.*

As variáveis conhecidas por todas as funções de seu programa são denominadas *variáveis globais*. As variáveis conhecidas apenas pela função que as utiliza são chamadas *variáveis locais.*

Todas as variáveis deverão ser declaradas antes de serem utilizadas, em C. O procedimento de declaração informa, além do nome da variável, o *tipo* de dado ao compilador. Primeiro você deve especificar o tipo de variável desejado para em seguida fornecer a lista das variáveis que devem ser do mesmo tipo. Por exemplo, para declarar como inteiros as variáveis **first, last** e **middle**, deveria ser utilizada a seguinte declaração:

```
int first, last, middle;
```

As declarações de variáveis, como todas outras declarações em C, terminam com ponto-e-vírgula. Lembre-se de que o termo **int** é a palavra-chave de C para dados inteiros.

Tabela 2.3 Tipos de dados inerentes de C e suas palavras-chaves

Tipos de dados	**Palavra-chave C equivalente**
charactere	char
inteiro	int
inteiro curto	short int
inteiro longo	long int
inteiro sem sinal	unsigned int
ponto flutuante	float
ponto flutuante dupla precisão	double

As variáveis podem ser declaradas em qualquer ponto de um programa C. Porém, elas normalmente são declaradas no início de uma função, logo após a chave de abertura. Por exemplo, o que segue declara uma variável de cada tipo:

```
sample()
{
          int      count;
          char     c;
          float    value;
short     int      top;
long      int      eof_counter;
          double   pay_out;
unsigned  int      u;

        .
        .
        . /* codigo do corpo da funcao */
        .

}
```

Neste exemplo, as variáveis declaradas serão conhecidas apenas para a função **sample( )**; são *locais* para esta função. Se desejar utilizar uma variável durante todo o programa (em outras palavras, uma variável global), ela deverá ser declarada *fora* de qualquer função. Por exemplo, caso você esteja escrevendo um programa que utiliza a variável **line_count** pelo programa todo, ela deve ser declarada *antes* que seja definido **main( )**:

```
int line_count;

main()
{
          .
          .
          .
          .
          .
}
```

PALAVRAS-CHAVES DE C

C tem 28 palavras-chaves que não podem ser utilizadas como variáveis ou nomes de função. Essas palavras, ao serem combinadas com a sintaxe formal de C, formam a linguagem de programação C. As palavras-chaves estão relacionadas na Figura 2.4. C exige que todas as palavras-chaves tenham letras minúsculas. Por exemplo, **RETURN** *não* será reconhecida como a palavra-chave **return**.

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

Figura 2.4 Lista de palavras-chaves

PONTO-E-VÍRGULAS, CHAVES, COMENTÁRIOS E POSIÇÃO

Você pode estar pensando por que são tantas as declarações que terminam em ponto-e-vírgula. Em C, o ponto-e-vírgula é um *terminador* da declaração, isto é, toda declaração individual deve obrigatoriamente terminar por ponto-e-vírgula. Indica o fim de uma entidade lógica. (Tenha cuidado se você conhece Pascal; no Pascal o ponto-e-vírgula é um *separador* de declarações.)

Lembre-se de que um bloco é um conjunto de declarações conectadas logicamente, entre uma chave de abertura e de fechamento. Se você considerar um bloco como um grupo de declarações com um ponto-e-vírgula após cada declaração, faz sentido o fato de o bloco como um todo não estar seguido por um ponto-e-vírgula isolado.

Ao contrário do BASIC, C não reconhece o final da linha como um terminador. Isto significa que C não apresenta nenhuma restrição quanto à posição, permitindo desta maneira que se agrupe declarações, para maior clareza, conforme é indicado pelos dois fragmentos de código, equivalentes:

```
x = y;

y = y + 1;

mul(x,y);
```

como um grupo de três linhas, é o mesmo que

```
x = y;    y = y + 1;    mul(x,y);
```

Os comentários em C podem ser colocados em qualquer local do programa e estão entre dois sinais. A marca inicial de um comentário é /* e a marca final de um comentário é */. O acréscimo de comentários ao programa HELLO poderia fazer com que ficasse conforme a Figura 2.5.

```
main()
{
/* este programa imprime a palavra hello no monitor
   do computador */

printf("HELLO\n");

} /* este e' o fim do programa */
```

Figura 2.5 O programa HELLO com comentários

PRÁTICAS DE DENTEAÇÃO

Como você deve ter observado nos exemplos anteriores, algumas declarações são denteadas. A linguagem C pode ter uma forma livre porque não importa em que posição da linha você coloca uma declaração.

Entretanto, um estilo de denteação aceitável, e comum, foi desenvolvido durante diversos anos, permitindo que se tenha programas legíveis. Este livro segue esse estilo, e recomendamos que você faça o mesmo. Utilizando esse estilo, você faz uma denteação depois de cada chave de abertura, e retorna para a chave de nível anterior. Existem determinadas declarações que permitem denteações adicionais, que serão expostas posteriormente.

A BIBLIOTECA C PADRÃO

Ao discutir o programa HELLO da Figura 2.1, foi mencionado que **printf( )** havia sido escrito pelo projetista do compilador. A função **printf( )** nem sequer é parte da linguagem C, mas você vai utilizá-la em quase todo o programa C que escrever. Qual é a sua origem? Veio da *biblioteca C padrão.*

Todos os compiladores C têm uma biblioteca C padrão que fornece funções que executam as tarefas mais comuns. Os projetistas de seu compilador C já escreveram muitas das funções de uso geral que você vai utilizar. Quando você utiliza uma função que não foi previamente identificada em seu programa, o compilador C "lembra" o nome da função: parte do compilador encontra a função faltante e acrescenta-a ao seu código-objeto. A parte do compilador que executa esse processo denomina-se *encadeador*, e o processo denomina-se *encadeamento*. Alguns compiladores C têm seus encadeadores próprios, e outros utilizam o encadeador padrão fornecido junto com o sistema operacional.

O processo de encadeamento adiciona códigos que já foram compilados ao seu programa. As funções mantidas na biblioteca estão em *formato relocável*. Significa que os endereços de memória para as diversas instruções em código de máquina não foram definidos de forma absoluta, tendo sido mantidas apenas informações defasadas. Quando o seu programa se liga

com as funções da biblioteca padronizada, estas defasagens de memória são utilizadas para criar os endereços reais utilizados. Existem diversos manuais e livros técnicos que explicam esse processo com maiores detalhes. Porém, não há necessidade de um entendimento maior do processo de relocação para poder programar em C.

A frase *biblioteca C padrão* é otimista. Não existe nada na linguagem C que defina *exatamente* quais as funções que devem estar em uma biblioteca ou como vão operar. Os projetistas de cada compilador C é que definem isto. Entretanto, a maioria dos compiladores C comerciais utilizam as *funções de biblioteca padronizadas* UNIX. C foi desenvolvido originalmente no sistema operacional UNIX, e apesar de não haver ligação direta, aquela versão de C é normalmente considerada como a implementação padrão. (No Apêndice B, você vai encontrar uma lista das funções mais comuns encontradas na biblioteca C padrão.)

Apesar de alguns compiladores C denominarem-se padrão UNIX, a biblioteca de funções de cada um deles poderá chamar as funções padronizadas por nomes completamente diferentes. Por exemplo, a função que copia uma série em outra é geralmente denominada **strcpy( )**, mas já foi denominada por **strcopy( )** e **stringcpy( )**. Leia o manual do usuário para saber quais são os nomes utilizados pelo seu compilador.

Muitas das funções que você vai precisar em C, já foram escritas. Funcionam como blocos que você vai utilizando. Caso você escreva uma função que vai utilizar com freqüência, ela poderá ser colocada na biblioteca. Alguns compiladores permitem que você a coloque na biblioteca padrão; outros fazem com que você crie outra biblioteca. De qualquer maneira, o código está lá para ser utilizado à vontade.

REVISÃO DE TERMOS

Antes que você aprenda como utilizar o compilador C, deverá rever alguns termos importantes. Você encontrará esses termos no manual do usuário de seu compilador C.

Código-fonte. O texto do programa que o usuário pode ler; comumente encarado *como o programa.*

Código-objeto. A tradução do código-fonte de um programa em código de máquina, que o computador consegue ler e executar.

Encadeador. Um programa que liga funções compiladas separadamente, e um único programa; o encadeador combina as funções da biblioteca C padrão com o código que você escreveu.

Biblioteca. O arquivo contendo as funções padronizadas que podem ser utilizadas por seu programa. Essas funções incluem todas as operações de E/S bem como outras rotinas úteis.

Tempo de compilação. Os eventos que ocorrem enquanto seu programa está sendo compilado. Uma ocorrência comum é um erro de sintaxe.

Tempo de processamento. Os eventos que ocorrem enquanto o seu programa está sendo executado.

COMPILANDO UM PROGRAMA C

A linguagem de programação C normalmente é compilada. Ao compilar um programa C, geralmente você utiliza quatro etapas.

- Cria seu programa.
- Compila seu programa.
- Encadeia seu programa com as funções necessárias, existentes na biblioteca.
- Executa o programa.

Ao contrário do interpretador BASIC, em que o editor de código-fonte é inerente, um compilador C (ou qualquer compilador) deve ter um editor separado para a criação de programas. Se você estiver utilizando CP/M poderá utilizar ED, o editor padrão do CP/M. Se estiver utilizando MS-DOS ou PC-DOS poderá utilizar EDLIN. Se estiver utilizando UNIX poderá usar VI. Existem ainda, no mercado, numerosos editores de tela de qualidade, que podem ser utilizados.

Os compiladores aceitam apenas arquivos de texto padronizados, como entrada. Por exemplo, seu compilador não vai aceitar arquivos criados por determinados processadores de palavras, porque apresentam códigos de controle e caracteres não-impressos. Caso não haja certeza se o editor é adequado, pergunte ao seu distribuidor ou a um amigo que tenha conhecimento.

UTILIZANDO COMANDOS EM LOTES PARA COMPILAR SEU PROGRAMA

Tanto o CP/M como o MS-DOS/PC-DOS permitem que sejam definidos comandos em lotes que automaticamente executarão um número de tarefas. Ao trabalhar com um compilador, você poderá utilizar um comando em lote para ajudá-lo na compilação. O nome de seu programa será representado por %1 e $1. Os comandos em lote relacionados na Tabela 2.4 vão compilar, enlaçar e processar o seu programa. Os comandos em lote são relacionados apenas como orientação; consulte o manual C para saber quais são as seqüências certas da compilação.

Se você está utilizando um sistema operacional diferente do MS-DOS/PC-DOS ou CP/M, terá de verificar se existe a possibilidade de lotes.

Tabela 2.4 Exemplos de comando em lote

CP/M	**MS-DOS/PC-DOS**
cc $1.c	cc %1.c
link $1,lib.c	link %1, lib.c
$1	%1

```
main()
{
int a,b,c;

printf("Digite dois numeros\n");
a = getnum();
b = getnum();
c = mul(a,b);
printf("a * b = %d\n",c);
}

mul(x,y)
int x,y;
{
return(x * y);
}

getnum()
{
char s[80];

gets(s);
return(atoi(s)); /* converte para inteiro */
}
```

```
10 PRINT "Digite dois numeros"
20 INPUT A,B
30 GOSUB 100
40 PRINT "a * b = "; C
50 END
100 C = A*B
130 RETURN
```

Figura 2.6 As versões em C e BASIC de um programa que utiliza os princípios gerais do Capítulo 2

UM PROGRAMA EXEMPLO

Será analisado um programa exemplo, curto, que ilustra os princípios gerais desenvolvidos neste capítulo. Agora você já deve ter uma idéia de como é um programa C, sendo que em alguns casos uma listagem "vale por mil palavras". Para comparação, a versão BASIC correspondente é apresentada em seguida. Esse programa, listado na Figura 2.6, insere dois números pelo teclado e imprime seu produto na tela.

A função **getnum( )** normalmente é encontrada na biblioteca padrão e lerá um número digitado no teclado. Caso a biblioteca padrão de seu compilador não a tenha, ela está incluída neste exemplo, e facilmente poderá fazer parte de seu programa. A função **mul( )** é exatamente aquilo que já foi descrito anteriormente, com o acréscimo das partes faltantes.

Esse programa exemplo permite que você tenha uma idéia de como se apresenta um programa C. Na versão C, a função **mul( )** é geral; vai multiplicar quaisquer variáveis inteiras, ao passo que a sub-rotina da linha 100 do BASIC vai multiplicar apenas **A** e **B**. Essa característica, a de escrever funções generalizadas, é um dos aspectos mais importantes da linguagem C.

EXERCÍCIOS

1. Caso você ainda não o tenha feito, processe o programa HELLO da Figura 2.1. Confirme a possibilidade de poder compilar, com sucesso, um programa C com seu compilador.

2. Escreva um programa curto que imprima na tela:

```
Esta e' a linha um.
Esta e' a linha dois.
```

3. Escreva declarações **printf( )** que escrevam o seguinte em sua tela:

```
Isto e' um teste. 1 2 3
123.23
Estes sao caracteres : a b c
```

4. Mostre as declarações que vão declarar as variáveis, conforme abaixo:

up, down	como inteiros
first, last	como de ponto flutuante de precisão simples
c	como um caractere

RESPOSTAS

2.
```
main()
{
printf("Esta e' a linha um.\n");
printf("Esta e' a linha dois.\n");
}
```

3.
```
printf("Isto e' um teste. %d %d %d\n",1,2,3);

printf("%f\n",123.23);

printf("Estes sao caracteres : %c %c %c\n",
        'a','b','c');
```

4. int up, down;
   float first, last;
   char c;

CAPÍTULO **3**

VARIÁVEIS, CONSTANTES, OPERADORES E EXPRESSÕES

Variáveis e constantes são manipuladas por operadores para formar expressões. Esta é a base de toda a programação. Este capítulo explica como esses conceitos relacionam-se com a linguagem de programação C.

VARIÁVEIS

Os nomes das variáveis em C podem ter de um até vários caracteres, com o primeiro caractere sendo obrigatoriamente uma letra e os demais podendo ser letras, números ou caractere de sublinhar. Uma variável não pode ser a mesma que uma palavra-chave de C, e não deve ter o mesmo nome que uma função já escrita ou que exista na biblioteca C.

TIPOS DE DADOS

Como já foi visto no Capítulo 2, existem diversos tipos de variáveis inerentes. O tamanho e o âmbito desses tipos de dados variam com o tipo de processador e com a implementação do compilador C. Para a maioria dos microcomputadores, o tamanho e o âmbito apresentados na Tabela 3.1 está correto.

O âmbito de **float** e **double** é geralmente dado em *dígitos de precisão*. As amplitudes de **float** e **double** dependerão do método utilizado para representar os números em ponto flutuante. Qualquer que seja o método utilizado, o número será bem grande.

Algumas implementações de C permitem que os modificadores **short, long** e **unsigned** sejam aplicados a outros tipos de dados além de **int**; por exemplo, **unsigned double**. Você deve verificar no manual de usuário de seu compilador C, se isto é possível. Lembre-se de que essas informações adicionais poderão não ser possíveis em todas as implementações C, e não são, por este motivo, portáteis.

Tabela 3.1 Tamanho e âmbito das variáveis para microcomputadores

Tipo	Largura em bits	Âmbito
char	8	0 a 255
int	16	−32768 a 32767
short int	8	−128 a 127
unsigned int	16	0 a 65535
long int	32	−4294967296 a 4294967295
float	32	aproximadamente 6 casas de precisão
double	64	aproximadamente 12 casas de precisão

Em C um caractere é equivalente a um byte. Um byte tem oito bits de comprimento. Na maioria das implementações C em microcomputadores, um inteiro curto também tem um byte de comprimento.

Não há em BASIC o conceito de número sem sinal; todos os números são positivos ou negativos. Em C, entretanto, é possível a declaração de inteiros sem sinal, utilizando o bit de sinal como parte do número, em vez de como um sinalizador. Assim, é dobrada a magnitude que o maior número inteiro pode ter. Na maioria dos computadores, o bit mais à esquerda, ou de *maior-ordem*, de um inteiro, é considerado como *indicador de sinal*. Quando este indicador é 0, o número é positivo; quando é 1 o número é negativo. Por exemplo, 127 em binário é

0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1

e −127 em binário é

1 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1

Assim, quando utilizamos números com sinal, como em BASIC, o maior inteiro pode ser 32767, que em binário fica:

0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

Se o bit de maior ordem fosse 1, o número seria interpretado como −32767. Entretanto, se ele foi declarado como não tendo sinal, quando o bit de maior ordem for 1, o número será 65535.

DECLARAÇÃO DE VARIÁVEIS

Todas as variáveis C devem ser declaradas antes de sua utilização. O nome de uma variável nada tem a ver com o seu tipo. A sintaxe para a declaração de cada tipo de variável é indicada nos exemplos que seguem:

```
int i;
short int i;
unsigned int ui;
long int li;
float f;
double d;
```

Existem três locais principais em um programa C, onde as variáveis serão declaradas: dentro das funções, na definição dos parâmetros das funções ou fora das funções. Essas variáveis são chamadas variáveis locais, parâmetros formais e variáveis globais.

VARIÁVEIS LOCAIS

As variáveis locais são declaradas dentro de uma função. Apenas poderão ser referenciadas pela declaração que está dentro da função em que as variáveis são declaradas. As variáveis locais não são conhecidas por outras funções externas à sua; por exemplo,

```
func1()
{
int x;
x = 10;
}

func2()
{
int x;

x = -199;
}
```

A variável inteira x foi declarada duas vezes, uma vez em **func1( )** e outra vez em **func2( )**. O x de **func1( )** não tem nenhuma relação com o x de **func2( )**.

Em C, as variáveis locais são *criadas* quando a função é chamada e *destruídas* quando ela "sai de cena". De modo similar, o armazenamento dessas variáveis locais é criado e destruído dinamicamente. Apesar de serem denominadas de *variáveis dinâmicas* ou *variáveis automáticas* em outras literaturas C, neste livro continuaremos a utilizar o termo *variável local* para este tipo de variável.

Pelo fato de as variáveis locais serem criadas e destruídas cada vez que a função é chamada, seu conteúdo se perde cada vez que a função "sai de cena". No trecho dado acima, nenhum dos valores x existirão fora dessas funções. Ao contrário do BASIC, em que todas as variáveis existem o tempo todo, a maioria das variáveis em quase todos os programas C está sendo continuamente criada e destruída.

PARÂMETROS FORMAIS

Como foi visto no Capítulo 2, quando uma função tem argumentos, estes devem ser declarados. São denominados *parâmetros formais* da função. Comportam-se como qualquer outra variável local dentro da função. A declaração é feita após o nome da função e antes da chave de abertura; por exemplo,

```
func1(first, last, ch)
int first, last;
char ch;
{
    int count;
    count=first*last;
    ch='a';
    .
    .
    .
}
```

Neste exemplo **func1( )** tem três argumentos denominados **first**, **last** e **ch**. Você precisa informar C de que tipos de variáveis são, declarando-as da maneira em que está indicado neste trecho de programa. Uma vez feito isso, poderão ser utilizados dentro da função como variáveis locais normais. Lembre-se de que, como variáveis locais, eles também são dinâmicos e são destruídos ao se sair da função.

Você deve certificar-se de que os parâmetros formais declarados são do mesmo tipo que os argumentos a serem utilizados para chamar a função. Caso haja uma discordância de tipos, poderão ocorrer resultados inesperados. Ao contrário do BASIC e da maioria das outras linguagens, C é possante e geralmente faz alguma coisa, mesmo que não seja o que você quer. Existem poucos erros de tempo de processamento e nenhuma verificação de limites. Como programador, você deve ter certeza de que estes erros não ocorram.

Assim como com as variáveis locais, você pode criar atribuições aos parâmetros formais de uma função, ou utilizá-los em qualquer expressão C permissível. Apesar dessas variáveis executarem a tarefa especial de receber o valor dos argumentos passados para a função, poderão ser utilizadas como qualquer outra variável local.

VARIÁVEIS GLOBAIS E extern

Você deve estar pensando como fazer para que a variável e seus dados continuem existindo durante a execução total do programa. Isso pode ser feito utilizando as variáveis globais. Ao contrário das variáveis locais, as globais mantêm o valor durante todo o tempo em que o programa estiver sendo processado. Criamos as variáveis globais, declarando-as fora de qualquer função. Poderão ser acessadas por qualquer expressão, independentemente da função em que esta expressão se encontra. Considere este exemplo:

```
int count; /*count is global */
main( )
{
    count=mul(10,123);
    .
    .
    .
}
func1( )
{
    int temp;
    temp=count;
    .
    .
    .
}
func2( )
{
     int count;
     count=10;
     .
     .
     .
}
```

Observe como a variável **count** foi declarada fora de todas as funções. Porém, poderia ter sido colocada em qualquer lugar, desde que não fosse em uma função. Lembre-se, porém, de que, uma vez que as variáveis devem ser declaradas antes de serem utilizadas, que é melhor declarar variáveis globais no começo do programa.

O exemplo mostra também que, mesmo que nem **main( )** nem **funcl( )** tenham declarado a variável **count**, ambos podem utilizá-la. A função **func2( )**, porém, declarou uma variável local denominada **count**. Quando **func2( )** faz referência à **count**, está referindo-se apenas à variável local e não à variável global. É muito importante lembrar que se uma variável local e uma variável global têm o mesmo nome, todas as referências a este nome de variável, dentro da função em que a variável local foi declarada, não causarão efeito sobre a variável global. Isso normalmente é um benefício bastante conveniente. Entretanto, se for esquecido, seu programa poderá agir de forma estranha, mesmo que "pareça" estar correto.

Uma vez que C permite que módulos compilados separadamente de um programa grande sejam encadeados para acelerar a compilação, deve haver certeza de que os dois arquivos podem se referir às variáveis globais. Uma variável global pode ser declarada apenas uma vez. Caso haja tentativa de declarar duas variáveis globais com o mesmo nome, o compilador C vai imprimir a mensagem de erro **duplicate variable name**, significando que o compilador não sabe qual é a variável que está sendo utilizada em um determinado momento. Ocorre o mesmo tipo de problema se todas as variáveis globais forem declaradas em cada arquivo. Na verdade está-se

tentando criar duas cópias de cada variável. Ao tentar encadear os dois módulos, aparece a mensagem de erro **duplicate label**, porque o encadeador não saberá que variável utilizar. A solução é declarar todas as variáveis globais em um arquivo e utilizar declarações de modificação externa (**extern**) no outro, conforme é indicado na Tabela 3.2.

No arquivo dois, a lista de variáveis globais foi copiada do arquivo um e o modificador **extern** foi acrescentado às declarações. O modificador **extern** informa ao compilador que os nomes e tipos que seguem já foram declarados em outro local. Em outros termos, **extern** permite ao compilador saber o que são os nomes e tipos para as variáveis globais, sem precisar criá-los de novo. Quando o encadeador tentar encadear os dois módulos, todas as referências às variáveis externas serão resolvidas.

Tabela 3.2 Utilizando variáveis globais em módulos compilados separadamente

Arquivo um	**Arquivo dois**
int x, y;	extern int x, y;
char ch;	extern char ch;
main( )	func22( )
{	{
.	x=y/10;
.	}
}	func23( )
func1( )	{
{	y=10;
x=123	}
}	

Ao utilizar uma variável global dentro de uma função que está no mesmo arquivo que a declaração da variável global, podemos escolher por utilizar **extern**, mesmo não sendo obrigados. Por exemplo, esta é a maneira de se utilizar esta opção:

```
int first,last; /* definicao de first e last
                   como globais */

main()
{
extern int first; /* uso opcional
                     da declaracao extern */

}
```

Apesar da declaração de variáveis **extern** poder ocorrer dentro do mesmo arquivo que a declaração global, elas não estão necessariamente ali. Caso o compilador C encontre uma variável que não foi declarada, ele vai verificar se a variável coincide com uma variável global. Se coincidir, o compilador assumirá que esta é a variável sendo referenciada.

Na maioria das implementações de C, o armazenamento das variáveis globais é feita em uma região fixa da memória, reservada pelo compilador C. As variáveis globais têm grande utilidade quando os mesmos dados são utilizados em várias funções do programa. Porém, por três motivos, devem ser evitadas as utilizações desnecessárias das variáveis globais: (1) ocupam espaço de memória durante todo o tempo em que o programa está sendo executado, e não apenas quando são necessárias; (2) a utilização de uma variável global onde uma variável local é satisfatória torna sua função menos generalizada porque fica dependente de algo que deve ser definido fora dela; e (3) a utilização de um grande número de variáveis globais poderá levar a erros de programa, porque ocorrem efeitos secundários desconhecidos e indesejáveis. No BASIC, onde todas as variáveis são globais, um dos problemas ao desenvolver-se um programa grande é a alteração acidental de um valor de variável, porque foi utilizado em outro local do programa. Isso poderá ocorrer em C, caso forem utilizadas muitas variáveis globais em seus programas.

Um dos pontos principais de uma linguagem estruturada é a compartimentalização de códigos e dados. Em C, esta compartimentalização é obtida pela utilização de variáveis locais e funções. Por exemplo, a Tabela 3.3 mostra duas maneiras de se escrever **mul( )**.

As duas funções vão entregar o produto de **x** e **y**. Porém, a versão generalizada, ou *parametrizada*, poderá ser utilizada para entregar o produto de *quaisquer* dois números, ao passo que a versão específica pode ser utilizada para encontrar exclusivamente o produto das variáveis globais **x** e **y**.

Tabela 3.3 Duas maneiras de escrever **mul( )**

Geral	**Específico**
	`int x,y;`
`mul(x,y)`	`mul( )`
`int x,y;`	`{`
`{`	`  extern x,y;`
`  return(x*y);`	`  return(x*y);`
`}`	`}`

AS VARIÁVEIS ESTÁTICAS

As variáveis **estáticas** são variáveis permanentes dentro de suas próprias funções ou arquivo. Diferem das variáveis globais porque, apesar de não serem conhecidas fora de suas funções ou arquivo, mantêm seus valores entre chamadas. Essa característica torna-as bastante úteis quando você escreve funções generalizadas e biblioteca de funções, utilizadas por outros programadores.

Um exemplo adequado de uma função que necessita de uma variável desse tipo é um gerador de uma série numérica, que fornece um número novo de acordo com o anterior. Você poderia declarar uma variável global para esse valor cada vez em que a função é utilizada, mas terá que lembrar de fazê-lo cada vez, o que é um contratempo. Com a utilização de uma variável global fica difícil colocar essa função em uma biblioteca de funções. A solução seria declarar de estatica a variável que mantém o número gerado, conforme indicado:

```
series()
{
static int series_num;

series_num = series_num + 23;
return(series_num);
}
```

Neste exemplo, a variável **séries_num** continua existindo entre chamadas de funções, em vez de ir e vir como o faria uma variável local normal. Significa que cada chamada de **séries( )** pode formar um novo membro da série, baseado no último número, sem declarar aquela variável globalmente.

Você pode ter observado alguma irregularidade sobre a função séries( ), conforme aparece no exemplo. A variável estática **séries_num** nunca é inicializada em um valor conhecido. Significa que a primeira vez em que a função é chamada, **séries_num** terá um valor randômico. Enquanto isto é aceitável para algumas aplicações, a maioria dos geradores de séries necessita de um ponto de partida bem definido. Uma solução seria inicializar **séries_num** antes da primeira chamada em **séries( )**, o que é feito facilmente quando **séries_num** é uma variável global. Porém, esta solução dificultaria a colocação da função **séries( )** em uma biblioteca de utilização geral. A melhor solução é utilizar as variáveis globais estáticas.

Vamos reescrever o exemplo de geração de série, de modo que o valor inicial possa ser inicializado por uma chamada à função **séries_start( )**. O exemplo agora fica:

```
static int series_num;

series()
{

series_num = series_num + 23;
return(series_num);
}

series_start(seed)
int seed;
{
series_num = seed;
}
```

Chamando **séries_start( )** com um valor inteiro conhecido, inicializa-se o gerador de séries. Em seguida, as chamadas para **séries( )** geram o próximo elemento da série.

Neste ponto você poderia colocar uma pergunta: Se as variáveis globais continuam existindo durante toda a execução do programa, por que é que **séries_num** foi definida como **static**? A função **séries( )** não é inútil para ser colocada em uma biblioteca de utilização geral? As respostas para essas perguntas dependem de um benefício secundário de **static**.

Lembre-se de que os nomes das variáveis estáticas locais são conhecidos apenas para as funções em que são declaradas, e que os nomes das variáveis **estáticas** globais são conhecidos apenas pelo arquivo em que residem. Significa que se você colocar as funções **séries( )** e **séries_start( )** em uma biblioteca, poderá utilizar as funções, mas não poderá fazer referência à variável **séries_num**. Está escondida de você. De fato, você poderá até declarar e utilizar outra variável denominada **séries_num** em seu programa (evidentemente que, em outro arquivo) sem criar problemas. Em resumo, o modificador **static** permite a existência de variáveis que são conhecidas para as funções que necessitam delas, sem confundir outras funções.

As variáveis **estáticas** permitem que você, o programador, esconda trechos de seu programa de outros trechos. Isso poderá ser bem vantajoso quando você estiver tentando manobrar um programa muito grande e complexo. A utilização do modificador **static** permite que você crie funções gerais que podem ser incluídas em bibliotecas, para utilização posterior.

AS VARIÁVEIS REGISTER

C tem ainda um último modificador variável/declaração que se aplica apenas aos tipos **int** e **char**, na maioria dos casos. O modificador **register** obriga o compilador C a manter o valor das variáveis declaradas com este modificador em *registrador* da UCP em vez de em memória, onde as variáveis são normalmente armazenadas. Significa que as operações com as variáveis **register** ocorrem a uma velocidade muito maior do que com aquelas variáveis armazenadas em memória, porque o valor das variáveis **register** estão na UCP e não necessitam de acesso à memória. Isso torna as variáveis **register** ideais para o controle de malha. O modificador **register** aplica-se apenas às variáveis locais e aos parâmetros formais em uma definição de função. Variáveis **register** globais não são permitidas. Aqui temos um exemplo de como declarar variáveis **register** de **int** e **char**:

```
func1(s,u)
register int s;
register char u;
{
    float temp;
    register int counter;
    .
    .
    .
}
```

O número exato de variáveis **register** em uma determinada função é estabelecido pelo tipo de processador e pela implementação de um determinado C que você está utilizando. Para a maioria dos sistemas de 8-bits, é permitida apenas uma variável **register**. Para os sistemas de 16-bits, normalmente podem ser utilizadas duas ou mais variáveis **register**. Você deverá se preocupar com isto apenas quando houver um problema de velocidade pois, uma vez que tenham sido declaradas variáveis **register** em excesso, o compilador C automaticamente as transformará em variáveis que não são **register**. Neste livro, a maioria das variáveis de controle de malha serão do tipo **register**.

VETORES

Apesar do Capítulo 8 explicar vetores em detalhe, alguns conceitos importantes terão de ser observados agora. Os **vetores** podem ser de variáveis de qualquer tipo. O formato geral da declaração de vetor é

tipo nome_variável[número de elementos];

Por exemplo, para construir um vetor chamado **q** com dez elementos inteiros, deveria escrever

```
int q[10];
```

A referência aos elementos de um vetor é feita pela especificação do número do elemento entre chaves, após o nome do vetor. Para fazer a referência dos três primeiros elementos do vetor **q**, escreveríamos

```
q[0]
q[1]
q[2]
```

Ao contrário de algumas versões BASIC, todos os vetores em C iniciam pelo elemento zero. Significa que os dez elementos de um vetor **q** são indexados de zero até nove.

C não executa nenhuma verificação de contorno, sendo de sua inteira responsabilidade certificar-se de que seus índices de vetor estão dentro da faixa declarada. Se você exceder essa faixa, vai destruir o conteúdo de outras variáveis ou o código para o seu programa, com resultados eventualmente catastróficos em qualquer um dos dois casos.

Séries de caracteres são simplesmente vetores de caracteres em C. O programa da Figura 3.1 lê uma série de caracteres, faz a impressão e imprime em seguida apenas os três primeiros caracteres. (É apresentado também o programa equivalente em BASIC.) A versão C utiliza **gets( )**, uma função normalmente encontrada nas bibliotecas C, que vai ler uma série de caracteres do teclado e colocá-los em um vetor de caracteres.

Lembre-se de que você pode chamar **printf( )** e **gets( )** simplesmente pela utilização do nome da variável série, sem indexação, quando a referência é feita para todo o vetor.

```
main()
{
char string[80]; /* define um vetor
                    de oitenta caracteres */
gets(string); /* utiliza a rotina da biblioteca C
                 para ler uma serie de caracteres
                 do teclado */
printf(string); /* imprime a serie de caracteres */

/* agora imprime os tres primeiros caracteres */

printf("\n%c%c%c\n",string[0],string[1],string[2]);
}
```

```
10 INPUT A$
20 PRINT A$
30 PRINT MID$(A$,1,1);MID$(A$,2,1);MID$(A$,3,1);
40 END
```

Figura 3.1 As versões C e BASIC de um programa que lê e imprime uma série de caracteres e em seguida imprime apenas os três primeiros caracteres

DECLARAÇÕES DE ATRIBUIÇÃO

Até o momento, os exemplos atribuíam valores para as variáveis sem discussão. Esta seção vai explicar o procedimento geral de atribuição e a sintaxe, assim como algumas variações.

A forma generalizada da *declaração de designação* é

nome_variável = expressão;

onde "expressão" pode ter a simplicidade de uma constante ou a complexidade de uma expressão. Assim como o BASIC e o FORTRAN, C utiliza um sinal de igual simples para indicar a atribuição. (Pascal utiliza o := construção.) O *alvo*, ou o lado esquerdo, da atribuição tem que ser uma variável e não uma função ou uma constante.

CONVERSÃO DE TIPO EM ATRIBUIÇÕES

A conversão de tipo refere-se àquela situação em que as variáveis de um tipo são misturadas com variáveis de outro tipo. Quando isso ocorre em uma declaração de atribuição, a regra de conversão de tipo é bastante simples: o valor do lado direito da atribuição é convertido ao tipo do lado esquerdo, a variável denominada "alvo". Por exemplo,

```
int   x;
char  ch;
float f;

func()
{
ch = x;
x  = f;
f  = ch;
f  = x;
}
```

Em **ch=x**, os bits à esquerda, de ordem superior, da variável inteira **x** são cortados, ficando **ch** com os 8 bits da ordem inferior. Se, para começar, **x** estivesse entre 256 e 0, **ch** e **x** teriam o mesmo valor. Caso contrário, o valor de **ch** refletiria apenas os bits de ordem inferior de **x**. Em **x=f**, **x** recebe a parte não-fracionária de **f**. Em **f=ch**, **f** vai converter o valor inteiro de 8 bits, armazenado em **ch**, para o mesmo valor, exceto no formato de ponto flutuante. Isso ocorre também em **f=x**, exceto que **f** vai converter um valor inteiro para o formato de ponto flutuante.

Ao converter de inteiros para caracteres, inteiros longos para inteiros, e inteiros para inteiros curtos, a regra básica a ser lembrada é a de que será removido o número apropriado de bits de ordem superior. Significa que 8 bits serão perdidos ao passarmos de um inteiro para um caractere ou inteiro curto, e 16 bits serão perdidos ao passarmos de um inteiro longo para um inteiro.

A Tabela 3.4 representa essas conversões de tipo de atribuição. Você deve lembrar-se de dois pontos importantes:

1. A conversão de um **int** para um **float**, ou um tipo **float** para **double**, não vai acrescentar precisão ou certeza. Esses tipos de conversão apenas alteram a forma em que o valor é representado.
2. Alguns compiladores C (e processadores) sempre tratam uma variável **char** como positiva, independentemente do valor que tiver ao convertê-la em um **int** ou **float**. Outros compiladores vão tratar os valores das variáveis **char** maiores que 127 como números negativos na conversão. De uma maneira geral, você deve utilizar variáveis **char** para caracteres e utilizar variáveis **int** quando necessário, para evitar um problema eventual nessa área.

Para utilizar a Tabela 3.4 em conversões não demonstradas diretamente, basta fazer as conversões sucessivas. Por exemplo, para converter de **double** para **int**, converta primeiro de **double** para **float** e, em seguida, de **float** para **int**.

Caso você tenha utilizado uma linguagem de computador do tipo Pascal, que proíbe essa conversão de tipo automática, você poderá pensar que C é uma linguagem muito rudimentar. Porém, lembre-se de que C foi projetado para tornar mais fácil a vida do programador, ao permitir que o trabalho fosse feito em C em lugar do assembler. Para substituir o assembler, C deve permitir este tipo de conversão-de-tipo.

Tabela 3.4 Regras de conversão de tipo em atribuição

Tipo do Alvo	Tipo da Expressão	Possíveis Perdas de Informação
char	short int	sinal
char	int	8 bits de ordem superior
char	long int	24 bits de ordem superior
short int	int	8 bits de ordem superior
short int	long int	24 bits de ordem superior
int	long int	16 bits de ordem superior
int	float	parte fracionária e possivelmente mais
float	double	precisão; resultado arredondado

INICIALIZAÇÃO DE UMA VARIÁVEL OU VETOR

Para a maioria das variáveis C você pode atribuir um valor ao mesmo tempo em que são declaradas, pela colocação de um sinal de igual e uma constante após o nome da variável. A forma geral de inicialização é

```
tipo nome_variável = constante;
```

Alguns exemplos são

```
char ch = 'a';
int first = 0;
float balance = 123.23;
```

Variáveis globais e **estáticas** são inicializadas apenas no início do programa. Variáveis locais e **register** são inicializadas toda vez que a função é chamada. Todas as variáveis globais e **estáticas** supomos que são inicializadas em zero, caso não seja especificado outro valor de inicialização, mas não conte com isso porque é um detalhe normalmente esquecido quando os compiladores C são implementados. Valores locais e **register**, que não são inicializados terão valor desconhecido antes que a primeira atribuição seja feita para elas.

Podemos inicializar também vetores globais. Para vetor de caracteres utilizados para conter séries, você pode colocar a série entre aspas. Uma *série* é um vetor de caracteres terminado por um elemento nulo. Em C um *nulo* normalmente é um 0. Porém, uma vez que isto poderá variar de uma implementação para outra, deverá ser utilizado o símbolo especial **\0**, para nulo. As séries de seu texto de programa terão esse símbolo colocado automaticamente no final, pelo compilador C, de modo que você não precisa fazê-lo. Por exemplo, para inicializar o vetor **str** com a série **"Hi there"**, poderia escrever

```
char str[9] = "Hi there";
```

A razão pela qual **str** deve ter nove elementos de comprimento é que todas as séries em C terminam com nulo, o que é automaticamente colocado pelo compilador C.

Quando você imprimiu a série **HELLO** em sua tela no Capítulo 2, utilizou uma série constante. O comprimento da série **HELLO** não é cinco, mas seis, devido ao terminador nulo, que você não vê. Isso é importante quando são declarados vetores de caracteres que contêm séries. A declaração deverá ter um caractere a mais que o desejado para o vetor, de modo que haja espaço para o terminador nulo.

Os vetores podem ser inicializados também pela listagem de seus elementos, separados por vírgulas, entre chaves. Por exemplo, para inicializar **"Hi there"** com esse método, poderia escrever

```
char str[9] = {'H','i',' ','t','h','e','r','e','\0'};
```

O símbolo '\0' representa o nulo no final da série. Aqui ele deve ser utilizado explicitamente, porque o compilador C não sabe que se trata de uma série de caracteres.

Uma última observação: vetores locais não podem ser inicializados; porém, vetores locais estáticas podem.

CONSTANTES

As constantes, em C, referem-se a valores fixos que não podem ser alterados pelo programa. Poderão ser de qualquer tipo de dados, conforme é indicado na Tabela 3.5.

Tabela 3.5 Exemplos de constantes

Tipos de Dados	Exemplos de Constantes
char	'a' '\n' '9'
int	1 123 21000 −234
long int	35000 −34
short int	10 −12 90
unsigned int	10000 987
float	123.23 4.34e−3
double	123.23 12312333 −0.9876324

C suporta um outro tipo de constante além daqueles sete tipos de dados pré-definidos. É a série constante. Todas a séries constantes são envolvidas por aspas, como em **"this is a test"**. Você não deve confundir série com caracteres. Uma simples constante de caractere é envolvida por apóstrofo, como em 'a'. Lembre-se de que todas as séries terminam em nulo.

CARACTERES ESPECIAIS CONSTANTES

Envolver todas as constantes de caractere com apóstrofo funciona bem para a maioria dos caracteres impressos, mas alguns – como o retorno de carro – não podem ser colocados a partir do teclado. Por este motivo é que C criou os caracteres especiais constantes.

O símbolo '\n' significa linha nova e o '\0' representa o terminador nulo. C contém diversos códigos especiais desse tipo, conforme listado na Tabela 3.6, que permite a sua colocação como constantes. Esses códigos devem ser utilizados em vez de seus equivalentes ASCII, para ajudar a garantir a adaptabilidade.

OPERADORES

C é muito rico em operadores intrínsecos. *Operador* é um símbolo que informa ao compilador para que execute determinadas operações matemáticas ou lógicas. C tem três classes de operadores: *aritmético, de relação e lógico* e *por-bit*. Além disso, tem alguns operadores especiais.

Tabela 3.6 Códigos especiais

Código	Significado
\ b	retrocesso
\ f	mudança de página
\ n	linha nova
\ r	retorno do carro
\ t	tabulação horizontal
\ '	caractere com apóstrofo
\ 0	nulo

OPERADORES ARITMÉTICOS

A Tabela 3.7 lista os operadores aritméticos permitidos em C. Os operadores +, –, * e / funcionam da mesma maneira em C que em BASIC ou qualquer outra linguagem de computador. Eles podem ser aplicados em qualquer tipo de dado intrínseco permitido por C. Quando / for aplicado em um inteiro ou em caractere, qualquer resto será truncado; por exemplo, 10/3 será igual a 3 em uma divisão inteira.

O menos unário, em verdade, multiplica o seu operando por –1. Assim, todo número precedido por um sinal de menos muda de sinal.

Tabela 3.7 Operadores aritméticos

Operador	Ação
–	subtração; também menos unário
+	soma
*	multiplicação
/	divisão
%	resto
– –	decremento
+ +	incremento

Em C, o operador de resto, %, opera da mesma maneira que em outras linguagens. Lembre-se de que a operação de resto fornece o resto de uma divisão inteira. Entretanto, como tal, % não pode ser utilizado nos tipos **float** ou **double**. Eis um exemplo da utilização de %:

```
int x,y;

x = 10;
y = 3;
printf("%d\n",x / y); /* ira imprimir 3 */
printf("%d\n",x % y); /* ira imprimir 1,
                         o resto da divisao inteira */

x = 1;
y = 2;
printf("%d %d\n",x / y,x % y); /* imprime 0 e 1 */
```

A última linha imprime 0 e 1 porque 1/2 em uma divisão inteira é 0 com resto 1; portanto, 1%2 fornece resto 1.

C permite a utilização de dois operadores bastante úteis, pouco encontrados em outras linguagens de computador. São os operadores de *decremento* e *incremento:* ++ e – –. A operação ++ acrescenta 1 ao operando, e – – subtrai 1. Portanto, o que segue são operações equivalentes:

x = x + 1 é o mesmo que + + x;
x = x – 1 é o mesmo que – – x;

Tanto o operador de decremento como o de incremento podem preceder ou seguir o operando. Por exemplo,

```
x = x + 1 ;
```

pode ser escrito como

```
++x;
```

ou

```
x++;
```

Entretanto, existe uma diferença quando são utilizados em uma expressão. Quando um operador de incremento ou decremento precede seu operando, C vai executar a operação antes de utilizar o valor do operando. Caso o operador siga seu operando, C vai utilizar o valor do operando antes de incrementar ou decrementar. Considere o seguinte:

```
x=10;
y=++x;
```

Neste caso y será 11. Entretanto, caso tenha sido escrito

```
x = 10;
y = x++;
```

y assumirá o valor 10. Nos dois casos x é colocado em 11; a diferença está em quando ocorre a mudança de valor. Existem vantagens significativas em se poder controlar quando a operação de incremento ou decremento deve ocorrer.

A maioria dos compiladores C escreve códigos-objeto eficientes e rápidos, para operações de incremento e decremento, que são melhores que o código gerado pela utilização de uma declaração de atribuição. Assim, é uma boa idéia utilizar os operadores de incremento e decremento sempre que possível.

A Tabela 3.8 mostra a precedência dos operadores aritméticos. Operadores com o mesmo nível de precedência são avaliados pelo compilador, da esquerda para a direita. Evidentemente, os parênteses poderão ser utilizados para alterar a ordem da avaliação. Os parênteses são tratados pelo C da mesma forma que pela maioria das linguagens de computadores: forçam uma operação, ou um conjunto de operações a terem um nível de precedência mais elevado.

Tabela 3.8 Precedência das operações aritméticas

maior	++ −−
	− (menos unário)
	* / %
menor	+ −

OPERADORES RELACIONAIS E LÓGICOS

Nos termos *operador relacional* e *operador lógico, relacional* refere-se ao relacionamento que os valores têm entre si, e *lógico* refere-se a maneira como esses relacionamentos podem ser interligados. Pelo fato dos operadores relacionais e lógicos com freqüência operarem juntos, serão discutidos ao mesmo tempo.

O segredo do conceito de operador lógico e relacional é a idéia de verdadeiro ou falso. Em C, verdade é qualquer valor diferente de 0; falso é 0. Portanto, as expressões que utilizam operadores relacionais ou lógicos fornecem 1 para verdadeiro e 0 para falso.

A Tabela 3.9 mostra os operadores relacionais e lógicos. Caso você conheça BASIC, deve notar que existem diferenças entre C e BASIC. Por exemplo, **diferente** em C é !=, enquanto em BASIC é <>; a verificação de igualdade é o duplo sinal de igual (==), e não o sinal de igual simples do BASIC. Observe também que os operadores lógicos são diferentes: C utiliza os operadores especiais &&, || e !.

Tanto os operadores relacionais como lógicos têm precedência mais baixa que os operadores aritméticos. Significa que uma expressão do tipo 10 > 1 + 12 será avaliada como se tivesse sido escrita como 10 > (1 + 12). O resultado, evidentemente, é falso. Outro exemplo é a expressão

$$10 > 5 \;\&\&\; !\;(10 < 9) \;\|\| 3 <= 4$$

que será avaliada como verdadeiro. A Tabela 3.10 mostra a precedência relativa dos operadores lógicos e relacionais.

Tabela 3.9 Operadores relacionais e lógicos

Operador Relacional	**Ação**
>	maior que
>=	maior que ou igual
<	menor que
<=	menor que ou igual
==	igual
!=	diferente
Operador Lógico	**Ação**
&&	(AND) e lógico
||	(OR) ou lógico
!	(NOT) negação

Tabela 3.10 Precedência dos operadores relacionais e lógicos

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

Lembre-se de que, todas as expressões relacionais ou lógicas fornecem o resultado 0 ou 1. Portanto, esse fragmento de programa, além de estar correto, vai imprimir o número 1 na tela:

```
int x;

x = 100;
printf("%d\n",x > 10);
```

OPERADORES BIT-A-BIT

Ao contrário de muitas outras linguagens, C tem um conjunto completo de operadores bit-a-bit. Uma vez que C foi projetada para ser utilizada em lugar da linguagem assembly para a maioria das tarefas de programação, ela deve executar todas as operações possíveis em assembly. *Operações bit-a-bit* se referem ao teste, posicionamento e deslocamento dos bits de uma variável inteira ou de caractere. Estas operações não podem ser executadas nos tipos **float** ou **double**. A Tabela 3.11 relaciona esses operadores.

As operações bit-a-bit com freqüência encontram aplicações em acionadores de dispositivos – programas de modems, rotinas de arquivos em disco e rotinas de impressora – porque estas operações podem ser utilizadas para mascarar certos bits, como de paridade. (O bit de paridade é utilizado para confirmar que os demais bits do byte estão inalterados. É sempre o bit de maior ordem em cada byte.)

Tabela 3.11 Os operadores bit-a-bit

Operador	**Ação**
&	AND
|	OR
^	OR exclusivo
~	complemento de um
> >	deslocamento à direita
< <	deslocamento à esquerda

Você pode pensar no **AND** bit-a-bit como uma maneira de "desligar" os bits. Pela utilização do **AND**, qualquer bit que seja 0 em um dos operandos fará com que o bit correspondente da variável seja colocado em 0. Por exemplo, a função a seguir lerá um caractere da porta do modem, utilizando a função **read_modem( )**,e colocará o bit de paridade em zero:

```
char ch;

get_char_from_modem()
{
c = read_modem(); /* le um caractere
                     da porta do modem */
return(ch & 127);
}
```

A paridade é indicada pelo oitavo bit, que é colocado em 0 ao aplicarmos AND com um byte em que os bits de 1 até 7 estão em 1, e o bit 8 está em 0. A expressão **ch & 127** significa aplicar **AND** aos bits de **ch** com os bits que formam o número 127. O resultado é que o oitavo bit de **ch** ficará 0. No exemplo que segue, vamos assumir que **ch** recebeu o caractere 'A' com bit de paridade em 1.

```
    bit de paridade
    1 1 0 0 0 0 0 1  ch contendo 'A' com paridade 1
&   0 1 1 1 1 1 1 1  127 em binário
    ---------------  faça um AND bit-a-bit
    0 1 0 0 0 0 0 1  'A' sem paridade
```

O **OR** bit-a-bit pode ser utilizado para "ligar" bits. Qualquer bit de um dos operandos que estiver em 1 fará com que o bit correspondente da variável seja colocado em 1. Por exemplo, temos **128 | 3**:

```
    1 0 0 0 0 0 0 0  128 em binário
    0 0 0 0 0 0 1 1    3 em binário
|   ---------------  OR bit-a-bit
    1 0 0 0 0 0 1 1  resultado
```

Um OR exclusivo, normalmente abreviado XOR, "liga" um bit apenas se os bits comparados forem diferentes. Por exemplo, **127 ^ 120** é

```
    0 1 1 1 1 1 1 1  127 em binário
    0 1 1 1 1 0 0 0  120 em binário
^   ---------------  XOR bit-a-bit
    0 0 0 0 0 1 1 1  resultado
```

De uma maneira geral AND, OR e XOR bit-a-bit aplicam suas operações diretamente em cada bit da variável, individualmente. Por estes e outros motivos, as operações bit-a-bit não são normalmente utilizadas em declarações condicionais da maneira como o são os operadores relacionais e lógicos. Por exemplo, se x = 7, então x & & 8 avalia a verdade, ou 1, enquanto x & 8 avalia o falso, ou 0.

Os operadores relacionais ou lógicos sempre fornecem um resultado que é 0 ou 1; as operações similares bit-a-bit alteram a variável de acordo com a operação específica. Em outras palavras, as operações bit-a-bit são utilizadas para alterar o valor das variáveis, e não para avaliar se uma condição é verdadeira ou falsa.

Os *operadores de deslocamento,* >> e <<, movem todos os bits de uma variável para a direita ou para a esquerda, conforme é especificado. A forma geral da declaração de deslocamento para a direita é

variável >> número de posições de bits

e a declaração de deslocamento para a esquerda

variável << número de posições de bits

À medida que os bits são deslocados para um lado ou outro, são acrescentados zeros na outra extremidade. (Em alguns computadores, são acrescentados 1's, de modo que você deve consultar seu manual do compilador C para verificação de detalhes específicos.) Lembre-se, um deslocamento *não* é uma rotação; isto é, os bits que desaparecem em uma extremidade *não* reaparecem na outra. Os bits que desaparecem estão perdidos, e são substituídos por zeros que ocupam seu lugar.

Tabela 3.12 Efeitos de multiplicação e divisão com operadores de deslocamento

char x;	x à medida que cada deslocamento é executado	Valor de x
x=7;	0 0 0 0 0 1 1 1	7
x << 1;	0 0 0 0 1 1 1 0	14
x << 3;	0 1 1 1 0 0 0 0	112
x << 2;	1 1 0 0 0 0 0 0	192
x >> 1;	0 1 1 0 0 0 0 0	96
x >> 2;	0 0 0 1 1 0 0 0	24

As operações de deslocamento bit-a-bit poderão ser bastante úteis ao decodificar as entradas de dispositivos externos, como de conversores A/D, e ler a informação de status. Os operadores de deslocamento bit-a-bit poderão ser utilizados também para a execução rápida

de multiplicação e divisão de inteiros. Um deslocamento para a esquerda efetivamente multiplica um número por dois, e um deslocamento para a direita vai dividi-lo por dois, conforme indica a Tabela 3.12. Presume-se que são introduzidos zeros, enquanto os bits deslocados nas extremidades estão perdidos.

O operador complemento de um, ~, inverte o estado de cada bit na variável especificada; isto é, todos 1's são colocados em 0, e todos 0's são colocados em 1.

Os operadores bit-a-bit são freqüentemente utilizados em rotinas de codificação. Caso você queira fazer com que um arquivo em disco pareça ilegível, basta executar algumas manipulações bit-a-bit nele. Um dos métodos mais simples seria o de complementar cada byte pela utilização do complemento de um, para inverter cada bit do byte, conforme indicado:

Byte original	0 0 1 0 1 1 0 0
Após o 1º complemento	1 1 0 1 0 0 1 1
Após o 2º complemento	0 0 1 0 1 1 0 0

Observe que a seqüência de dois complementos em uma linha sempre fornece o número original. Assim, o primeiro complemento representa a versão codificada do byte; o segundo complemento decodifica-o ao valor original.

Para codificar um arquivo, utilizando o complemento de um, você poderia utilizar a função **encode( )**:

```
encode(ch) /* um funcao de codificacao simples */
char ch;
{
ch = ~ch; /* complemento de 1 */
return(ch);
}
```

O OPERADOR ?

C tem um operador muito poderoso e conveniente, que pode ser utilizado para substituir declarações da forma if-then-else. O par de operador ternário ? : toma a forma genérica:

Exp1 ? Exp2 : Exp3

onde Exp1, Exp2 e Exp3 são expressões.

O operador ? opera da seguinte forma: Exp1 é avaliada. Caso seja verdadeira, Exp2 é avaliada e torna-se o valor da expressão. Caso Exp1 seja falsa, Exp3 é avaliada e seu valor torna-se o valor da expressão. Por exemplo, considere

```
x = 10;
y = x > 9 ? 100 : 200;
```

Neste exemplo, y recebe o valor 100. Se x fosse menor que 9, y receberia o valor 200. O mesmo código escrito em BASIC ficaria:

```
10 X=10
20 IF X>9  THEN GOSUB 100 ELSE GOSUB 200
30 END
100 X=100
110 RETURN
200 X=200
210 RETURN
```

O operador ? será discutido com maiores detalhes no Capítulo 4, em relação com a declaração if-then em C.

OS OPERADORES DE PONTEIROS: & e *

Em C, um **ponteiro** é o endereço da memória de uma variável. O conhecimento do endereço de uma variável pode ser de grande ajuda em certos tipos de rotinas. Porém, os ponteiros têm duas funções principais em C: primeiro, podem fornecer um meio rápido de referência aos elementos de um vetor e, em segundo lugar, permitem que as funções C modifiquem seus parâmetros de chamada. Esses tópicos e suas utilizações serão explicados em maiores detalhes no Capítulo 7, dedicado exclusivamente aos ponteiros. Por ora, entretanto, você vai aprender a respeito dos dois operadores especiais que permitem a existência dos ponteiros.

O primeiro operador é &. É um operador unário que devolve o endereço de memória do seu operando. (Lembre-se de que um operador unário necessita apenas de um operando.) Por exemplo,

```
m = & count;
```

coloca em **m** o endereço de memória da variável **count**. Esse endereço é o posicionamento interno do computador, na variável. Nada tem a ver com o *valor* de **count**.

Vamos presumir que a variável **count** utiliza a posição de memória 2000 para armazenar o seu valor, e que tenha um valor de 100. Assim, após a atribuição **m=&count**, **m** terá o valor 2000.

O segundo operador é *. É um operador unário que devolve o valor da variável localizado no endereço que segue. Por exemplo, se **m** contém o endereço de memória da variável **count**, então

```
q = * m;
```

coloca o valor de count em q. Continuando com este exemplo, q terá o valor 100, porque 100 é armazenado na posição 2000, que era o endereço de memória armazenado em m.

Infelizmente, o AND bit-a-bit e o sinal do primeiro operador são o mesmo, assim como o sinal de multiplicação e o sinal do segundo operador. Esses operadores não apresentam nenhum relacionamento entre si. Tanto & como * têm precedência maior que todos os outros operadores aritméticos, com exceção do menos unário, igual a eles.

As variáveis que contêm endereços de memória, ou ponteiros, como são denominados em C, terão de ser declaradas pela colocação de um * em frente ao nome da variável, indicando ao compilador que existirá um ponteiro para esse tipo de variável. Por exemplo, para declarar uma variável tipo-ponteiro para **char ch**, você deve escrever

```
char * ch;
```

Você pode colocar informações de ponteiro e não-ponteiro na mesma declaração. Por exemplo,

```
int x, * y, count;
```

declara **x** e **count** como sendo inteiros e **y** como um ponteiro para um inteiro.

No programa da Figura 3.2, os operadores & e * são utilizados para colocar o valor 10 em uma variável denominada alvo.

```
/*
** atribuicao com * e &
*/
main()
{
int  target,source;
int *m;

source = 10;
m = &source;
target = *m;
}
```

Figura 3.2 Um programa que utiliza os operadores & e *

RESUMO DE PRECEDÊNCIA

A Tabela 3.13 relaciona a precedência de todos os operadores C, incluindo alguns que serão discutidos mais adiante. Observe que todos os operadores, com exceção dos operadores unários e &, unem-se da esquerda para a direita. Os operadores unários (*, &, –), e o operador &: unem-se da direita para a esquerda.

Tabela 3.13 Precedência dos operadores C

mais alto	( ) [ ] − → .
	! ~ ++ −− − (tipo) * & sizeof
	* / %
	+ −
	<< >>
	< <= > >=
	== !=
	&
	^
	|
	&&
	||
	?:
	= += −= *= /=
mais baixo	,

EXPRESSÕES

Os operadores, as constantes e as variáveis formam as *expressões*. Uma expressão em C é qualquer combinação válida desses itens. Você já deve conhecer a forma geral de uma expressão de sua experiência anterior em programação. Alguns aspectos das expressões que se relacionam especificamente com C serão discutidas agora.

CONVERSÃO DE TIPO NAS EXPRESSÕES

Quando constantes e variáveis de diferentes tipos são misturadas em uma expressão, são convertidas ao mesmo tipo. O compilador C vai converter todos os operandos ao tipo do maior operando. Isto é feito em um processo de operação-por-operação, seguindo as seguintes regras de conversão de tipo:

1. Todos **chars** e **short ints** são convertidos em **ints**. Todos **floats** são convertidos em **doubles**.
2. Para todos os pares de operandos: caso um dos operandos seja **double**, o outro operando será convertido para **double**. Por outro lado, caso um dos operandos seja **long**, o outro operando será convertido para **long**; ou, caso um dos operandos seja **unsigned** o outro será convertido para **unsigned**.

Depois que essas regras de conversão forem aplicadas, cada par de operandos será do mesmo tipo, e o resultado de cada operação será o mesmo que o tipo dos dois operandos. Observe que a regra 2 tem diversas condições que devem ser aplicadas em seqüência.

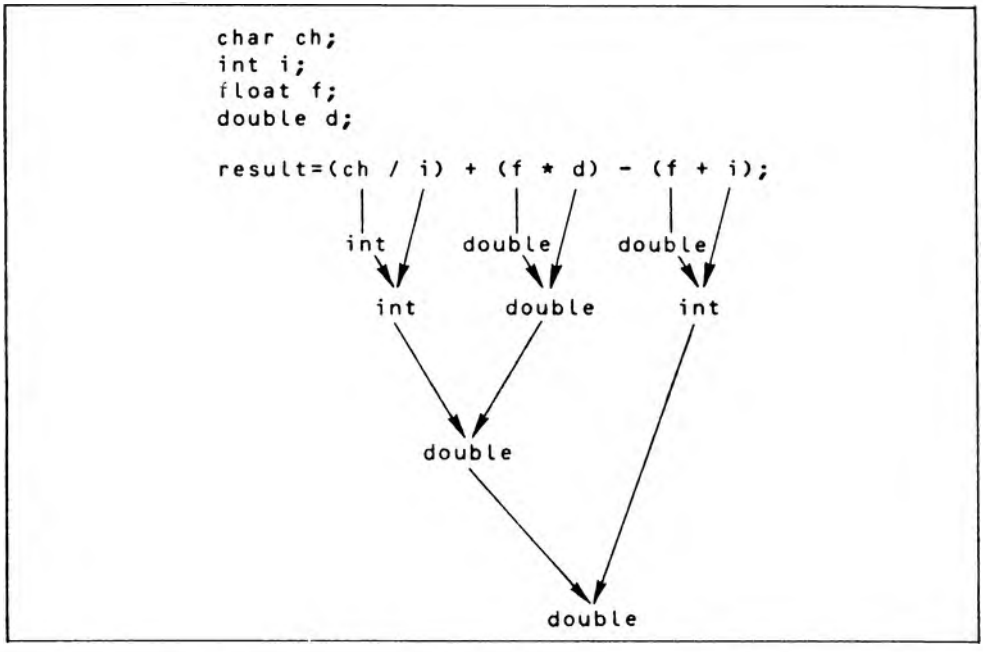

Figura 3.3 Um exemplo de conversão de tipo

Por exemplo, considere a conversão de tipo que ocorre na Figura 3.3. Primeiro, o caractere **ch** é convertido em um inteiro e **float f** é convertido em um **double**. Depois, o resultado de **ch/i** é convertido para **double** porque **f*d** é **double**. O resultado final é **double** porque, neste momento, os dois operandos são **double**.

MOLDAGENS

É possível forçar uma expressão a ser de um determinado tipo, utilizando uma construção denominada moldagem (*cast*). A forma geral do cast é

(tipo) expressão

onde **tipo** representa um dos tipos de dados padrão de C. Por exemplo, se você quer ter a certeza de que a expressão **x/2** seja avaliada para o tipo **float**, poderia escrevê-la como abaixo:

(float) x/2

Casts normalmente são considerados operadores. Como operador o cast é unário e tem a mesma precedência que qualquer outro operador unário.

Apesar de não serem utilizados com muita freqüência em programação, existem ocasiões em que podem ser bastante úteis. Por exemplo, a Figura 3.4 mostra um programa que poderia ser escrito para utilizar um inteiro controle de malha, mas para que seja executada uma computação nele exige-se uma parte fracionária. Sem o cast (**float**), apenas uma divisão inteira teria sido executada; mas cast garante que a parte fracionária da resposta seja apresentada na tela.

```
/*
** imprime i e i/2 com fracoes
*/
main()
{
int i;

for(i = 0;i <= 100;++i)
  printf("%d / 2 e': %f\n",i,(float)i / 2);
}
```

Figura 3.4 Um programa utilizando cast

ESPAÇAMENTO E PARÊNTESES

Para facilitar a leitura podem ser acrescentados espaços e tabulações em C. Por exemplo, as duas expressões seguintes são as mesmas.

```
x=10/y~(127/x);
x = 10 / y ~(127/x);
```

A utilização de um parênteses, redundante ou adicional, não provoca erros nem retarda a execução da expressão. Você é encorajado a utilizar os parênteses para tornar bem clara a ordem exata da avaliação, tanto para você mesmo como para outros que terão a necessidade de analisar seu programa posteriormente. Por exemplo, qual das duas expressões seguintes é mais fácil de ser lida?

```
x=y/3-34*temp&127;
x=(y/3) - ((34*temp) & 127);
```

ABREVIAÇÃO EM C

C tem uma "taquigrafia" especial que simplifica a codificação de um determinado tipo de declaração de designação. Por exemplo,

```
x=x+10;
```

que pode ser escrito de forma reduzida conforme

```
x+=10;
```

O par operacional + = informa ao compilador que deve ser determinado para x o valor de x mais **10**.

Essa abreviação funciona com todos os operadores de C. A forma geral é

variável = variável operador expressão

É o mesmo que

variável operador = expressão

Como outro exemplo,

```
x=x-100;
```

é o mesmo que

```
x-=100;
```

Você deve se familiarizar com estas abreviações, pois são muito utilizadas em programas C profissionais.

EXERCÍCIOS

1. Relacione os sete tipos de dados inerentes ao C.
2. Relacione os tipos de dados que possam vir precedidos pelo modificador **register**.
3. Escreva uma função denominada **add( )**, que tenha dois argumentos inteiros. A função vai entregar o valor da soma desses dois argumentos.
4. Imagine um par de funções que opere em uma variável global. Esta variável global é denominada **counter**. A função **func1( )** vai incrementar **counter**, e **func2( )** vai decrementá-la. Mostre uma maneira de escrever essas duas funções, pressupondo que os dois estão no mesmo arquivo.
5. Reescreva as funções do Exercício 4 como se estivessem em arquivos diferentes.
6. Adicione às funções do Exercício 4 uma terceira função denominada **init( )** que inicialize **counter** com o valor 0. Coloque todas essas funções em um mesmo arquivo e escreva-as de modo que possam ser colocadas em uma biblioteca.
7. Avalie as expressões abaixo, indicando qual é verdadeira e qual é falsa:

   a. 10==9+1
   b. 10 && 8
   c. 8 || 0
   d. 0 && 0
   e. considere x = 10 e y = 9;
   x>= 8 && y<=x

8. Fornecido o padrão binário, mostre o resultado binário do complemento de um.

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

9. Execute um AND bit-a-bit com os números binários:

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

10. Qual será o valor de count após a execução do seguinte código:

```
int x,*y,count;

func()
{
      x=100;
      count=999;
      y=&x;
      count=*y;
}
```

RESPOSTAS

1. char
   int
   short int
   long int
   unsigned int
   float
   double

2. char e int

3.
```
add (x,y)
int x,y;
{
     return(x+y);
}
```

4.
```
int counter;

func1()
{
     counter++;
}

func2()
{
     counter--;
}
```

5. Arquivo 1:

```
int counter;

func1()
{
     counter++;
}
```

Arquivo 2:

```
extern int counter;

func2()
{
     counter--;
}
```

6.
```
static int counter;

func1()
{
     counter++;
}

func2()
{
     counter--;
}

init()
{
     counter=0;
}
```

7. *a*. verdadeiro
   b. verdadeiro
   c. verdadeiro
   d. falso
   e. verdadeiro

8. 1 0 1 1 0 0 0 1

9. 0 1 0 0 1 1 0 0

10. 100

CAPÍTULO **4**

DECLARAÇÕES PARA CONTROLE DO PROGRAMA

Este capítulo apresenta as diversas declarações para controle do programa contidas em C. Estão incluídas as declarações condicionais **if** e **switch** e aquelas utilizadas na formação de malha. **while, for** e **do/while.**

DECLARAÇÕES CONDICIONAIS

C contém dois tipos de declarações condicionais: as declarações **if** e **switch.** No Capítulo 3 você aprendeu sobre o operador ?, que pode executar determinados tipos de teste condicional. O operador ? é uma alternativa da declaração **if**, em determinadas circunstâncias. Ambos serão discutidos nesse capítulo. A declaração **if** em C é muito parecida com a declaração IF-THEN do BASIC. A declaração **switch** é semelhante ao ON-GOTO do BASIC.

A DECLARAÇÃO if

A forma geral da declaração **if** é

```
if(condição de teste) declaração1
    else declaração2
```

onde os objetos de **if** e **else** são declarações simples. A declaração **else** é opcional. Os objetos, tanto de **if** como de **else**, podem ser blocos de declarações. A forma geral de **if** com blocos de declarações como objetos é

```
if(condição de teste)
{
   declarações/*bloco1*/
}
   else
{
   declarações/*bloco2*/
}
```

Caso a condição seja verdadeira (isto é, qualquer coisa diferente de zero), **declaração1** ou **bloco1** é executado; caso contrário, se existir um dos dois, é executado **declaração2** ou **bloco2**. Lembre-se de que apenas uma declaração ou bloco será executado, e não ambos.

```
/*
** programa do numero magico
*/
main()
{
int magic = 123; /* numero magico */
int guess;

guess = getnum(); /* leia um inteiro do teclado */
if(guess == magic) printf("** Certo **\n");
}
```

```
10 M=123;
20 INPUT G
30 IF M=G THEN PRINT "** CERTO **"
40 END
```

Figura 4.1 Programa do número mágico em C e BASIC

Por exemplo, a Figura 4.1 mostra em C e em BASIC um programa que imprime a mensagem ****CERTO**** quando você adivinha o número mágico. Este programa será denominado "Programa do Número Mágico" em discussões posteriores.

Esse programa utiliza o operador relacional == para determinar se a adivinhação feita coincide com o número mágico. Em caso afirmativo, a mensagem é impressa na tela.

A função **getnum( )** fornece um número inteiro, digitado no teclado. A função **getnum( )** é encontrada em muitos compiladores C, nas suas bibliotecas padrão. Caso não exista no seu, você pode utilizar aquela demonstrada aqui e incluí-la como parte do programa, quando necessário.

```
getnum()
{
char s[80];

gets(s,80);
return(atoi(s));
}
```

A Figura 4.2 mostra o Programa do Número Mágico com um passo adicional que imprime uma mensagem quando é introduzido o número errado.

```
/*
** programa do numero magico - melhoramento 1
*/
main()
{
int magic = 123; /* numero magico */
int guess;

guess = getnum(); /* leia um inteiro do teclado */
if(guess == magic) printf("** Certo **\n");
else printf(".. Errado ..\n");
}
```

```
10 M=123;
20 INPUT G
30 IF M=G THEN PRINT "** Certo **"
35 ELSE PRINT ".. Errado .."
40 END
```

Figura 4.2 Uma versão do número mágico em C e em BASIC, que imprime mensagens de erro

Outra melhoria é fornecer ao jogador uma informação da aproximação de cada adivinhação com o número mágico. A versão da Figura 4.3 utiliza blocos de declarações como objetos das declarações **if** e **else**. Lembre-se, dependendo de se colocar ou não o número mágico, apenas um dos blocos será executado.

Um dos aspectos mais confusos das declarações **if** em qualquer linguagem de programação são os ifs encadeados. Um **if** *encadeado* é uma declaração **if** que é o objeto de um **if** ou um **else**. Se você acha que viu um na Figura 4.3, está certo; o bloco **else** externo contém uma combinação **if-else**. Pelo fato do **if** interno estar contido em um bloco, não há confusão a respeito de sua função ou execução. Entretanto, no que segue:

```
if(x)
    if(y) declaração1;
    else declaração2;
```

a qual dos dois **if** é que **else** se refere? Em C o **else** é ligado ao **if** mais próximo que não tem uma declaração **else**. Neste caso, o **else** é associado à declaração **if(y)**. Caso você queira que o **else** fique associado ao **if(x)**, terá de utilizar chaves para forçar uma avaliação diferente; por exemplo,

```
/*
** programa do numero magico - melhoramento 2
*/
main()
{
int magic = 123; /* numero magico */int guess;

guess = getnum(); /* leia um inteiro do teclado */
if(guess == magic)
  {
  printf("** Certo **\n");
  printf("%d e' o numero magico\n",magic);
  }
else
  {
  printf(".. Errado ..\n");
  if(guess > magic) printf("Muito alto.\n");
  else printf("Muito baixo.\n");
  }
}
```

```
10 M=123;
20 INPUT G
30 IF M=G THEN GOSUB 100 ELSE GOSUB 200
40 END
100 PRINT "** Certo **"; M; " e' o numeor magico"
110 RETURN
200 PRINT ".. Errado .."
210 IF G > M THEN PRINT "Muito alto."
215 ELSE PRINT "Muito baixo."
220 RETURN
```

Figura 4.3 Outra versão do programa do número mágico em C (utilizando blocos de declarações) e em BASIC

```
if(x)
{
    if(y) declaração1;
}
else declaração2;
```

Else agora está associado a **if(x)** porque não é mais parte do bloco objeto de **if**. Em resumo, o **if** e o **else** estão no mesmo nível.

A CORRENTE if-else-if

Uma construção comum de programação é a corrente **if-else-if**. Assemelha-se ao seguinte:

```
if(condição)
   declaração;
else if(condição)
   declaração;
else if(condição)
   declaração;
.
.
.
else
   declaração;
```

As condições são avaliadas de cima para baixo. Assim que uma condição verdadeira é encontrada, a declaração a ela associada é executada e o restante da corrente é pulado. Caso nenhuma das condições seja verdadeira, o **else** final será executado. O **else** final freqüentemente age como uma *condição default*; isto é, caso todos os outros testes condicionais falhem, a última declaração **else** será executada. Caso o último **else** não exista, não haverá nenhuma ação se todas as outras condições forem falsas.

A Figura 4.4 mostra como ficaria o Programa do Número Mágico se fosse utilizada uma corrente **if-else-if**.

A ALTERNATIVA ?

Uma última alternativa de escrever o Programa do Número Mágico envolve o operador ?. Como você deve se lembrar do Capítulo 3, o operador ? avalia uma entre duas expressões, baseado na verdade de sua expressão condicional. Quando uma função é parte de uma expressão, ela é executada para conseguir seu valor. Você poderia escrever uma versão do Programa do Número Mágico, utilizando o operador ?, conforme indica a Figura 4.5.

```
/*
** programa do numero magico - versao 3
*/
main()
{
int magic = 123; /* numero magico */
int guess;

guess = getnum(); /* leia um inteiro do teclado */
if(guess == magic)
  {
  printf("** Certo **\n");
  printf("%d e' o numero magico\n",magic);
  }
else if(guess > magic)
       printf(".. Errado .. Muito alto.\n");
     else printf(".. Errado .. Muito baixo.\n");
}
```

Figura 4.4 Programa do número mágico com a corrente if-else-if

```
/*
** programa do numero magico
*/
main()
{
int magic = 123; /* numero magico */
int guess;

guess = getnum(); /* leia um inteiro do teclado */
if(guess == magic)
  {
  printf("** Certo **\n");
  printf("%d e' o numero magico\n",magic);
  }
else guess > magic ? printf("Alto.\n") :
                     printf("Baixo.\n");
}
```

Figura 4.5 Programa do número mágico, utilizando o operador ?

O operador ? na Figura 4.5 permite a execução de uma das declarações, dependendo do resultado do teste **guess>magic**. Após a avaliação, a primeira ou a segunda expressão torna-se o valor de toda operação. Como todas as outras funções C, **printf( )** pode ser utilizada em expressões. O valor de uma função é determinado pela sua execução, utilizando o seu valor devolvido. No caso da Figura 4.5, o valor devolvido não tem interesse, mas no processo todo, as séries apropriadas são impressas.

A DECLARAÇÃO switch

Apesar da corrente **if-else-if** conseguir executar uma série de testes, não pode ser considerada elegante. O código pode tornar-se difícil de seguir, e poderá inclusive confundir o programador posteriormente. Por essa razão é que C tem uma declaração de decisão, com múltipla ramificação e inerente à linguagem, denominada **switch**. A declaração **switch** age de modo semelhante ao **ON-GOTO** e **ON-GOSUB** do BASIC, comparando sucessivamente uma variável com uma lista de inteiros ou constantes de caracteres. Ao encontrar uma coincidência, uma declaração ou um bloco de declarações é executado. A forma geral da declaração **switch** é

```
switch(variável) {
   case constante1:
      declaração;
   case constante2:
      declaração;
   case constante3:
      declaração;
   .
   .
   .
   default:
      declaração;
   }
```

onde **default** será executado caso não seja encontrada nenhuma coincidência. O **default** é opcional, e se não estiver presente não haverá nenhuma ação caso a comparação falhe.

Switch é diferente do **if** porque testa apenas igualdade, ao passo que **if** pode avaliar uma expressão de relação ou lógica.

A declaração **switch** é freqüentemente utilizada para processar comandos do teclado, como a seleção do menu. Conforme é indicado aqui, com seu equivalente BASIC, a função **menu( )** apresentará um menu de um programa de verificação de soletração, chamando os procedimentos adequados:

```
menu()
{
char ch;
printf("1. Verifica ortografia\n");
printf("2. Corrige erros de ortografia\n");
printf("3. Mostra erros de ortografia\n");
printf("Tecle qualquer outra tecla para saltar\n");
printf("          Faca sua escolha: ");
ch = getchar(); /* leia a selecao */
switch(ch)
  {
  case '1' : check_spelling();
             break;
  case '2' : correct_errors();
             break;
  case '3' : display_errors();
             break;
  default  : printf("Selecionada nenhuma opcao\n");
  }
}
```

```
10 PRINT "1. Verifica ortografia"
20 PRINT "2. Corrige erros de ortografia"
30 PRINT "3. Mostra erros de ortografia"
40 PRINT "Tecle qualquer outra tecla para saltar"
50 PRINT "          Faca sua escolha: "
60 INPUT A
70 ON A GOSUB 100,200,300,400,400,400,400,400,400,400
80 END
100 REM VERIFICA ORTOGRAFIA
120 RETURN
200 REM CORRIGE ERROS
210 RETURN
300 REM MOSTRA ERROR
310 RETURN
400 PRINT "Selecionada nenhuma opcao"
410 RETURN
```

A versão em BASIC difere ligeiramente da versão C, por utilizar **INPUT**, que aguarda um retorno de carro. (A versão BASIC não utilizou o comando **INKEY$** porque versões mais antigas de BASIC não têm esse comando.)

A declaração **break** utilizada em cada **case** do **switch** faz com que o fluxo do programa saia da declaração **switch** e continue na próxima declaração, exteriormente ao **switch**. Caso não seja incluída a declaração **break**, todas as declarações, *na e abaixo da coincidência serão executadas.*

Você pode encarar **case** como uma etiqueta que indica onde a execução deve continuar após a leitura de uma seleção do teclado. Ao contrário de **break**, **case** não interrompe a execução. Esses aspectos de **switch** poderão ter suas vantagens em certos casos, conforme indicado:

```
func1()
{
int ch,flag;

ch = getnum();
flag = -1;
switch(ch)
  {
  case 1 :
  case 2 :
  case 3 : flag = 0;
           break;
  case 4 : flag = 1;
  case 5 : error(flag);
           break;
  }
}
```

```
10 INPUT C
20 F = -1
30 ON C GOTO 100,200,300,400,500
40 END
100 REM CONTINUE
200 REM CONTINUE
300 F =0
310 GOTO 40
400 F = 1
500 GOSUB 1000
600 GOTO 40
1000 REM CHAMA FUNCAO DE TRATAMENTO DE ERRO
1010 RETURN
```

Essa rotina ilustra duas facetas da declaração **switch**. Primeiro, você poderá ter condições vazias. Neste caso, as três primeiras constantes vão todas executar as mesmas declarações:

```
flag = 0;
break;
```

Segundo, a execução continua no próximo **case**, se não estiver presente uma declaração **break**. Caso **ch** coincida com 4, **flag** será colocado em 1, e pelo fato de não haver nenhuma declaração **break**, a execução continuará com a declaração **error(flag)**. Neste caso **flag** terá o valor 1. Se **ch** coincidisse com 5, **error(flag)** teria sido chamado com um valor de −1 para **flag**.

MALHAS

Em C, assim como em todas as modernas linguagens de programação, as *malhas* permitem que um conjunto de instruções seja executado até a obtenção de uma determinada condição. Essa condição pode ser previamente estabelecida, como na malha **for**, ou aberto como nas malhas **while** e **do-while**.

A MALHA for

A malha **for** é utilizada quando desejamos executar declarações mais de uma vez. A Figura 4.6 compara um programa C utilizando **for** e um programa BASIC que usa **FOR-NEXT**. Os dois programas vão imprimir a palavra **HELLO** e o valor de x, 100 vezes na tela. Na malha FOR-NEXT do BASIC, as linhas 10 a 30 serão executadas 100 vezes, incrementando x com 1 para cada execução. No código C, o valor inicial de x é colocado em 1. Já que x é menor que 100, **printf( )** será chamado; x é acrescido de 1 e é novamente testado para verificar se continua menor ou igual a 100. Esse processo se repete até que x seja maior que 100, quando a chamada pelo **print( )** cessa e o programa pára. Nesse exemplo, x é a *variável de controle da malha*, que é modificada e verificada para cada repetição da malha.

```
10 FOR X=1 TO 100 STEP 1
20   PRINT "HELLO ",X
30 NEXT
```

```
main()
{
int x;

for(x = 1;x <= 100;++x)
  printf("HELLO %d\n",x);
}
```

Figura 4.6 Uma comparação de **for** em C e **FOR-NEXT** em BASIC

A malha **FOR-NEXT** do BASIC e a malha **for** de C operam quase que da mesma forma: BASIC utiliza a palavra **NEXT** para definir o fim do código repetido, ao passo que o código iterativo de C é a declaração imediatamente posterior à definição **for** ou uma série de declarações em um bloco.

A forma geral de **for** para repetir uma declaração simples é

```
for(inicialização;condição;incremento)
    declaração;
```

Para a repetição de um bloco, a forma geral é

```
for(inicialização;condição;incremento)
{
   declaração 1;
   .
   .
   .
   declaração n;
}
```

A *inicialização* é normalmente uma declaração de atribuição utilizada para determinar a variável de controle da malha. A *condição* é uma expressão relacional que determina o fim da malha. O *incremento* define como a variável de controle da malha se altera a cada repetição dela. Essas três seções devem ser separadas por ponto-e-vírgula. A malha **for** continua a execução enquanto o resultado do teste é verdadeiro. Assim que a condição resulta falsa, a execução do programa continua na declaração que segue o bloco **for**.

Este é o exemplo de uma malha **for** que contém declarações múltiplas:

```
for(x = 100;x != 65;x = x - 5)
   {
   z = sqrt(x);
   printf("A raiz quadrada de %d, %f\n",x,z);
   }
```

Tanto as chamadas de **sqrt( )** como **printf( )** serão executadas enquanto x não ficar igual a 65. Observe que a malha se processa em *sentido negativo:* **x** foi inicializado em 100, e é feita uma subtração de 5 para cada repetição da malha.

Um ponto importante a ser observado a respeito das malhas **for** em C, é que o teste condicional é sempre feito no início da malha. Significa que o código no interior da malha poderá não ser executado caso a condição seja inicialmente falsa. Esta malha, por exemplo,

```
x = 10;
for(y = 10;y != x;++x)
   {
   printf("%d\n",y);
   }
printf("%d\n",y);
```

nunca será executada, porque **x** e **y** de fato são iguais. Após a malha, **y** continuará com o valor 10, e a única saída será uma impressão do número 10.

Uma malha **for** em C tem diversas possibilidades sem paralelo no BASIC. Por exemplo, múltiplas variáveis da malha podem ser utilizadas e suas condições verificadas, conforme indicado nesta malha:

```
for(x = 0,y = 0;x + y < 10;++x)
   {
   y = getchar();
   y = y - '0'; /* subtrai o codigo de 0 em ASCII */
   }
```

Uma vírgula separa as duas declarações de inicialização. Isto é necessário para que o compilador entenda que existem dois elementos de inicialização, e não uma inicialização e uma condição. Para cada incremento de x repete-se a malha e o valor de y é fornecido pelo teclado. Tanto x como y devem estar com o valor correto para o encerramento da malha. y deve ser inicializado com 0, de maneira que seu valor seja definido. No caso de y não estar definido, poderia ser possível que contivesse, devido ao acaso ou devido a uma utilização anterior, um 10. Isto tornaria o teste condicional falso e impediria a execução da malha.

De modo geral, a condição pode ser qualquer declaração C, relacional ou lógica. Não está limitada apenas ao teste das variáveis de controle da malha, como pode ser visto no seguinte exemplo:

```
sign_on()
{
char str[20]; /* serie de 20 caracteres */
int  x;

for(x = 0;x < 3 && strcmp(str,"password");++x)
   {
   printf("por favor, entre com a senha:");
   gets(str);
   }
if(x == 3) hang_up();
}
```

Você pode utilizar a função **sign_on( )** como uma medida de segurança para permitir acesso ao computador apenas àqueles que conhecem a senha. Caso o usuário digite a senha corretamente nas três primeiras tentativas, x será menor que 3 porque o teste condicional se torna falso. Caso contrário, a malha termina quando x for igual a 3.

A função **strcmp( )** é uma função da biblioteca padrão. Compara duas séries e devolve um 0 em caso de coincidência. Caso contrário, vai devolver a posição do caractere em que divergiram pela primeira vez.

Outra possibilidade do **for** permite que haja argumentos múltiplos nas porções de inicialização e incremento. Por exemplo, a seguinte malha é perfeitamente válida:

```
for(x = 0,y = 100;x < y;++x,--y)
   printf("%d %d\n",x,y);
```

Lembre-se de que as três partes principais da malha – a inicialização, a condição e o incremento – são separadas por ponto-e-vírgula. Todos os argumentos adicionais são separados por vírgulas.

Outro aspecto da malha **for**, que é diferente em C, é que partes da definição da malha poderão estar faltando. Se você quisesse escrever uma malha que é processada até que um determinado número aparecesse na tela, ficaria como segue:

```
for(x=0;x!=123; )
{
   x=getnum( ); /* pegue um número do teclado */
   .
   .
   .
}
```

A parte do incremento da definição **for** está em branco. Significa que para cada repetição da malha, é feito um teste para verificar se x é igual a 123, sem que haja outra ação. Se, porém, você digitar 123 pelo teclado, a condição da malha será falsa e a malha será encerrada. Ao contrário da malha **FOR-NEXT** do BASIC, que presume um incremento de 1 quando o comando **step** está faltando, a malha **for** em C não executa nenhuma ação quando a seção de incremento está faltando.

A inicialização pode ser externa à malha. Por exemplo,

```
x = 0;
for(;x < 10;)
   {
   printf(%d\n",x);
   ++x;
   }
```

Aqui a parte de inicialização foi deixada em branco e x foi inicializado antes da entrada na malha.

Mais eliminações ainda podem ser feitas da definição da malha. Você pode testar a condição dentro do código repetido e encerrar a malha com a utilização da declaração **break**, que força uma saída da malha. (**break** será discutido posteriormente neste capítulo.) Em seguida, o controle do programa continua no código que segue a malha, conforme indicado:

```
x = 10;
for(;;)
   {
   x = getchar(); /* le um caractere */
   if(x == 'A') break;  /* sai da malha */
   }
printf("voce digitou A\n");
```

Esta malha será processada até que um A seja digitado no teclado. *Malhas de atraso* são normalmente utilizadas em programas C. Este é um exemplo em C e em BASIC:

```
for(x = 0;x < 1000;++x);
```

```
10 FOR X=0 TO 1000
20 NEXT
```

Os dois códigos executam a mesma coisa: incrementam x até 1000 e não praticam outra ação. O ponto-e-vírgula é necessário na versão C porque **for** espera uma declaração, que pode estar vazia, ou um bloco de declarações.

A MALHA while

while é outra forma de malha inerente. A forma geral da declaração é

while(condição)declaração;

onde *declaração* pode ser uma única declaração ou um bloco de declarações que devem ser repetidas. A *condição* pode ser qualquer expressão em que a verdade é qualquer valor diferente de zero. A declaração é executada enquanto a condição é verdadeira. Quando a condição se tornar falsa, o controle do programa passará para a linha seguinte ao código da malha.

Diversas versões de BASIC têm uma declaração **WHILE-WEND**, que opera de modo semelhante à malha **while** em C. O exemplo que segue mostra as versões, em C e em malha BASIC, de uma rotina de entrada do teclado, que se repete até a digitação de A.

```
wait_for_char()
{
char ch;

ch = 0; /* inicializa ch */
while(ch != 'A') ch = getchar();
}
```

```
10 A$=""
20 WHILE A$<>"A"
30   INPUT A$
40 WEND
```

Na versão C observamos que o primeiro **ch** é inicializado com 0. Por ser uma variável local, seu valor não é conhecido quando **wait_for_char( )** é executado. A malha **while** inicia verificando se **ch** não é igual ao A. Como **ch** foi previamente inicializado com 0, o teste é verdadeiro e a malha é iniciada. Para cada operação no teclado, o teste é verificado. Uma vez que A é digitado, a condição torna-se falsa porque **ch** iguala-se a A, e a malha termina.

Da mesma forma que na malha **for**, as malhas **while** verificam a condição no início da malha significando que o código da malha poderá não ser executado. Isso elimina a necessidade de se executar um teste antes da malha. Uma boa ilustração dessa situação é a função **pad( )**, que vai acrescentar espaços no final de uma série, até um comprimento predeterminado. Caso a série já esteja no comprimento certo, não serão acrescentados espaços.

```
pad(s,length)
char *s;
int   length;
{
int l;

l = strlen(s); /* determina comprimento de s */
while(l < length)
  {
  s[l] = ' '; /* insere espaco */
  l++;
  }
s[l] = '\0'; /* series devem ser teminadas por nulo */
}
```

Em **pad( )** os dois argumentos são s, a série, e **length**, o número de caracteres que s deverá ter. No Capítulo 3 foi visto que ao passar uma série para uma função, basta utilizar seu nome de vetor sem indexação. Para receber a série (vetor de caracteres) você deve declarar o parâmetro formal s como sendo do tipo **char pointer**, utilizando a linha

```
char *s;
```

Isso permite que você acesse a série diretamente, elemento por elemento. (No Capítulo 5 será melhor explicado como passar séries para funções.)

Caso a série s tenha comprimento igual ou maior do que **length**, o código na malha **while** nunca será executado. Caso s seja menor do que **length**, **pad( )** acrescentará o número necessário de espaços à série. A função **strlen( )**, que pode ser encontrada na biblioteca padrão, devolverá o comprimento da série.

A existência de diversas declarações diferentes dentro do **while**, cada uma delas podendo encerrar a malha, é uma prática muito comum. Uma malha **while** poderá ter também apenas uma variável como condição e pode ser utilizada simplesmente para repetir o conjunto de instruções até o término do procedimento:

```
func1()
{
int working;

working = 1; /* i.e., verdadeiro */
while(working)
  {
  working = process1();
  if(working) working = process2();
  if(working) working = process3();
  }
}
```

Qualquer uma das três rotinas pode resultar falsa (0), provocando a saída da malha.

Não é obrigatória a existência de uma declaração no corpo da malha **while**. Por exemplo,

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

vai se repetir até que um A seja digitado no teclado. Caso você ache estranha a atribuição dentro da condição **while**, lembre-se de que o sinal = é apenas um operador no sentido em que devolve um valor: o valor do operador de atribuição é o valor da expressão do lado direito.

A MALHA do-while

Ao contrário das malhas **for** e **while** que testam a condição da malha no início, a malha **do-while** verifica a condição no final. Significa que uma malha **do-while** será executada ao menos uma vez. A forma geral da malha **do-while**:

```
do{
   declarações;
} while(condição);
```

Apesar das chaves não serem necessárias quando existe apenas uma declaração presente, são normalmente utilizadas para melhorar a leitura da construção **do-while**.

Não existe no BASIC um equivalente direto do **do-while**, de modo que quando esse tipo de laço for necessário nessa linguagem, deverá ser construído utilizando-se diversos GOTO. Temos aqui um exemplo bastante simples de um **do-while** em C e um GOTO em BASIC:

```
do                          10 INPUT N
  {                         20 IF N>100 THEN GOTO 10
  num = getnum();
  } while(num > 100);
```

Números serão lidos do teclado até que se encontre um número menor que 100.

Talvez a utilização mais comum do **do-while** seja na rotina de seleção de menu, onde o usuário faz uma seleção em um menu apresentado na tela. Quando é digitada uma resposta válida, ela é devolvida como o valor da função. Respostas inválidas forçam o programa a fazer uma nova solicitação. O que segue é uma melhoria de **menu( )**, do menu de verificação de escrita desenvolvido no início deste capítulo.

```
menu()
{
char ch;
printf("1. Verifica ortografia\n");
printf("2. Corrige erros de ortografia\n");
```

```
printf("3. Mostra erros de ortografia\n");
printf("Tecle qualquer outra tecla para saltar\n");
printf("          Faca sua escolha: ");
do
  {
  ch = getchar(); /* leia a selecao */
  switch(ch)
    {
    case '1' : check_spelling();
               break;
    case '2' : correct_errors();
               break;
    case '3' : display_errors();
               break;
    default  : printf("Selecionada nenhuma opcao\n");
    }
  } while(ch != '1' && ch != '2' && ch != '3');
}
```

No caso de uma função menu, é desejável que ela execute ao menos uma vez. Após a apresentação das opções, o programa será colocado em uma malha até que uma opção válida seja selecionada.

SAINDO DAS MALHAS COM A UTILIZAÇÃO DE break E exit( )

A declaração **break** e a função de biblioteca **exit( )** permitem que você force uma saída de uma malha, contornando a condição normal da malha.

A DECLARAÇÃO break

Quando a declaração **break** é encontrada dentro de uma malha, esta é imediatamente encerrada e o controle do programa reassume na declaração seguinte à malha. A Figura 4.7 fornece um exemplo simples juntamente com uma comparação em BASIC. Antes de encerrar, as duas versões imprimem os números de 0 até 10 na tela. (Observe que algumas versões de BASIC não suportam um **GOTO** fora de uma malha **FOR**.)

A declaração **break** normalmente é utilizada em malhas onde condições especiais podem provocar encerramento imediato. Este é o exemplo de uma situação assim, onde o acionamento de uma tecla pode interromper a execução do programa ou da rotina:

```
look_up(name)
char *name;
{
char tname[40];
```

```
do
   {
   read_next_name(tname);
   if(key_press()) break;
   } while(!strcmp(tname,name));
}
```

```
main()
{
int t;

for(t = 0;t < 100;t++)
   {
   printf("%d\n",t);
   if(t == 10) break;
   }
}
```

Figura 4.7 A utilização do **break** em um programa C com o equivalente BASIC

Você pode utilizar essa função para encontrar um nome em um arquivo de banco de dados. Caso o arquivo seja muito extenso e você esteja cansado de esperar, opere uma tecla e retorne da expressão, mais cedo. **read_next_name( ) e key press( )** são funções definidas pelo usuário. usuário.

strcmp( ) é uma função da biblioteca padrão. Compara duas séries, e devolve um 0 se forem iguais. Caso contrário devolve a posição do caractere em que diferiram pela primeira vez.

Um **break** provoca uma saída apenas da malha mais interna. Por exemplo,

```
for(t = 0;t < 100;++t)
   {
   count = 1;
   do
      {
      printf("%d\n",count);
      count++;
```

```
    if(count == 10)break;
    } while(1);          /* infinitamente */
}
```

imprimirá os números de 1 até 10 na tela, 100 vezes. Cada vez que o **break** é encontrado, o controle é devolvido à malha **for.**

Um **break** utilizado em uma declaração **switch** afetará apenas o **switch** e não a malha em que o **switch** possa estar.

A FUNÇÃO exit( )

Uma segunda maneira de encerrar uma malha de dentro é pela utilização da função **exit( )**, encontrada na biblioteca padrão. Pelo fato da função **exit( )** provocar um encerramento imediato de seu programa e um retorno ao sistema operacional, sua utilização é limitada. A função **exit( )** tradicionalmente é chamada com um argumento de 0, para indicar que a conclusão é normal. Outros argumentos são utilizados para indicação de certos erros. Porém, muitos compiladores C baseados em microcomputadores não utilizam o argumento para **exit( )**, de modo que com uma certa freqüência você verá **exit( )** utilizando 0 como argumento em todas as circunstâncias. Uma utilização comum de **exit( )** ocorre quando uma condição imperativa para a execução do programa não é satisfeita. Por exemplo, imagine um jogo em que uma placa de funções gráficas coloridas deve estar presente no sistema. A função **main( )** pode ficar como

```
main( )
{
   if(!color_card( )) exit (1);
   .
   .
   .
}
```

onde **color_card** é uma função definida pelo usuário que é verdadeira, caso a placa esteja presente. Se a placa não estiver no sistema, **color_card( )** é falso e o programa termina.

A função **exit( )** é também associada, com freqüência, às rotinas que abrem arquivos de discos. Esses tipos de rotinas serão discutidas no Capítulo 6.

A DECLARAÇÃO continue

A declaração **continue** opera de maneira praticamente oposta ao da declaração **break**: força a iteração seguinte da malha a ser executada, evitando qualquer código intermediário. Em BASIC seria utilizado um **GOTO**. Por exemplo,

```
do
   {
   x = getnum();
   if(x < 0) continue;
   printf("%d\n",x);
   } while(x != 100);
```

Aqui apenas os números positivos são impressos; uma resposta negativa obrigaria a malha a executar o teste de conclusão, x! = 100, e recomeçar tudo.

Nas malhas **while** e **do-while**, uma declaração **continue** fará com que o controle vá diretamente para a condição, para em seguida continuar com o processo da malha. No caso do **for**, o trecho de incremento da malha é executado, a condição é executada e a malha continua. Se você estender o exemplo anterior de maneira que ele pergunte apenas por um máximo de 100 números, poderá ser escrito da seguinte maneira:

```
for(t = 0;t <100;++t)
   {
   x = getnum();
   if(x < 0) continue;
   printf("%d\n",x);
   }
```

Como pode ser visto no exemplo seguinte, **continue** pode ser utilizado para acelerar a saída de uma malha, forçando a condição a ser executada mais cedo.

```
code()
{
char done,ch;

done = 0;
while(!done)
   {
   ch = getchar();
   if(ch == '$')
      {
      done = 1;
      continue;
      }
   putchar(ch + 1); /* desloca o alfabeto uma posicao */
   }
}
```

Você poderia utilizar esta função para codificar uma mensagem pelo deslocamento de todos os caracteres uma letra para frente; por exemplo, um "a" se tornaria um "b". A função termina quando um $ é digitado, porque a condição, tornada efetiva com **continue**, vai encontrar **done** como verdadeiro, forçando a saída do laço.

RÓTULOS e goto

Este livro não vai utilizar **goto** fora desta seção, porque em uma linguagem como C, que tem um conjunto rico de estruturas de controle e permite controle adicional utilizando **break** e **continue**, existe pouca necessidade para o **goto**. A preocupação principal da maioria dos programadores a respeito do **goto** é a sua tendência em confundir um programa tornando-o quase ilegível. Porém, existem ocasiões em que a utilização do **goto** torna o fluxo do programa mais claro em lugar de torná-lo confuso. Caso você esteja pensando em utilizar C como um substituto do assembly, deve ao menos ser apresentado ao **goto**, porque, sob certas circunstâncias, permite que o código seja curto e rápido.

Para sua operação o **goto** exige um rótulo (label). O **label** pode ser qualquer nome iniciado por uma letra ou um caractere de sublinha, seguido de letras, números, um número simples ou o caractere de sublinha. O nome do rótulo deve ser seguido por dois-pontos. Por exemplo, uma malha de 1 até 100 poderia ser escrita utilizando **goto**, como:

```
x = 1;
loop1:
x++
if(x < 100) goto loop1;
```

Uma utilização apropriada para **goto** é, para diversos níveis de aninhamento, uma maneira de sair. Neste exemplo

```
for(...) {
        for(...) {
                while(...) {
                        if(...) goto stop;
                        .
                        .
                        .
                }
        }
}
stop:
        printf("error in program \n");
```

a eliminação de **goto** faria com que um número de testes adicionais fossem executados. Uma declaração **break** não funcionaria neste caso porque sairíamos apenas da malha interna. Caso você substitua verificações em cada malha, o código ficará como abaixo:

```
done=0
for(...) {
        for(...) {
                while(...) {
                        if(...) [
                                done=1;
                                break;
```

```
                                    }
                                    .
                                    .
                                    .
                                {
                                if(done) break;
                            }
                            if(done) break;
                    }
```

goto deve ser utilizado com economia. Mas, caso o código seja mais difícil de ser lido ou se a velocidade de execução do código é crítica, utilize **goto**.

JUNTANDO OS PEDAÇOS

A Figura 4.8 mostra o Programa do Número Mágico na íntegra. Utiliza muito do que foi descrito neste capítulo, e você deve estar certo de que compreende todos os conceitos nele utilizados, antes de seguir para o próximo capítulo.

```
/*
** Programa do numero magico
*/
#include <stdio.h>

main()
{
char option;
int  magic;

do
  {
  printf("1. Define um novo numero magico\n");
  printf("2. Jogo\n");
  printf("3. Fim\n");  do
    {
    printf("Entre com sua opcao:");
    option = getchar();
    fflush(stdin);
    } while(option < '1' || option > '3');
  switch(option)
    {
    case '1' : magic = next_magic();
               break;
    case '2' : play(magic);
               break;
```

Figura 4.8 O Programa do número mágico na íntegra

```
        case '3' : printf("Tchau!!\n");
                   break;
        }
    } while(option != '3');
}

next_magic()
{
printf("Entre com novo numero magico:");
return(getnum());
}

getnum()
{
char s[30];

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

play(m)
int m;
{
register int t;
         int x;

for(t = 0;t < 100;t++)
  {
  printf("Advinhe o numero:");
  x = getnum();
  if(x == m)
    {
    printf("*** Certo ***\n");
    return;
    }
  else if(x < m) printf("Muito baixo.\n");
       else printf("Muito alto.\n");
  }
printf("Voce esgotou o numero de tentativas...");
printf(" Tente novamente.\n");
}
```

Figura 4.8 *(continuação)*

EXERCÍCIOS

1. Escreva uma função chamada **max( )**, que forneça o valor do maior de seus dois argumentos inteiros.
2. Escreva uma função denominada **look_up( )**, que tenha como argumento um caractere. Se o argumento é um dos caracteres seguintes, devolva os caracteres indicados; caso contrário forneça o caractere "0". Dica: você deve utilizar a declaração **switch**.

argumento	**retorno**
1	a
2	b
3	c
4	d

3. Escreva a função **look_up( )** do exercício anterior, utilizando uma corrente **if-else-if**.
4. Mostre três maneiras de escrever uma função chamada **count( )**, que apenas imprime os números de 1 até 100 na tela. (Não utilize **goto**.)
5. Escreva um programa que pegue um inteiro do teclado, imprima a mensagem **hello** o número de vezes de número, e pare caso o número seja negativo.
6. Compile e processe o Programa do Número Mágico final da Figura 4.8.
7. Melhore o Programa do Número Mágico de modo que ele informe ao jogador quantas adivinhações restam antes que termine o número de tentativas. Crie também um vetor de números inteiros com dez elementos, inicialize o vetor com alguns valores inteiros, e altere **next_magic( )** para que forneça o próximo elemento do vetor.

RESPOSTAS

```
1. max(a,b)
   {
   if(a > b) return(a);
   else return(b);
   }

2. look_up(c)
   char c;
   {
   switch(c)
```

```
    {
    case '1' : return('a');
    case '2' : return('b');
    case '3' : return('c');
    case '4' : return('d');
    default  : return('0');
    }
}
```

3.
```
look_up(c)
char c;
{
if(c == '1') return('a');
else if(c == '2') return('b');
     else if(c == '3') return('c');
          else if(c == '4') return('d');
               else return('0');
}
```

4.
```
count1()
{
int t;

for(t = 0;t < 100;++t) printf("%d\n",t);
}

count2()
{
int t;

t = 0;
while(t < 100) printf("%d\n",t++);
}

count3()
{
int t;

t = 0;
do
  {
  printf("%d\n",t);
  t++;
  }  while(t < 100);
}
```

5.
```
main()

{
int t;
t = getnum();
if(t < 0) exit();
for(;t > 0;t--) printf("hello\n");
}
```

7.
```
/*
** Programa Numero Magico melhorado
*/
#include <stdio.h>

int mcount;  /* esta variavel global ira indexar
                o vetor de numeros magicos m[] */
int m[10];   /* vetor de numeros magicos */

main()
{
char option;
int  magic;

init_magic();
do
  {
  printf("1. Define um novo numero magico\n");
  printf("2. Joga\n");
  printf("3. Fim\n");
  do
    {
    printf("Entre com sua opcao:");
    option = getchar();
    fflush(stdin);
    } while(option < '1' || option > '3');
  switch(option)
    {
    case '1' : magic = next_magic();
               break;
    case '2' : play(magic);
               break;
    case '3' : printf("Tchau!!\n");
               break;
    }
```

```
  } while(option != '3');
}

next_magic()
{
mcount++;
if(mcount > 9) mcount = 0; /* reinicie */
return(m[mcount]);
}

getnum()
{
char s[30];

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

}

play(m)
int m;
{
register int t;
         int x;

for(t = 0;t < 100;t++)
  {
  printf("Advinhe o numero:");
  x = getnum();
  if(x == m)
    {
    printf("*** Certo ***\n");
    return;
    }
  else if(x < m) printf("Muito baixo.\n");
       else printf("Muito alto.\n");
  printf("Voce tem mais %d tentativas restantes.\n",
          99-t);
  }
printf("Voce esgotou seu numero de tentativas...");
printf(" Tente novamente.\n");
}

init_magic()
{
```

```
mcount = 0;
m[0] = 123;
m[1] = 23456;
m[2] = 245;
m[3] = 4634;
m[4] = 345;
m[5] = 7865;
m[6] = 341;
m[7] = 17956;
m[8] = 19;
m[9] = 2032;
}
```

CAPÍTULO 5

FUNÇÕES EM DETALHES

As funções são blocos de construção em C, onde ocorrem todas as atividades do programa. Assim que uma função tenha sido escrita e depurada, poderá ser reutilizada quantas vezes for necessário. Este é um dos aspectos mais importantes de C como uma linguagem de programação. Este capítulo discute diversos aspectos a respeito da escrita e utilização das funções em C. Porém, ao contrário do que foi feito nos capítulos anteriores, não serão fornecidos exemplos comparativos em BASIC, porque as funções C e as funções BASIC são muito diferentes.

A FORMA DE UMA FUNÇÃO

Como você deve lembrar-se do Capítulo 1, a forma geral de uma função C é

```
nome_função(lista de parâmetros)
declarações dos parâmetros;
{
    corpo da função;
}
```

O número de parâmetros poderá ser zero. Se não houver parâmetros você não precisará do trecho de declaração.

VALORES DE RETORNO

Todas as funções devolvem valores. Este valor poderá ser especificado explicitamente pela declaração **return**, ou poderá ser 0 caso não seja especificado outro valor. Como condição normal

as funções devolvem valores inteiros. Como será discutido mais adiante, neste capítulo, outros tipos de valores podem ser devolvidos caso seja especificado.

Uma função pode ser utilizada em expressões porque cada função tem um valor que, ou é um valor devolvido ou é um 0, como situação normal. Portanto, cada uma das seguintes expressões são válidas em C:

```
x = power(y);
if(max(x,y) > 100) printf("maior\n");
for(ch = getchar();isdigit(ch);) ... ;
```

Mas, uma função não pode ser o alvo de uma atribuição; uma declaração do tipo

```
swap(x,y) = 100 /* comando incorreto */
```

está errada. Seu compilador dará uma indicação de erro e não compilará o programa.

Apesar de todas as funções em C terem valores devolvidos, ao escrever um programa, suas funções geralmente podem ser de três tipos. O primeiro tipo de função é projetado especificamente para executar operações em seus argumentos, e devolver um valor baseado nesta operação. Exemplos desse tipo de função são **sqrt( )** e **max( )**.

O segundo tipo de função manipula informação e devolve um valor que apenas indica o sucesso ou a falha dessa manipulação. Um exemplo é **write( )**, utilizado para escrever informação em um arquivo em disco. Caso a operação de escrita seja bem-sucedida, **write( )** devolverá um valor "verdadeiro"; caso não seja bem-sucedida, devolverá um valor "falso".

O último tipo de função não terá como devolução um valor explícito. Em resumo, a função é apenas um procedimento do qual não resulta valor. Um exemplo é **sort( )**, uma função que classifica dados. Uma vez que todos os dados podem ser classificados, a devolução de um valor não terá sentido, já que será sempre verdadeiro. (Entretanto, caso você tenha escrito uma função de classificação que opere em apenas um determinado tipo de dado, o retorno de um código de erro que indica entrada incorreta terá sentido.) Assim, apesar de todas as funções devolverem valores, não haverá necessariamente obrigação de utilizá-las para alguma coisa.

Uma pergunta muito comum referente a valores devolvidos de uma função é, "Não devo atribuir este valor para alguma variável, já que um valor está sendo devolvido?" A resposta é não. Se não há nenhuma atribuição especificada, o valor simplesmente é descartado. Por exemplo, na linha 1 do programa da Figura 5.1, o valor devolvido por **mul()** é atribuído para z. Na linha 2, o valor devolvido realmente não é atribuído, mas é utilizado pela função **printf( )**. Finalmente, na linha 3, o valor devolvido é perdido, porque nem é atribuído para outra variável, nem é utilizado como parte de uma expressão.

O ALCANCE DAS VARIÁVEIS

No Capítulo 3 você aprendeu a respeito do escopo e do tempo de vida das variáveis. Uma variável local é *dinâmica*; é criada quando a função é executada e destruída com a conclusão da função. Em outras palavras, uma variável local é reconhecida apenas pela função em que é declarada.

```
main()
{
int x,y,z;

x = 10;
y = 20;
z = mul(x,y);                    /* 1 */
printf("%d\n",mul(x,y));         /* 2 */
mul(x,y);                        /* 3 */
}
```

Figura 5.1 Um programa mostrando a utilização de valores de retorno de funções

Uma variável global é declarada fora de qualquer função e é conhecida por qualquer função do programa. As variáveis globais existem durante toda duração do programa.

Uma variável estática mantém seu valor dentro de uma função, de chamada em chamada. É conhecida apenas pela função ou arquivo e existe durante o programa.

ARGUMENTOS DE FUNÇÕES

Os argumentos das funções foram utilizados até aqui com poucas explicações, porque, geralmente, a utilização de um argumento de função é intuitiva. Dificilmente você terá de pensar a seu respeito ao escrever funções. Entretanto, como um programador em C, você deve entender como C passa os argumentos para as funções, de modo que possa controlar como os dados são manipulados pela utilização de algumas das características mais avançadas e eficientes de C.

CHAMADA POR VALOR E CHAMADA POR REFERÊNCIA

Os argumentos normalmente podem ser passados para as funções de uma entre duas maneiras. A primeira é denominada *chamada por valor*. Esse método copia o *valor* dos argumentos nos parâmetros formais da função. Com este método, as alterações realizadas nos parâmetros da função não surtirão efeito nas variáveis utilizadas para chamar a função.

A segunda maneira de passar argumentos para uma função é pela *chamada por referência*. Com este método, o *endereço* de cada argumento é copiado nos parâmetros da função. Isso significa que as alterações feitas nos parâmetros afetam a variável utilizada para chamar a função.

As funções C utilizam chamada por valor. Significa, de maneira geral, que você não pode alterar as variáveis utilizadas para chamar a função. (Você vai aprender mais adiante, neste capítulo, como "forçar" uma chamada por referência para permitir alterações nas variáveis de chamada.) Por exemplo, na função seguinte:

```
sqr(x)
int x;
{
x = x * x;
return(x);
}
```

caso o argumento de **sqr( )** seja um inteiro, como 10, o valor 10 será copiado no parâmetro **x**. Quando ocorre a atribuição x=x*x, a única alteração feita é na variável local **x**. A constante 10 não é afetada.

Ocorre o mesmo processo quando **sqr()** é chamado com uma variável. Por exemplo, caso **y** seja igual a 100, a chamada **sqr(y)** terá como resultado a cópia da variável y no parâmetro **x**. Sob nenhuma circunstância o valor de **y** será modificado. Lembre-se de que uma cópia do valor da variável ou da constante, utilizada para chamar uma função, é passada para a função. O que ocorre dentro da função não surtirá efeito sobre a variável utilizada na chamada.

Uma vez que todos os argumentos C são passados por valor, não é possível mudar as variáveis utilizadas na chamada. Porém, C permite que você simule uma chamada por referência, utilizando ponteiros para passar o endereço da variável para uma função e alterar a variável utilizada na chamada. Como você deve lembrar-se, do Capítulo 3, ponteiros são simplesmente endereços de variáveis. Como qualquer outro valor você pode passar um endereço para uma função. A função **swap( )**, que altera os valores de seus dois argumentos inteiros, é escrita como

```
swap(x,y)
int *x,*y;
{
int temp;

temp = *x; /* recebe valor no endereco x */
*x = *y; /* coloca valor de y em x */
*y = temp;
}
```

Mas isso é apenas parte da história. Após escrever a função **swap( )** você deverá utilizar *os endereços das variáveis que devem ser comutadas como argumentos, ao chamar a função*. A Figura 5.2 mostra a maneira correta de chamar **swap( )**.

O programa da Figura 5.2 estabelece o valor 10 para a variável **x** e o valor 20 para a variável **y**. Em seguida **swap( )** é chamado, e o operador unário & é utilizado para fornecer os endereços das variáveis **x** e **y**. Portanto, os endereços de **x** e **y**, e não os seus valores, são passados para a função **swap( )**.

```
main()
{
int x,y;

x = 10;
y = 20;
swap(&x,&y);
}
```

Figura 5.2 Um programa mostrando a maneira correta de chamar swap( )

Dentro do **swap( )**, o operador unário * é utilizado para indicar o valor do local apontado pelo endereço. Na declaração **temp=*x**, o operador * fornece o valor apontado pelo endereço em **x** e atribui para **temp** este valor. Na declaração ***x=*y**, o operador * "instrui" o valor do endereço **y** para ser colocado no endereço encontrado em **x**. Em seguida o local **y** é designado com o valor do endereço **x**.

O Capítulo 7 trata de ponteiros, porque são muito importantes em C. Entretanto, os conceitos básicos foram discutidos aqui, já que são tantas as funções que precisam alterar o valor das variáveis utilizadas como argumentos.

Para evitar resultados bizarros ou inesperados, antes que uma função receba um ponteiro, devem ser definidos os tipos de dados que serão endereçados. Apesar de ser possível misturar livremente inteiros e caracteres, em C, isto não é permitido na utilização de ponteiros.

CHAMANDO FUNÇÕES COM VETORES

Como você aprendeu no Capítulo 2, basta utilizar o nome do vetor de caracteres, sem indexação, caso deseje chamar uma função com um argumento série. O mesmo se aplica aos vetores passados como argumentos para funções. Porém, C não copia todo o vetor na função. Quando você chama uma função com um nome de vetor, o endereço do primeiro elemento do vetor é passado para a função. Significa que a declaração do parâmetro deve ser um ponteiro. Por exemplo, se você quisesse escrever um programa que colocasse 10 números do teclado e os imprimisse, poderia escrever um programa semelhante ao da Figura 5.3.

Na função **display( )** mostrada na Figura 5.3, apesar do argumento ser declarado como um ponteiro para inteiro, uma vez no interior da função, todo o vetor poderia ser acessado, utilizando a indexação normal de vetor. A razão disso é que os vetores em C, na verdade, são ponteiros para uma região de memória, e na verdade, o par [ ] é um operador que localiza o valor dos dados na indexação do vetor especificado entre as chaves. Pelo fato de C não apresentar nenhum limite na verificação de vetores, a função também não se preocupa com o tamanho do vetor (mas você deve).

```
/*
** Imprime numeros
*/
main()
{
int t[10],i;

for(i = 0;i < 10;++i) t[i] = getnum();display(t);
}

display(num)
int *num;
{
int i;

for(i = 0;i < 10;i++) printf("%d\n",num[i]);
}
```

Figura 5.3 Um programa que insere 10 números e os imprime

Um elemento de vetor utilizado como argumento é tratado como qualquer outra variável simples. Por exemplo, o programa da Figura 5.3 poderia ter sido escrito sem passar o vetor inteiro, conforme a Figura 5.4. O parâmetro de **display( )** é um inteiro. Não é pertinente que **display( )** seja chamado por meio de um elemento de vetor inteiro, porque apenas este valor do vetor é utilizado.

```
main()
{
int t[10],i;

for(i = 0;i < 10;++i) t[i] = getnum();
for(i = 0;i < 10;i++) display(t[i]);
}

display(num)
int num;
{
printf("%d\n",num);
}
```

Figura 5.4 Uma nova versão do programa da Figura 5.3, utilizando um elemento de vetor como argumento

```
/*
** Imprime serie de caracteres em maiusculas
*/
#include <stdio.h>

main()
{
char s[80];

gets(s);
print_upper(s);
}

print_upper(string)
char *string;
{
register int t;

for(t = 0;string[t];++t)
   {
   string[t] = toupper(string[t]);
   putchar(string[t]);
   }
}
```

Figura 5.5 Um programa que imprime um string em maiúscula

Entretanto, quando você utiliza um nome de vetor como argumento de uma função, ele é passado como referência. Você estará operando sobre e alterando potencialmente o conteúdo dos elementos do vetor utilizado para chamar a função. Por exemplo, o programa da Figura 5.5 imprime uma série em letra maiúscula. A função **toupper( )**, encontrada na maioria das bibliotecas C, converte um caractere minúsculo para um caractere maiúsculo. Após chamar **print_upper( )**, o conteúdo do vetor s em **main( )** será mudado para maiúscula.

Se você não tiver a intenção de mudar o vetor s permanentemente, pode reescrever o mesmo programa conforme indicado na Figura 5.6. Nesta versão, os conteúdos do vetor s permanecem inalterados, porque apenas seus valores – não o endereço – são passados para a função **print_upper_ch( )**.

Um exemplo clássico de passar vetores para funções é encontrado na função padrão **gets( )**, que você já utilizou em alguns exemplos. Após ter sido chamado com um argumento de vetor de caracteres simples, **gets( )** devolve uma série que é digitada no teclado. Apesar do **gets( )** que existe em sua biblioteca padrão ser muito mais sofisticado e complexo, a função apresentada na próxima relação lhe dará uma idéia de como ela opera. Para evitar confusão com a função padrão, esta será denominada **xgets( )**:

```
/*
** versao simplificada da funcao gets()
** da biblioteca padrao
*/
#include <stdio.h>

xgets(s)
char *s;
{
char ch;
int  t;

for(t = 0;t < 80;++t)
  {
  ch = getchar();
  switch(ch)
    {
    case '\n' : s[t] = '\0'; /* fim da serie */
                return;
    case '\b' : if(t > 0) t--;
                break;
    default   : s[t] = ch;
                break;
    }
  }
s[80] = '\0';
}
```

A função **xgets( )** deverá ser chamada com um ponteiro para caracteres, que pode ser uma variável declarada como sendo um ponteiro de caracteres, ou o nome de um vetor de caracteres, que por definição é um ponteiro de caracteres. Ao entrar, **xgets( )** estabelece uma malha **for**, de 0 até 80. Isso evita que séries muito longas sejam colocadas através do teclado. (A maioria das funções **gets( )** das bibliotecas padrão não apresenta limite; estamos incluindo uma, para ilustrar a verificação manual de limites para evitar extravasamentos de um vetor.) Caso sejam digitados mais que 80 caracteres, a função retornará. Já que C não apresenta nenhuma verificação automática de limites, você deve estar seguro de que qualquer variável utilizada para chamar **xgets( )** aceita ao menos 80 caracteres. À medida que você digita os caracteres, estes são colocados na série. Caso você digite um retrocesso, o contador t é decrementado. Ao operar um retorno de carro, será colocado um nulo no final da série, indicando sua conclusão. Pelo fato do vetor utilizado para chamar **xgets( )** ser modificado, no retorno vai conter os caracteres digitados.

Você não poderia escrever nem **swap( )** nem **xgets( )** sem utilizar ponteiros, para criar uma chamada por referência. Ao escrever esse tipo de função, lembre-se de que deve passar o *endereço* da variável que será alterada, para a função. Dentro da função, você deve executar todas as operações na variável com a utilização do operador unário *****.

```
/*
** Imprime serie de caracteres em maiusculas
*/
#include <stdio.h>

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

gets(s);
for(t = 0;s[t];++t) print_upper_ch(s[t]);
}

print_upper_ch(ch)
char ch;
{
ch = toupper(ch);
putchar(ch);
}
```

Figura 5.6 Uma nova versão do programa da Figura 5.5

O operador unário * pode ser pensado como "no endereço". Por exemplo, em

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

a primeira declaração de atribuição, x=&z;, pode ser lida como "atribua para x o endereço de z". A segunda declaração de atribuição, *x=10;, pode ser lida como "no endereço x, coloque o valor 10". A última declaração, y=*x;, pode ser lida como "y igual ao valor do endereço x".

Lembre-se do unário & como "o endereço de" e o * como "no endereço". Se você conseguir lembrar do que os operadores fazem, terá menos problemas na sua utilização.

OS ARGUMENTOS argc E argv

Algumas vezes é necessário passar informação ao programa durante seu processamento. O método genérico é passar a informação para a função **main( )** com a utilização de dois argumentos especiais e inerentes: **argv** e **argc**. São os únicos argumentos que **main( )** pode ter. A Figura 5.7

lista um programa curto, que imprime seu nome na tela desde que você o digite logo após o nome do programa. Se o seu nome fosse Tom e você chamasse o programa de "nome", deveria digitar **nome Tom** para processar o programa. O resultado do programa seria uma saída **Hello Tom.** Por exemplo, se estivesse ligado ao acionador A e estivesse processando MS-DOS, veria

```
A>nome Tom
Hello Tom
A>
```

após o processamento do programa **nome.**

```
/*
** Programa do nome
*/
main(arc,argv)
int   argc;
char *argv[];
{
if(argc != 2)
  {
  printf("Voce esqueceu de digitar o nome\n");
  exit(0);
  }
printf("Alo %s\n",argv[1]);
}
```

Figura 5.7 Um programa que imprime o nome do usuário na tela

Observe na Figura 5.7 que **main( )** tem os dois argumentos **argv** e **argc.** Até agora, não houve nenhum argumento para **main( ).** Os argumentos **argc** e **argv** são duas variáveis inerentes que permitem a utilização de argumentos da linha-de-comando. Um *argumento de linha-de-comando* é a informação que segue ao nome do programa, na linha de comando do sistema operacional. Por exemplo, quando você compila programas C, digita algo parecido com

```
> cc nome_do_programa
```

após a solicitação, onde **nome_do_programa** é o programa que você quer compilar. O nome do programa é passado ao compilador C como um argumento do programa.

Você pode escrever seus programas de modo que tirem proveito dos argumentos de linha-de-comando, utilizando **argc** e **argv. argc** contém o número de séries de linha-de-comando individuais. Cada série é separada por um espaço. Por exemplo,

```
run Spot, run
```

é feito de três séries, enquanto

```
Herb, Rick, Fred
```

é uma única série. **argc** será sempre ao menos 1, porque o nome do programa que você está executando conta como o primeiro argumento.

argv é um ponteiro para um vetor de séries. Você deve declarar **argv** exatamente como

```
char * argv [ ];
```

O colchete vazio indica que é um vetor de comprimento indeterminado. Você agora pode acessar os argumentos individuais indexando **argv**. Por exemplo, **argv[0]** apontará para a primeira série, que sempre é o nome do programa; **argv[1]** apontará para o primeiro argumento, e assim em diante.

```
/*
** Contador regressivo
*/
#include <stdio.h>

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

if(argc < 2)
  {
  printf("Voce deve dizer o comprimento da contagem\n");
  printf("na linha de comando.  Tente de novo.\n");
  exit(0);
  }
if(argc == 3 && !strcmp(argv[2],"display")) disp = 1;
else disp = 0;
for(count = atoi(argv[1]);count;--count)
  if(disp) printf("%d\n",count);
putchar(7);  /* sinal sonoro - na maioria
                dos computadores */
}
```

Figura 5.8 Um programa que utiliza argumentos de linha-de-comando

Um pequeno exemplo de utilização de argumentos de linha-de-comando é o programa denominado **countdown** mostrado na Figura 5.8. Ele fará uma contagem regressiva a partir de um certo valor e emitirá um som ao chegar no 0. A extensão da contagem é especificada pelo primeiro argumento de linha-de-comando, assumido como sendo um número. A série que contém o número é convertida em um inteiro, utilizando a função padrão **atoi( )** (ascii para inteiro), encontrada na biblioteca C, antes que o programa possa continuar. Caso a série **display** esteja presente como o segundo argumento de linha-de-comando, a contagem também será apresentada na tela. Em **countdown**, caso nenhum argumento tenha sido especificado, a mensagem de erro será impressa e a execução encerrada.

Caso você deseje acessar um elemento individual em uma das séries de comando, poderá fazê-lo pelo acréscimo de uma segunda referência indexada, para **argv**. Por exemplo, o programa da Figura 5.9 vai apresentar todos os argumentos com os quais ele foi chamado e apresentá-los na tela, um caractere por vez.

A chave dupla em **putchar(argv[t] [i])**; da Figura 5.9 pode parecer estranha para você, mas é perfeitamente válida. O primeiro índice acessa a série e o segundo acessa o caractere daquela série. Você aprenderá mais sobre este tipo de indexação no capítulo dedicado a vetores.

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

for(t = 0;t < argc;++t)
  {
  i = 0;
  while(argv[t][i])
    {
    putchar(argv[t][i]);
    i++;
    }
  }
}
```

Figura 5.9 Um programa que imprime todos os argumentos utilizados para chamá-lo

Normalmente você utilizaria **argv** e **argc** para colocar os comandos iniciais em seu programa. Teoricamente, você pode ter 32767 argumentos, mas a maioria dos sistemas operacionais não permite mais do que alguns. Você normalmente utiliza esses argumentos para indicar um nome de arquivo ou uma opção. A utilização de argumentos de linhas-de-comando dará ao seu programa uma aparência bastante profissional e facilitará a utilização do programa em arquivos-em-lote.

FUNÇÕES QUE RETORNAM VALORES NÃO-INTEIROS

Todos os exemplos de funções até o momento, devolveram apenas valores inteiros ou valores de caracteres, que automaticamente são convertidos para inteiros. O valor inteiro é a forma normal dos dados que as funções C devolvem quando não existe outra especificação. Você verá que a grande maioria de suas funções vai utilizar esta situação. Entretanto, existirão situações em que será necessário devolver outros tipos de dados.

As funções podem ser solicitadas para devolver em qualquer um dos tipos de dados inerentes encontrados em C. O método da declaração é semelhante aos das variáveis: são utilizados os mesmos especificadores de tipo, e o especificador de tipo precede o nome da função. A forma geral da declaração é

```
especificador_detipo nome_função(lista de parâmetros)
    declarações de parâmetros;
    {
        corpo de declaração da função;
    }
```

O especificador de tipo não precisa estar na mesma linha que o nome da função; por exemplo,

```
float fsum(x,y)
float x,y;
{
return(x + y);
}
```

O especificador de tipo informa ao compilador que tipo de dado a função terá de devolver. Essa informação é crítica para que o programa processe corretamente, porque tipos de dados diferentes têm tamanhos e representação interna diferentes. Um inteiro poderá ter um comprimento de 2 bytes, ao passo que um número em ponto flutuante poderá ter um comprimento de 8 bytes. Por exemplo, se sua função devolver um **float** e a função chamadora identificá-lo um inteiro, somente os dois primeiros bytes do **float** serão utilizados. Isto, sem dúvida, seria inútil.

Existe mais a fazer ao utilizar um valor devolvido diferente de um inteiro. Sua rotina de solicitação deverá saber que tipo de dado a função está devolvendo, de maneira que você deve declarar a função dentro da rotina chamadora. Por exemplo, a Figura 5.10 mostra como utilizar a função de ponto flutuante, **sum( )**, dentro do programa. A linha de declaração, além de declarar as variáveis **first** e **second** declara também a função **sum( )**. Isto apenas informa ao compilador que **sum( )** vai devolver um dado do tipo ponto-flutuante. Não declara uma variável com o nome **sum** e não afeta o tipo de dado que **sum( )** realmente devolve.

Caso haja um desencontro entre o tipo de dado que a função devolve e o tipo de dado que a rotina chamadora está esperando, você terá um resultado estranho e imprevisível. Caso as duas funções estejam no mesmo arquivo, o compilador descobre a diferença de tipo. Entretanto, caso

estejam em arquivos diferentes, o compilador não encontrará o erro. A verificação de tipo não é feita no tempo de ligação ou no tempo de processamento, mas apenas durante o tempo de compilação. Portanto, você deve tomar cuidado para que os dois tipos sejam compatíveis.

Conforme foi mencionado anteriormente, quando um caractere é devolvido de uma função, que é declarada como sendo do tipo inteiro, o valor do caractere é convertido em um inteiro. Pelo fato de C poder converter de caractere para inteiro e vice-versa, as funções que devolvem valores de caracteres com freqüência utilizam a conversão de tipo "default" em inteiros e não declaram especificamente uma função que devolve um tipo caractere.

```
main()
{
float first,second,sum();

first = 123.23;
second = 99.09;
printf("%f\n",sum(first,second));
}

float sum(a,b)
float a,b;
{
return a + b;
}
```

Figura 5.10 Um programa utilizando **sum( )**

RETORNO DE PONTEIRO

Apesar das funções que retornam ponteiros serem manipuladas exatamente da mesma maneira que qualquer outro tipo de função, devem ser discutidos alguns conceitos importantes.

Ponteiros de variáveis *não são* inteiros *nem* inteiros sem sinal. São ponteiros. O motivo dessa distinção deve-se ao fato de que ponteiros podem ser incrementados ou decrementados. Por exemplo, cada vez que um ponteiro for incrementado ele endereçará para o próximo item de dado de seu tipo. Já que cada tipo de dado pode ter comprimento diferente, C deverá saber para que tipo de dado o ponteiro está endereçando, a fim de endereçar para o próximo item de dado. A aritmética de ponteiros será explicada no Capítulo 7, mas é importante que você não tente utilizar inteiros para devolver endereços de variáveis.

Se você quisesse escrever uma função que devolvesse um ponteiro em uma série no local em que houve uma concordância de caracteres, poderia utilizar o seguinte:

```
char *match(c,s)
char c,*s;
{
int count;
count = 0;
while(c != s[count] && s[count] != '\0') count++;
return(&s[count]);
}
```

A função **match( )** vai tentar devolver um ponteiro para o endereço da série em que foi encontrada a primeira concordância com o caractere em c. Se não foi encontrada nenhuma concordância, será devolvido um ponteiro ao terminador nulo.

A Figura 5.11 fornece um programa curto que utiliza **match( )**. Este programa lê uma série e em seguida um caractere. Se o caractere estiver na série, o programa imprimirá a série a partir do ponto de concordância. Caso contrário, imprimirá a mensagem **nenhuma concordância encontrada**.

```
main()
{
char s[80],*p;

gets(s);
ch = getchar();
p = match(ch,s);
if(p != 0) /* houve encontro */
  printf("%s\n",p);
else printf("nao foi encontrado\n");
}
```

Figura 5.11 Um programa utilizando **match( )**

RECURSÃO

Em C, as funções podem chamar a sí mesmas. Uma função é dita *recursiva* se uma declaração no corpo da função chama a ela mesma. Algumas vezes chamada *definição circular*, recursão é o processo de definir alguma coisa em termos de si mesmo. O BASIC padrão não permite a recursão.

Existem muitos exemplos de recursão. Um modo recursivo de definir um número inteiro é dizendo que é 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, mais ou menos um número inteiro. Por exemplo, o número 15 é o número 7 mais o número 8; 21 é 9 mais 12; e 12 é 9 mais 3.

Para que uma linguagem de computador possa ser recursiva, uma função deve ter condições de chamar a si mesma. Um exemplo simples é a função **factr( )**, que determina o fatorial de um inteiro. O fatorial de um número é o produto de todos os inteiros entre 1 e o número. Por exemplo, 3 fatorial é $1 \times 2 \times 3$, ou 6. Temos aqui uma versão não-recursiva, denominada **fact( )**:

```
fact(n) /* nao recursivo */
int n;
{
int t,answer;

answer = 1;
for(t = 1;t < n;t++) answer = answer * (t);
return(answer);
}
```

Esta é a versão recursiva **factr( )**:

```
factr(n) /* recursivo */
int n;
{
int answer;

if(n == 1) return(1);
answer = factr(n - 1) * n;
return(answer);
}
```

A versão não-recursiva de **fact( )** deve estar clara: utiliza uma malha iniciada em 1 e termina no número n, e progressivamente multiplica cada número pelo produto móvel.

Quando o **factr( )** recursivo é chamado com um argumento 1, a função entrega 1; caso contrário entrega o produto de **factr(n–1)*n**. Para avaliar esta expressão, **factr( )** é chamado com **n–1**. Isso acontece até que **n** se iguala com 1 e as funções começam a voltar.

Se você quisesse determinar o fatorial de 2, a primeira chamada de **factr( )** obrigaria que fosse feita uma segunda chamada com o argumento 1. Esta chamada entrega 1, que será em seguida multiplicado por 2 (o valor original de **n**). A resposta será 2. Em um dos exercícios no final deste capítulo, você colocará declarações de impressão em **factr( )**, que vai mostrar em que posição cada chamada se encontra e quais são as respostas intermediárias.

Quando uma função chama a si própria, novas variáveis locais e novos parâmetros recebem um local de armazenamento na pilha, e o código da função é executado desde o começo com essas novas variáveis. Uma chamada recursiva não faz uma nova cópia da função. Apenas os argumentos são novos. Com o retorno de cada chamada recursiva, as variáveis locais e os parâmetros antigos são removidos da pilha, e a execução é retomada no ponto de chamada da função, dentro dela. Podemos dizer que as funções recursivas vão e voltam de fora para dentro.

A maioria das rotinas recursivas não economiza significativamente em tamanho de código ou armazenamento de uma variável. A versão recursiva da maioria das rotinas poderá demorar um pouco mais para ser executada, em virtude do acréscimo de chamadas às funções, que na maioria dos casos não será perceptível. Muitas chamadas recursivas de uma função poderiam causar um estouro da pilha, o que é pouco provável. Porque o armazenamento dos parâmetros da função e variáveis locais é feito na pilha, e porque cada nova chamada cria uma nova cópia dessas variáveis, é possível que a pilha pudesse "sobrepor" algum outro dado ou memória de programa. Você, porém, provavelmente não precisará se preocupar com isto, a não ser que a função recursiva fuja ao controle.

Uma função recursiva poderá ser, algumas vezes, de maior clareza e simplicidade para ser escrita do que a mesma função iterativa. Algumas pessoas pensam recursivamente com maior facilidade que outras. Caso você se sinta à vontade com a recursão, então utilize-a. Caso contrário, utilize os métodos iterativos. São poucos os exemplos deste livro que serão recursivos, e quando o forem, a razão será dada.

Ao escrever funções recursivas, deve existir uma declaração **if** em algum lugar, forçando a função a voltar sem que seja executada a chamada recursiva. Se isso não for feito e você chamar a função, ela nunca voltará. Esse é um erro muito comum ao se escrever funções recursivas. Utilize **printf( )** e **getchar( )** com liberdade durante o desenvolvimento, para que você possa verificar o que está acontecendo e abortar a execução caso tenha cometido um erro.

ESCREVENDO AS SUAS PRÓPRIAS FUNÇÕES

Você já escreveu algumas funções, agora porém está na hora de tratar dos parâmetros e da eficiência.

PARÂMETROS E FUNÇÕES DE MÚLTIPLOS PROPÓSITOS

Você não deve basear as funções de propósito múltiplo em dados globais. Toda a informação necessária para uma função deve ser passada para ela pelos seus parâmetros. Nos poucos casos em que isso não é possível, você deve utilizar as variáveis estáticas.

Além de fazer sua função multifuncional, você deve utilizar parâmetros em lugar das variáveis globais, para manter seu código legível e menos susceptível a falhas, que resultam dos efeitos secundários.

EFICIÊNCIA

A utilização das funções melhora sensivelmente a legibilidade e a eficiência da maioria dos programas e ajuda na prevenção de erros decorrentes de efeitos secundários. Porém, em determinadas aplicações especializadas, é provável a necessidade de se colocar o código "na linha" em vez de na função. A *única vez* em que você deverá restringir a utilização das chamadas de função será quando o tempo de execução for crítico.

Duas são as razões que explicam porque um código de linha é mais rápido que uma chamada de função. Primeiro, uma instrução de chamada leva tempo para ser executada. Segundo, se existirem argumentos que devem ser passados, estes deverão ser colocados na pilha, o que também leva tempo. Para a maioria das aplicações, este pequeno aumento no tempo de execução não tem significado. Mas caso o tempo de execução seja importante, lembre-se de que cada chamada de função utiliza um tempo que seria economizado se o código da função fosse colocado na linha. Por exemplo, a Figura 5.12 fornece duas versões de um programa que imprime o quadrado dos números de 1 até 10. A versão de-linha será processada mais rapidamente que a outra, porque a chamada de função leva tempo.

```
direto

main()
{
int x;

for(x = 1;x < 11;++x)
  printf("%d\n",x * x);
}
```

```
por chamada de funcao

main()
{
int x;

for(x = 1;x < 11;++x)
   printf("%d\n",sqr(x));
}

sqr(a)
int a;
{
return a *  a;
}
```

Figura 5.12 As versões em-linha e a chamada de função de um programa que imprime o quadrado dos números de 1 até 10

ARQUIVOS E BIBLIOTECAS

Após ter escrito uma função, você pode utilizá-la de três maneiras: pode deixá-la no mesmo arquivo que a função **main( )**, pode colocá-la em um arquivo diferente, com outras funções que você escreveu, ou pode colocá-la em uma biblioteca.

JULGANDO O TAMANHO DO ARQUIVO DE UM PROGRAMA

Quando você começa a escrever os programas C, a tendência é deixar todas as suas funções no mesmo arquivo que **main( )**. Isso é aceitável inicialmente porque os seus primeiros programas tendem a ser curtos. Entretanto, à medida que você desenvolve programas maiores, vai deparar-se com diversos problemas.

Primeiro, o tempo de compilação é relacionado diretamente com o tamanho de programa que está sendo compilado. Um dos bons aspectos de C é que uma vez depurada uma função, você pode colocá-la em um arquivo separado e juntá-la com um encadeamento. O processo de encadeamento é muito mais curto que a compilação, e elimina a necessidade de recompilar um código de trabalho constantemente.

Segundo, os arquivos grandes normalmente são difíceis de ser editados, principalmente em um ciclo de desenvolvimento e depuração de um programa. Isso pode causar a perda de um tempo valioso e aumentar suas frustrações.

Para a maioria dos microcomputadores, um arquivo-fonte com mais do que 10000 bytes deveria ser desdobrado em diversas partes e compilados separadamente. A maioria dos programadores prefere trabalhar com arquivos que tenham menos do que 5000 bytes.

ARQUIVOS SEPARADOS

Antes que você inicie o desdobramento de suas funções em arquivos compilados separadamente, é importante fazer um planejamento prévio. Uma das tarefas mais frustrantes, mas comum, que você terá de fazer enquanto opera com programas grandes, é procurar cada arquivo, tentando achar onde foi que colocou uma determinada função. Um pouco de organização prévia ajuda a evitar esse problema.

Primeiro, agrupe todas as funções que são conceitualmente relacionadas em um arquivo. Por exemplo, se você estiver escrevendo um editor de texto, pode colocar todas as funções requeridas na deleção de texto em um arquivo, todas aquelas que procuram texto em outro, e assim em diante.

Segundo, ajunte todas as funções de propósito geral do seu programa. Por exemplo, nos programas de banco de dados as funções de formatação de entrada/saída pertencem a um arquivo separado porque serão utilizadas por diversas outras funções.

Terceiro, agrupe funções de nível superior em arquivos separados ou no arquivo **main( )**, se couber. As funções de nível superior são utilizadas para iniciar a atividade principal do programa. Essas rotinas definem essencialmente a operação do programa.

Alguma organização de seus arquivos de função é importante para, posteriormente, você encontrar uma determinada função. Existem diversas maneiras de organizá-las e você deve encontrar aquela que melhor se adapta ao seu estilo de programação.

BIBLIOTECAS

Uma biblioteca de funções difere de um arquivo de funções compilado separadamente. Em uma biblioteca, apenas aquelas funções que o seu programa realmente utiliza são carregadas e unidas ao seu programa. Em um arquivo compilado separadamente, *todas* as funções daquele arquivo são carregadas e ligadas ao seu programa. Para a maioria dos arquivos de função que você vai criar, provavelmente precisará de todas as funções do arquivo de qualquer modo. No caso de uma biblioteca C padrão, você nunca vai querer *todas* as funções ligadas ao seu programa, porque isso tornaria o seu código-objeto muito grande.

Poderá haver ocasiões em que você vai desejar criar uma biblioteca. Por exemplo, vamos supor que você escreveu um conjunto muito especial de funções estatísticas. Você não desejaria que todas essas funções fossem carregadas em seu programa, se este apenas precisasse encontrar a média de algum conjunto de valores. Neste caso, uma biblioteca seria bastante útil.

A maioria dos compiladores C inclui instruções de como criar uma biblioteca. Já que o processo varia de compilador para compilador, você deve estudar o manual do usuário para encontrar o procedimento a ser seguido.

EXERCÍCIOS

1. Escreva uma função denominada **pr_rev( )** que leia uma série colocada através do teclado e imprima-a inversamente. Por exemplo, hello seria impresso olleh.
2. Encontre o que há de errado com a função seguinte e tente corrigir o defeito:

```
next_record() /* devolve o proximo registro
                 do arquivo */
{
int next;

next++;
seek(next);
read_record();
}
```

3. Escreva uma função denominada **power( )** que aceite dois argumentos inteiros. O primeiro argumento é a base e o segundo é o expoente. Escreva **power( )** de maneira que o primeiro argumento contenha a resposta após o retorno da função. Utilize o valor devolvido da função para indicar estouro matemático; isto é, entregue 1 se a operação foi bem-sucedida ou 0 caso tenha havido estouro. Assuma que os dois argumentos são positivos. Dica: se houver estouro matemático, o número se torna negativo.
4. Modifique o programa **countdown** da Figura 5.8, de modo a permitir que os argumentos **count** e **display** fiquem em qualquer ordem. Por exemplo,

   ```
   countdown 1000 display
   ```

   é o mesmo que

   ```
   countdown display 1000
   ```

5. Escreva duas funções **code( )** e **decode( )** que aceitem uma série como argumento. A função **code( )** deveria modificar o argumento série pela adição de 1 em todos os caracteres, com exceção do terminador nulo. A função **decode( )** restaura a série codificada para a sua forma original.

6. Utilizando as funções **code( )** e **decode( )** do exercício anterior, escreva um programa curto que aceite uma série na linha de comando, imprima a série codificada e em seguida imprima a série decodificada. No caso de nenhuma série ser especificada na linha de comando, solicite uma.

7. Escreva uma função recursiva denominada **print_num( )** que tenha como argumento um inteiro. Vai imprimir os números de 1 a n na tela, onde n é o valor do argumento.

RESPOSTAS

1.
```
pr_rev()
{
int t;
char s[80];

gets(s);
for(t = strlen(s);t;t--) putchar(s[t - 1]);
}
```

2. A variável local **next** é dinâmica e não vai manter um determinado valor entre chamadas de função. A solução é fazer de **next** uma variável **estática**.

3.
```
power(x,y)
int *x,*y;
{
int temp;
temp = *x;
if(*y == 0)
  {
  *x = 1; /* potencia de zero */
  return 1;
  }
(*y)--;
for(;*y;(*y)--)
  {
  *x = (*x) * temp;
  if(*x < 0) return 0;      }
return 1;
}
```

4.
```
/*
** Contador regressivo
*/
#include <stdio.h>

main(argc,argv)
int   argc;
char *argv[];
{
int  disp,count;
char d[80],c[80];

if(argc<2)
  {
  printf("Voce deve dizer o comprimento da contagem\n");
  printf("na linha de comando.  Tente de novo.\n");
  exit(0);
  }
if(argc == 3)
  {
  if(!strcmp(argv[1],"display"))
    {
    strcpy(d,argv[1]);
    strcpy(c,argv[2]);
    }
  else
    {
    strcpy(d,argv[2]);
    strcpy(c,argv[1]);
    }
  }
if(argc == 3 && !strcmp(d,"display")) disp = 1;
else disp = 0;
for(count = atoi(c);count;--count)
  if(disp) printf("%d\n",count);
putchar(7);  /* sinal sonoro - na maioria
                dos computadores */
}
```

5.
```
code(s)
char *s;
{
register int t;
for(t = 0;s[t];++t)  s[t] = s[t] + 1;
}
```

```
   decode(s)
   char *s;
   {
   register int t;

   for(t = 0;s[t];++t)  s[t] = s[t] - 1;
   }

6. main (argc,argv)
   int   argc;   char *argv[];
   {
   char s[80];

   if(argc != 2)
     {
     printf("Digite sua mensagem:");
     gets(s);
     }
   else strcpy(s,argv[1]);
   code(s);
   printf(s);
   printf("\n");
   decode(s);
   printf(s);
   }

   code(s)
   char *s;
   {
   register int t;

   for(t = 0;s[t];++t)  s[t] = s[t] + 1;
   }

   decode(s)
   char *s;
   {
   register int t;

   for(t = 0;s[t];++t)  s[t] = s[t] - 1;
   }

7. print_num(n)
   int n;
   {
```

```
if(n == 1) printf("%d \n",n);
else
   {
   print_num(n - 1);
   printf("%d \n",n);
   }
}
```

CAPÍTULO **6**

ENTRADAS, SAÍDAS E ARQUIVOS DE DISCO

Este capítulo discute como ler o console e escrever nele e nos arquivos de disco. Você vai aprender sobre as diversas funções da biblioteca padronizada C, projetada para ajudá-lo nessa tarefa.

Lembre-se de que não existem funções inerentes ao C, para a execução das operações de E/S. Todas essas funções são encontradas na biblioteca padronizada C. Nos capítulos anteriores, você já utilizou as funções **printf( )**, **getchar( )** e **gets( )**, que são apenas algumas das funções que estão à sua disposição.

Em C, toda E/S é *orientada a caracteres.* Isso inclui não apenas leitura e escrita ao console (isto é, teclado e vídeo), mas as funções para disco também. Difere do BASIC, onde você pode ler e escrever séries e números diretamente. Em C você pode ler e escrever bytes. Como vimos, nas funções **gets( )** e **getnum( )**, é possível escrever funções que lêem séries e números, mas essas funções ainda utilizam as chamadas às funções de E/S, orientadas a caracteres.

E/S AO CONSOLE

E/S ao console refere-se às operações que ocorrem no teclado e no vídeo de seu computador. Apesar de você já ter utilizado algumas dessas funções, esta seção discutirá e esclarecerá alguns aspectos importantes de sua utilização.

Caso você conheça apenas E/S no BASIC, observe que E/S em C é completamente diferente. Em BASIC, toda E/S ao console é executada por funções inerentes de alto nível. Apesar do BASIC manter **INKEY$** para a leitura de um caractere, não é a forma mais importante de entrada do console.

AS FUNÇÕES getchar( ) e putchar( )

As funções mais simples de E/S ao console são **getchar( )**, que lê um caractere da entrada padrão (normalmente o teclado), e **putchar( )**, que imprime um caractere na saída padronizada (normalmente o vídeo).

A função **getchar( )** aguarda até o pressionamento de uma tecla, para em seguida devolver o seu valor. Normalmente, **getchar( )** também ecoa, automaticamente, o caractere digitado para o vídeo. Isso significa que o caractere digitado no teclado será escrito explicitamente no vídeo, para ser apresentado sem ser ecoado. Não aparecerá automaticamente porque não existe ligação entre o teclado e o vídeo.

A função **putchar( )** vai escrever o argumento do caractere no vídeo de seu computador apenas se o argumento for parte do conjunto de caracteres que seu computador pode apresentar.

O programa da Figura 6.1 insere caracteres pelo teclado e imprime-os em modo reverso: maiúsculas em minúsculas e minúsculas em maiúsculas. O programa pára ao se digitar um ponto.

A função **islower( )** devolve uma condição de "verdade" se **ch** for um caractere minúsculo. A função **toupper( )** converte uma letra minúscula em maiúscula e **tolower( )** converte uma letra maiúscula em minúscula. Essas funções não afetam os caracteres não-alfabéticos como + ou ?. Elas são encontradas na biblioteca padrão.

```
main()
{
char ch;

do
  {
  ch = getchar();
  if(islower(ch)) putchar(toupper(ch));
  else putchar(tolower(ch));
  } while(ch != '.'); /* use ponto para parar */
}
```

Figura 6.1 Um programa que coloca caracteres e os imprime em modo reverso

AS FUNÇÕES gets( ) e puts( )

No próximo passo, em termos de complexidade e força, temos as funções **gets( )** e **puts( )**. Permitem que você leia e escreva séries de caracteres do console.

A função **gets( )** retorna uma série terminada por nulo em seu argumento de vetor de caracteres. Isto significa que quando você utiliza **gets( )**, pode digitar caracteres no teclado até a operação do retorno de carro. O retorno de carro coloca um terminador nulo no fim da série

e **gets( )** retorna. O retorno de carro em si não estará contido na série, e é impossível utilizar **gets( )** para executar um retorno de carro; entretanto, **getchar( )** pode fazê-lo. **Gets( )** permite que você faça correção de erros (ou enganos) utilizando retrocesso (BACKSPACE) antes de operar RETURN.

A função **puts( )** escreve uma série no vídeo, recebido como parâmetro série. **puts( )** reconhece os mesmos códigos especiais que **printf( )**, como \ n para nova linha. Uma chamada à **puts( )** exige menor esforço que a mesma chamada para **printf( )**, porque **puts( )** coloca apenas uma série de caracteres no vídeo. Não pode colocar números ou fazer conversões de formato. Portanto, **puts( )** ocupa menos espaço e é mais rápido que **printf( )**, na apresentação de séries. Apesar de alguns compiladores terem abandonado essa função, **puts( )** está na maioria das bibliotecas-padrão. Se você não tem um **puts( )** disponível, um exemplo simples mostrado aqui funcionará bem. Caso você o compare com a maioria das funções **printf( )**, este **puts( )** é bem menor:

```
puts(s)
char *s;
{
register int t;

for(t = 0;s[t];++t) putchar(s[t]);
}
```

A função **puts( )** é importante quando o tamanho do código também o é. Se um programa não requer todas as funções de **printf( )** será vantajoso não utilizar uma função do tamanho de **printf( )** quando uma função do tipo de **puts( )** for suficiente. Por exemplo, temos aqui uma versão melhorada da função **getnum( )** daquela mostrada no Capítulo 3:

```
getnum() /* versao 2 */
{
char num[80],n;

do
  {
  gets(num);
  if(!number(num))
    {
    puts("Deve ser numero.\n");
    n = 0;
    }
  else n = 1;
  } while(!n);
return(atoi(num));
}
```

```
number(s);
char *s;
{
int t;

for(t = 0;s[t];++t) if(!isdigit(s[t])) return 0;
return(1);
}
```

Esta versão obriga que você coloque um número porque a função de apoio, **number( )** verifica que todos os caracteres da série **num** sejam dígitos. A utilização de **puts( )** em lugar de **printf( )** significa que, ao utilizar **getnum( )**, não será arrastada junto uma função do porte de **printf( )**. Lembre-se, caso qualquer função de seu programa exija **printf( )**, será carregada no momento da encadeação, de modo que é importante utilizar as menores funções nas funções padrões.

Tabela 6.1 As funções mais simples de E/S de console

Função	Operação
getchar( )	Lê um caractere do teclado
putchar( )	Escreve um caractere no vídeo
gets( )	Lê uma série do teclado
puts( )	Escreve uma série no vídeo

As funções mais simples que executam operações de entrada e saída no console estão resumidas na Tabela 6.1.

E/S DO CONSOLE FORMATADAS

Além das funções simples de E/S ao console descritas na seção anterior, a biblioteca padrão C contém duas funções que executam formatações de entrada e saída: **printf( )** e **scanf( )**. *Formatação de E/S* significa que essas funções podem moldar a informação sob sua orientação. Lembre-se de que, as rotinas simples, descritas na seção anterior, retiram e colocam informação na sua forma crua.

Você já tem uma certa familiaridade com **printf( )**. O complemento de **printf( )**, **scanf( )**, permite que seja feita a leitura de diversos tipos de dados do teclado, inclusive caracteres, séries e números. Tanto **printf( )** como **scanf( )** permitem a mistura de formatos de dados e a utilização de, por exemplo, especificadores de campo e pontos decimais.

A FUNÇÃO printf( )

Você pode pensar em **printf( )** como uma combinação da declaração PRINT e da declaração avançada PRINT USING do BASIC. Para resumir o que você aprendeu sobre **printf( )** no Capítulo 2, a forma geral de **printf( )** é

printf("série de controle",lista de argumentos);

A série de controle consiste em dois tipos de itens. O primeiro tipo será feito com caracteres que serão impressos na tela. O segundo tipo contém comandos de formatação que definem a maneira como os argumentos subseqüentes são apresentados. Deve existir o mesmo número de comandos de formatação que o número de argumentos, e os comandos de formatação e os argumentos são combinados em ordem. Por exemplo, a chamada **printf( )**

```
printf("Hi %c %d %s\n",'c',10,"there!");
```

apresenta na tela

Hi c 10 ali !

Tabela 6.2 Os códigos de controle de formatação de **printf( )**

Código printf( )	Formatação
%c	Um caractere simples
%d	Decimal
%e	Notação científica
%f	Ponto flutuante
%g	Utiliza %e ou %f, qual for mais curto
%o	Octal
%s	Série de caracteres
%u	Decimais sem sinal
%x	Hexadecimal

A Tabela 6.2 relaciona novamente a informação fornecida pela Tabela 2.1 (no Capítulo 2). Os códigos de controle de formatação poderão ter modificadores que especifiquem a largura do campo, o número de casas decimais e um indicador de alinhamento à esquerda. Um inteiro colocado entre o sinal % e o comando de formatação atua como um *especificador de largura-de-campo mínimo,* o que preenche a saída com brancos ou zeros para assegurar que tenha ao menos um comprimento mínimo. Caso uma série ou um número sejam maiores que o mínimo, serão completamente impresso, mesmo que ultrapassem o mínimo. O preenchimento normal é feito com espaços. Caso você queira preencher com zeros, basta colocar um zero antes do especificador de largura-de-campo. Por exemplo, **%05d** vai preencher um número com menos de 5 dígitos, com 0's.

Para especificar o número de casas decimais impressas para um número em ponto flutuante, coloque um ponto decimal seguido pelo número de casas decimais que você quer apresentar, após o especificador da largura-de-campo. Por exemplo, **%10.4f** apresentará um número com pelo menos 10 caracteres de comprimento, com quatro casas decimais. Esse método também funciona quando você quer especificar o comprimento máximo do campo, para séries e valores inteiros. Por exemplo, **%5.7s** apresentará uma série com pelo menos cinco caracteres de comprimento e que não exceda sete. Caso a série seja mais comprida que a máxima largura do campo, os caracteres serão truncados na extremidade final.

Como condição normal, toda saída é alinhada *pela direita*: se a largura do campo for mais larga que os dados impressos, os dados serão colocados no lado direito do campo. Você pode forçar a informação a ser colocada no lado esquerdo, pela colocação de um sinal de menos logo após %. Por exemplo, **%–10.2f** fará um alinhamento à esquerda de um número em ponto flutuante, com duas casas decimais, em um campo com 10 caracteres de largura.

O modificador **l** informa **printf( )** que um tipo de dado **long int** segue.

Com **printf( )**, você pode extrair qualquer tipo de formato de dado que desejar, como pode-se verificar nos exemplos da Tabela 6.3. Escrevendo os seus próprios exemplos, e verificando os seus resultados, você determina se compreende completamente o processo.

Tabela 6.3 Exemplos de formatação da saída **printf( )**

declaração printf( )	**saída**
("%–5.2f",123.234)	|123.23 |
("%5.2f",3.234)	| 3.23|
("%10s","hello")	| hello|
("%-10s","hello")	|hello |
("%5.7s","123456789")	| 1234567|

A FUNÇÃO scanf( )

A função **scanf( )** é uma rotina de entrada de utilização geral, que permite que você leia dados formatados e converta automaticamente informação numérica em inteiros e **floats**, por exemplo. É quase que o reverso de **printf( )**. A forma geral de **scanf( )** é:

scanf(série de controle,lista de argumentos);

A série de controle é formada por códigos de formatação de entrada, precedidos por um sinal %. Estes códigos estão relacionados na Tabela 6.4.

Os comandos de formatação podem utilizar modificadores de largura de campo, formado por números inteiros colocados entre o sinal % e o código de formatação. Um * colocado após o % vai cancelar a designação e avançar para o próximo campo de entrada. Qualquer outro caractere na série de controle será combinado e descartado.

Tabela 6.4 Os códigos de formatação **scanf( )**

Código	Significado
c	Leia um único caractere
d	Leia um inteiro decimal
e	Leia um número em ponto flutuante
f	Leia um número em ponto flutuante
h	Leia um inteiro curto
o	Leia um número octal
s	Leia uma série
x	Leia um número hexadecimal

Os itens de entrada de dados devem ser separados por espaços, tabulações ou linhas novas. Os símbolos de pontuação, como vírgulas e ponto-e-vírgulas, não são considerados separadores. Como em **printf( )**, os códigos de formatação **scanf( )** combinam as variáveis, recebendo a entrada em ordem.

Todas as variáveis utilizadas para receber valores por **scanf( )** devem ser passadas através de seus endereços. Significa que todos os argumentos, diferentes da série de controle, devem ser ponteiros para as variáveis que vão receber os valores de entradas. Lembre-se, esta é uma maneira utilizada por C para criar uma "chamada por referência". Por exemplo, se você quiser ler um inteiro para a variável **count**, deverá utilizar a seguinte chamada **scanf( )**:

```
scanf("%d",&count);
```

Séries serão lidas em vetor de caracteres, e o nome do vetor, sem indexação, é o endereço do primeiro elemento do vetor. Assim, para ler uma série para o endereço (**address**) do vetor de caracteres, você utilizaria

```
scanf("%s",address);
```

Neste caso, **address** já é um ponteiro e não precisa estar precedido pelo operador &.

O *modificador de máxima largura de campo* pode ser aplicado aos códigos de formatação. Se você quisesse ler um máximo de 20 caracteres para **address**, escreveria

```
scanf("%20s",address);
```

Caso o fluxo de entrada fosse maior que 20 caracteres, seria iniciada uma chamada subseqüente à entrada, no ponto em que esta chamada parou. Por exemplo, se

```
1100 Parkway Ave, apt 2110 B
```

foi colocado em resposta à última chamada de **scanf( )**, apenas os primeiros 20 caracteres, ou até o **p** de **apt**, teriam sido colocados em **address**, em virtude do especificador de tamanho-máximo. Significa que os 8 caracteres restantes, **t 2110 B**, não foram utilizados ainda. Caso você faça outra chamada **scanf( )**, como

```
scanf("%s",str);
```

então **t 2110 B** será colocado em **str**. Porém, muitos sistemas operacionais de microcomputadores perderiam caracteres digitados que não seriam atribuídos para algo. Apenas quando o sistema tivesse áreas auxiliares (buffers) nos dispositivos de E/S, é que esses caracteres estariam em condição de uso para processamento.

Apesar de espaços, tabulações e caracteres de linha-nova serem utilizados como separadores de campo, quando um caractere simples estiver sendo lido, estes serão lidos como qualquer outro caractere. Por exemplo, com uma lista de entrada **x y**,

```
scanf("%c%c%c",&a,&b,&c);
```

voltará com o caractere **x** em **a**, um espaço em **b**, e o caractere **y** em **c**.

Tome cuidado: se você tiver qualquer outro caractere na série de controle – inclusive espaços, tabulações e caracteres de linha-nova – esses caracteres serão utilizados para comparar caracteres da lista de entrada. Qualquer caractere que combine com um deles será descartado. Por exemplo, dada a lista de entrada **abcdttttttefg**,

```
scanf("%st%s",name1,name2);
```

colocará os caracteres **abcd** em **name1** e os caracteres **efg** em **name2**. Os **t** são descartados devido ao **t** na série de controle. Em outro exemplo,

```
scanf("%s ",name);
```

não retorna antes que você digite um caractere *após* digitar um terminador: o espaço após %s instruiu **scanf( )** para ler e descartar espaços, tabulações e linha-nova.

Ao contrário da declaração **INPUT** do BASIC, é impossível acionar uma solicitação como parte da chamada **scanf( )**. A série de controle não pode ser utilizada para escrever caracteres como em **printf( )**. Portanto, todas as solicitações devem ser feitas explicitamente antes da chamada a **scanf( )**.

A capacidade de **scanf( )** em processar diversos tipos de dados é freqüentemente utilizada em programas de banco de dados. Será desenvolvido um programa curto que mantém uma lista dos últimos nomes e um balanço atualizado de clientes. Isto ajudará na ilustração da utilização de E/S formatada do console.

Utilizando uma técnica top-down, primeiro você vai escrever a função **main( )** do programa. O programa será arbitrariamente restringido a 50 clientes. O último nome do cliente terá um máximo de 19 caracteres acrescido de um terminador nulo, dando um comprimento total de 20 caracteres para o campo. Existirão cinco procedimentos na operação do programa. Que são:

- Insira um cliente
- Procure e mostre um cliente
- Atualize um cliente
- Cancele um cliente
- Saia do programa

Uma declaração **switch( )** acessa esses cinco procedimentos. Cada entrada no **switch( )** chama a função correta para manipular uma das opções. A Figura 6.2 mostra como a função **main( )** fica no programa de banco de dados de cliente.

A primeira função que precisaremos é **init( )**, que inicializa o banco de dados com zeros:

```
init()
{
register int t ;

for(t = 0;t < 1000;++t) name[t] = '\0';
}
```

Preencher com zeros o vetor **name**, simplifica as funções que são utilizadas para manipular esse dado. A razão pela qual **balance** não é inicializado deve-se ao fato de que todas as referências ao banco de dados serão pelo nome do cliente, e o próprio balanço é também referenciado por este nome.

Para inserir dados no banco de dados, precisamos da função **enter( )**:

```
enter()
{
char s[20];
int  slot;

for(slot = 0;slot < 50;slot++)
 {
 if(!name[slot * 20]) break; /* achou aberto */
 }
if(slot == 50)
```

```
{
printf("Lista de clientes cheia.\n");
return 0;
}
printf("Entre nome e saldo:\n");
scanf("%19s%d",&name[slot * 20],&balance[slot]);
}
```

Em **enter( )** encontra-se a primeira utilização de **scanf( )**. **scanf( )** lerá até 19 caracteres do sobrenome do cliente, antes de ler o resultado do balanço. Pelo fato de **scanf( )** parar de ler em um espaço, é possível inserir apenas o último nome. Para conseguir ler o primeiro e o segundo nomes seriam necessárias de duas variáveis. Por exemplo, se a entrada do teclado, para a primeira posição fosse

```
Johnston 123
```

name[0] conteria **Johnston'\0'** e **balance[0]** seria **123.**

Após a colocação de um nome, você pode recuperar o balanço atualizado, procurando no sobrenome do cliente. A primeira das duas funções necessárias para fazer isso é **find( )**, uma função de pesquisa de uso geral que entrega o número do registro ou **slot** do dado:

```
find()
{
char s[20];
int  slot;

printf("Entre o nome do cliente:\n");
scanf("%19s",s);
fflush(stdin);
for(slot = 0;slot < 50;slot++)
  if(!strcmp(s,&name[slot * 20])) break;
     /* encontrou, abandone a malha */
if(slot == 50)
  {
  printf("Cliente nao esta na lista.\n");
  return(-1);
  }
return slot;
}
```

Quando o número do registro é conhecido, o nome e o balanço do cliente podem ser mostrados, utilizando-se a segunda função, **display( )**:

```
display(slot)
int slot;
{
printf("\nO saldo de %s %s e' %d\n",&fname[slot * 20],
        &name[slot * 20],balance[slot]);
}
```

As informações de um cliente podem ser atualizadas, chamando **update( )** com o número do registro que deve ser alterado; utilizamos **scanf( )** para inserir a nova informação:

```
update(slot)
int slot;
{
printf("Entre novo sobrenome, nome e saldo:");
scanf("%19s%19s%d",&fname[slot * 20],&name[20 * slot],
                   &balance[slot]);
}
```

Note que serão lidos apenas um máximo de 19 caracteres. Desta maneira sobrará espaço para colocar o terminador nulo no final.

Com a utilização de **delete( )** fica fácil eliminar um cliente do banco de dados, o que coloca um nulo na primeira posição de caractere do **slot**:

```
delete(slot)
int slot;
{
printf("Entrada retirada.\n");
name[20 * slot] = '\0';
balance[slot] = 0;
}
```

Se você digitar e compilar estas funções, poderá se divertir brincando com este programa, acrescentando funções aqui e acolá para fazer coisas novas.

LENDO E ESCREVENDO ARQUIVOS

Existem poucos programas úteis que não "lêem" nem "escrevem" arquivos de dados. No banco de dados de cliente da seção anterior, todos os dados são perdidos com a conclusão do programa. Esta é uma limitação séria à sua utilidade. Nesta seção, você vai aprender como ler e escrever um arquivo de disco.

Basicamente existem duas maneiras distintas de ler e escrever arquivos de disco em C. A primeira maneira é E/S de alto nível: todas as leituras e escritas são feitas um caractere por vez.

```
#include <stdio.h>
char name[1000];  /* 50 nomes de cliente,
                     cada um com 20 chars */
int balance[50];

main()  /* programa de base de dados
           de clientes simples */
{
char ch;
int  client;

init();  /* inicializa a base de dados com 0 */
do
  {                          /* malha principal */
  printf("Suas opcoes:\n");
  printf("\n\n  1.Inserir um novo cliente.\n");
  printf("   2.Procurar um cliente.\n");
  printf("   3.Atualizar um cliente.\n");
  printf("   4.Retirar um cliente.\n");
  printf("   5.Sair.\n");
  do
    {
    printf("Selecione opcao: ");
    ch = getchar ();
    fflush(stdin);
    } while(ch < '1' || ch > '5');
  printf("\n");
  switch(ch)
    {
    case '1' : enter();
               break;
    case '2' : client = find();
               if(client != -1) display(client);
               break;
    case '3' : client = find();
               if(client != -1) update(client);
               break;
    case '4' : client = find();
               if(client != -1) delete(client);
               break;
    case '5' : break;
    }
  } while(ch != '5');
}
```

Figura 6.2 Banco de dados de cliente

Isso normalmente é denominado *E/S com memória intermediária*, porque você não precisa se preocupar com tamanhos de setor, comprimento do buffer e outras considerações dependentes do sistema operacional. Em outras palavras, essas funções fornecem o seu próprio sistema de memória intermediária. A segunda maneira é E/S de baixo nível, que com freqüência é denominado *tipo-UNIX*. Com este método, toda leitura e escrita terá de ser feita manualmente, fornecendo suas próprias memórias intermediárias e ponteiros.

ARQUIVOS DE CABEÇALHO

Devemos fazer uma pequena digressão, antes que você aprenda a respeito de E/S de discos. C tem duas declarações que simplificam sobremaneira certos aspectos da programação. O primeiro é **#define**, utilizado para definir uma série de caracteres como uma constante. Por exemplo, esta declaração faz com que a série **MAX_NUM** seja representada por 100:

```
#define MAX_NUM 100
```

Observe que não é utilizado nenhum sinal de igual e que a declaração não termina em ponto-e-vírgula.

A declaração **#define** obriga o compilador a fazer uma macrossubstituição – neste caso o número 100 – cada vez em que a série **MAX_NUM** é encontrada. Uma macrossubstituição é simplesmente a substituição de um texto por outro, à medida que o seu programa está sendo compilado. No caso das séries, a macro freqüentemente é mais curta e mais fácil de trabalhar no programa. No caso de números, a utilização da macro permite que uma constante, utilizada durante todo o programa, seja facilmente alterada utilizando a declaração **#define**. A utilização de nomes em lugar de números fornece significado àquelas constantes e ajuda ao leitor do programa a entender melhor o que está acontecendo. Você poderia então escrever legalmente

```
for(t = 0; t<MAX_NUM; ++t) ...
```

Você pode ainda utilizar **#define** para executar macrossubstituições em séries, como

```
#define err1  "Erro de sintaxe"
```

Você poderia então escrever, legalmente, a declaração

```
printf(err1);
```

A segunda declaração que você deve conhecer é **#include**, que é utilizada durante o tempo de compilação para ler outro arquivo-fonte, normalmente informação de cabeçalho, em seu programa. Por exemplo, se você tem diversos arquivos para um programa denominado **spreadsheet**, pode ser que haja interesse para criar um arquivo de cabeçalho padrão, contendo todas as variáveis globais necessárias pelos diversos arquivos. Caso este arquivo fosse denominado **spreadsheet**, na declaração que segue seria a primeira linha de seu programa:

```
#include "spreadsheet.h"
```

O motivo pelo qual discutimos aqui **#include** e **#define** é que você vai precisar deles para utilizar as funções de arquivo em disco, fornecidas com seu compilador. Todo arquivo que utiliza E/S de disco exige que você leia um arquivo de cabeçalho denominado **stdio.h**, fornecido pelos criadores de seu compilador C. Deve estar no mesmo disco em que estão seus arquivos de programa. Você deve colocar a linha

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

próxima à primeira linha de cada arquivo de programa. Esse arquivo não vai definir apenas algumas estruturas de dados dependentes do sistema, mas vai definir também certas macros, como **EOF** para fim-de-arquivo. O manual de seu compilador C deveria dizer-lhe exatamente quais são as macros definidas e disponíveis para a sua utilização.

E/S DE ARQUIVO DE ALTO-NÍVEL

No sistema de E/S de alto-nível em C, existem cinco funções essenciais. São:

- **fopen( )**, que abre um arquivo para ser utilizado
- **putc( )**, que escreve um caractere em um arquivo
- **getc( )**, que lê um caractere de um arquivo
- **fclose( )**, que fecha um arquivo
- **fseek( )**, que é utilizado para executar operações randômicas em disco

A FUNÇÃO fopen( )

A função **fopen( )** serve a dois objetivos: primeiro, abre um arquivo em disco para ser utilizado, e em segundo lugar, devolve um ponteiro de arquivo. A forma geral de **fopen( )** é

```
FILE *fp;
fp = fopen(nomearquivo,modo);
```

onde "modo" é uma série contendo **r** de read (ler), **w** de write (escrever) ou **a** de acrescentar. Normalmente o modo ler/escrever é especificado com a série **rw**. (Seu compilador poderá ter modos adicionais, de maneira que você deve verificar o seu manual.) O nome do arquivo deve ser uma série de caracteres que formem um "nomedearquivo" válido para o sistema operacional.

A variável **fp**, do tipo **FILE**, é o ponteiro de arquivo. **FILE** é um tipo específico de dado definido em **stdio.h** para você, pelo projetista do compilador. Todos os ponteiros de arquivo devem ser declarados como sendo do tipo **FILE**. (Alguns compiladores poderão chamar esse **tipo** de dado por um nome diferente, de modo que você deve conferir sua documentação.)

Se você quisesse abrir um arquivo para escrita, com o nome **test**, escreveria

```
fp = fopen("teste","w");
```

Entretanto, você normalmente o veria escrito do seguinte modo:

```
if((fp = fopen("teste","w")) == NULL)
  {
  puts("nao pode abrir arquivo\n");
  exit();
  }
```

Esse método detecta qualquer erro na abertura do arquivo, como um disco protegido contra escrita ou completamente cheio, antes de tentar escrever nele. Um nulo, que normalmente é um 0, é utilizado porque nenhum ponteiro de arquivo terá o valor 0.

Se você utilizar **fopen( )** para abrir um arquivo para escrita, qualquer arquivo anterior com esse nome será destruído e um arquivo novo iniciado. Se você quiser fazer acréscimos ao final do arquivo, deverá utilizar o modo **a**.

A FUNÇÃO putc( )

A função **putc( )** é utilizada para escrever caracteres em um arquivo em disco que foi aberto utilizando **fopen( )** com modo **w**. A forma geral da função é

```
putc(c,fp);
```

onde **fp** é o ponteiro do arquivo devolvido por **fopen( )** e **c** é o caractere que será escrito. O ponteiro de arquivo informa **putc( )** em que arquivo de disco escrever.

A FUNÇÃO getc( )

A função **getc( )** é utilizada para ler os caracteres de um arquivo aberto em modo de leitura com **fopen( )**. A forma geral da função é

```
char ch;
ch = getc(fp);
```

onde **fp** é um ponteiro de arquivo do tipo **FILE** devolvido por **fopen( )**. O ponteiro de arquivo informa a **getc( )** de qual arquivo deve ler.

Quando o fim de arquivo for atingido, **getc( )** devolverá um marcador de fim-de-arquivo. O arquivo de cabeçalho **stdio.h** utiliza **#define** para gerar a macro **EOF** que será o marcador de fim-de-arquivo. Portanto, para ler até o marcador de fim-de-arquivo você poderia utilizar o seguinte código:

```
ch = getc(fp);
while(ch!= EOF) {
        .
        .
        .
     ch=getc(fp);
}
```

Já que o marcador de fim-de-arquivo não pode ser um caractere de impressão, não tente imprimi-lo na tela.

A FUNÇÃO fclose( )

A função **fclose( )** é utilizada para fechar um arquivo que foi aberto com uma chamada **fopen( )**. *Você tem de fechar todos os arquivos antes de terminar o programa.* **fclose( )** faz mais do que simplesmente liberar o ponteiro de arquivo; escreve qualquer dado que ainda não tenha sido escrito no disco e realiza um fechamento normal a nível de sistema operacional no arquivo. Uma falha no fechamento do arquivo é um convite a todo tipo de problemas, incluindo a perda de dados, destruição de arquivos e eventuais erros intermitentes em seu programa.

A forma geral da função **fclose( )** é

```
fclose(fp);
```

onde **fp** é o ponteiro de arquivo devolvido pela chamada **fopen( )**.

UTILIZANDO fopen( ), getc( ), putc( ) e fclose( )

Antes de aprender a respeito de **fseek( )**, você saberá como as funções individuais que formam a E/S de alto nível operam em conjunto.

Um exemplo simples de utilização de **fopen( )**, **putc( )** e **fclose( )** é o programa **ktod** da Figura 6.3. (O próximo programa vai incluir **getc( )**.) Simplesmente lê os caracteres do teclado e escreve-os no arquivo de disco até que a> seja digitado. O nome do arquivo é especificado a partir da linha de comando.

Em **ktod**, os macros **CR** e **LF** foram definidos (utilizando **#define**) como sendo seus valores decimais, de modo que quando um retorno for digitado no teclado, a seqüência correta de retorno-de-carro/alimentação-de-linha poderá ser colocada no arquivo do disco. Sem essa seqüência, apenas alimentação-de-linha (ou retorno-de-carro, em alguns sistemas) seria escrito no arquivo do disco.

Utilizando **getc( )**, o programa complementar **dtos**, listado na Figura 6.4, lerá qualquer arquivo ASCII e colocará o conteúdo no vídeo.

```
#include "stdio.h"

#define CR    13
#define LF    10

main(argc,argv) /* ktod - tecla para disco */
int   argc;
char *argv[];
{
FILE *fp;
char ch;

if(argc != 2)
  {
  printf("Voce esqueceu o nome do arquivo\n");
  exit(0);
  }
if((fp = fopen(argv[1],"w")) == NULL)
  {
  printf("nao pode abrir arquivo\n");
  exit(0);
  }
do
  {

  ch = getchar();
  if(ch == '\n')
    {
    putc(CR,fp); /* escreve combinacao cr/lf */
    putc(LF,fp);
    }
  else putc(ch,fp);
  } while(ch != '>');
fclose(fp);
}
```

Figura 6.3 O programa **ktod**, que utiliza **fopen( )**, **putc( )** e **fclose( )**

As funções de E/S de alto nível e suas operações em nada se parecem com as operações de disco do BASIC. Entretanto, caso você conheça o BASIC, as versões em BASIC dos programas **ktod** e **dtos** estão nas Figuras 6.5 e 6.6. Existem ligeiras diferenças na maneira como operam, porque o BASIC não emula com facilidade o sistema de E/S de orientação-por-caractere do C.

```
#include "stdio.h"
main(argc,argv) /* dtos - disco para video */
int   argc;
char *argv[];
{
FILE *fp;
int   ch;

if(argc != 2)
  {
  printf("Voce esqueceu o nome do arquivo\n");
  exit(0);
  }
if((fp = fopen(argv[1],"r")) == NULL)
  {
  printf("nao pode abrir arquivo\n");
  exit(0);
  }
ch = getc(fp); /* le um caractere */
while(ch != EOF)
  {
  putchar(ch); /* mostra no video */
  ch = getc(fp);
  }
fclose(fp);
}
```

Figura 6.4 O programa dtos, que utiliza getc( )

```
10 INPUT F$ : REM KTOD - TECLA PARA DISCO
20 OPEN  F$ FOR OUTPUT AS #1
30 INPUT A$
60 PRINT #1,A$
80 IF A$=">"  THEN 90 ELSE GOTO 30
90 CLOSE #1
100 END
```

Figura 6.5 A versão em BASIC de ktod

```
10 INPUT F$ : REM DTOS - DISCO PARA VIDEO
20 OPEN  F$ FOR OUTPUT AS #1
30 IF EOF(1) THEN CLOSE:END
40 INPUT #1,A$
50 PRITN A$
60 GOTO 30
```

Figura 6.6 A versão BASIC de dtos

Os nomes dos arquivos são colocados durante o tempo de processamento, em vez de na linha de comando, porque a maioria das versões em BASIC não permitem argumentos na linha-de-comando. Em vez de ser capaz de digitar a > a qualquer momento, como na versão C, o > deve ser o único caractere em uma linha no BASIC.

A FUNÇÃO fseek( )

Com a ajuda de **fseek( )**, você pode executar operações randômicas de leitura e escrita, utilizando o sistema de E/S de alto nível. **fseek( )** é utilizado para posicionar a colocação atual (byte especificado) em um arquivo. A forma geral da função é

```
fseek(fp,deslocamento,origem);
```

onde **fp** é o ponteiro de arquivo devolvido pela chamada à **fopen( )**, **deslocamento** é o número de bytes da **origem** para acertar a posição atual, e a **origem** ou é 0 para o começo do arquivo 1 para a posição atual, ou 2 para fim-do-arquivo.

Por exemplo, se você desejasse ler o 234º byte de um arquivo denominado **teste**, poderia utilizar o seguinte:

```
func1()
{
FILE *fp;

if((fp = fopen("teste","r")) == NULL)
  {
  printf("nao pode abrir arquivo\n");
  exit(1);
  }
fseek(fp,234,0);
return getc(fp); /* le um caractere */
                 /* na posicao numero 234 */
}
```

AS FUNÇÕES getw( ) e putw( )

Além de **getc( )** e **putc( )**, a maioria dos compiladores C tem ainda duas funções de E/S de alto nível: **getw( )** e **putw( )**. São utilizados para ler e escrever inteiros de dois bytes de e para um arquivo em disco.

As funções **getc( )** e **putc( )** operam apenas sobre caracteres. Como, então, um inteiro de dois bytes pode ser lido e escrito de um arquivo em disco? A resposta está em ler/escrever os dois bytes que formam o inteiro. A maneira mais fácil é utilizar as funções **getw( )** e **putw( )**. Essas funções operam exatamente da mesma maneira que **getc( )** e **putc( )**, exceto que em lugar de escrever um único caractere, escrevem um inteiro de dois bytes. Por exemplo, você poderia modificar o programa **ktod** de maneira a colocar um contador de caractere no final do arquivo. Este programa, apresentado na Figura 6.7, seria chamado **ktodc**.

Para ler arquivos escritos por **ktodc**, será necessário modificar **dtos**, de modo que ele leia o contador no fim do arquivo e procure o indicador de final de arquivo > como o sinal que o contador deve ler. O novo programa **dtosc** é apresentado na Figura 6.8.

O programa **dtosc** trabalhará com a maioria dos arquivos desde que tenha um > e uma contagem no final ou se > não está no arquivo. Você poderia alterar o caractere utilizado para sinalizar fim de texto, mas o que é importante aqui é o conceito. Não confunda o sinal >, utilizado para indicar fim de texto, com o marcador **EOF**. O **EOF** é um indicador padrão definido pelo seu compilador. Utilizamos o sinal > como um método arbitrário de informar ao **dtosc** que o texto terminou e que a contagem é o próximo item no arquivo.

Se o seu compilador C tem uma biblioteca que não contém **putw( )** ou **getw( )**, você pode utilizar a versão simples mostrada aqui:

```
putw(i,fp)
int   i;
FILE *FP;
{
char *t;

t = &i; /* atribui o endereco de i para t */
putc(t[0],fp);
putc(t[1],fp);
}

getw(fp);
FILE *fp;
{
char *t;
int   i;

t = &i; /* atribui endereco de inteiro a t */
t[0] = getc(fp);
```

```
t[1] = getc(fp);
return(i); /* note que retorna i e nao t !!! */
}
```

```
#include "stdio.h"

#define CR   13
#define LF   10
main(argc,argv) /* ktodc - tecla para disco
                           com contador */
int   argc;
char *argv[];
{
FILE *fp;
char  ch;
int   count;

count = 0;
if(argc != 2)
  {
  printf("Voce esqueceu o nome do arquivo\n");
  exit(0);
  }
if((fp = fopen(argv[1],"w")) == NULL)  {
  printf("nao pode abrir arquivo\n");
  exit(0);
  }
do
  {
  ch = getchar();
  if(ch == '\n')
    {
    putc(CR,fp); /* escreve combinacao cr/lf */
    putc(LF,fp);
    count++;
    }
  else putc(ch,fp);
  count++;
  } while(ch != '>');
putw(count,fp); /* escreve o contador no disco */
fclose(fp);
}
```

Figura 6.7 O programa ktodc, que utiliza getw( ) e putw( )

Nenhuma dessas funções, conforme indicadas, fornecem uma verificação de erro. Entretanto, se desejar pode acrescentar essa possibilidade.

O que realmente existe por detrás de **getw( )** e **putw( )** é que os inteiros realmente têm dois bytes de comprimento. Portanto, os inteiros podem ser divididos em dois bytes, como em **putw( )**, designando seu endereço para um ponteiro de caractere e escrevendo um byte por vez, utilizando **putc( )**. O inverso também é verdade: um inteiro pode ser reconstruído um byte por vez, designando seu endereço ao ponteiro de caractere e executando duas chamadas sucessivas para **getc( )**.

```
#include "stdio.h"
main(argc,argv) /* dtosc - disco para video
                           com contador */
int   argc;
char *argv[];
{
FILE *fp;
int   ch;
int   count;

if(argc != 2)
  {
  printf("Voce esqueceu o nome do arquivo\n");
  exit(0);
  }
if((fp = fopen(argv[1],"r")) == NULL)
  {
  printf("nao pode abrir arquivo\n");
  exit(0);
  }
ch = getc(fp); /* le um caractere */
while(ch != EOF && ch != '>') /* procura sinal '>' */
  {
  putchar(ch); /* mostra no video */
  ch = getc(fp);
  }
if(ch == '>')
  {
  count = getw(fp);
  printf("contador de caracteres e' : %d\n",count);
  }
fclose(fp);
}
```

Figura 6.8 O programa dtosc, uma adaptação do programa dtos

LENDO E ESCREVENDO OUTROS TIPOS DE DADOS

A maioria das bibliotecas C não apresenta funções inerentes de leitura e escrita para outros tipos de dados, que não sejam caracteres e inteiros. Entretanto, você pode escrever esses outros tipos de dados, construindo funções que operam de maneira similar ao **getw( )** e **putw( )**. Por exemplo, caso um dado do tipo **float** tivesse oito bytes de comprimento, esta função, **putfloat( )**, poderia ser utilizada para escrever um número em ponto flutuante a um arquivo em disco:

```
putfloat(num,fp);
float num;
FILE *fp;
{
char *t;
int   count;

t = &num;
for(count = 0;count < 8;++count)
putc(t[count],fp);
}
```

De fato, você poderia construir funções específicas **put_x( )** e **get_x( )**, onde **x** é uma estrutura de dados arbitrária ou uma unidade de dados arbitrária. Não precisa ser limitado apenas a tipos de dados predeterminados.

ARQUIVOS stdin, stdout e stderr

Sempre que um programa C inicia sua execução, três arquivos são abertos automaticamente: *entrada padrão* ou **stdin**; *saída padrão* ou **stdout**; e erro padrão ou **stderr**. Normalmente estes se referem ao console. São, no entanto, ponteiros de arquivo e podem ser utilizados, pelo sistema de E/S de alto nível, para executar operações de E/S no console. Em resumo, permitem que o console seja tratado como se fosse um arquivo. Por exemplo, **putchar( )** poderia ser definido como

```
putchar(c)
char c;
{
putc(c,stdout);
}
```

Geralmente **stdin** é utilizado para ler do console, e **stdout** e **stderr** são utilizados para escrever no console. Nos sistemas operacionais que permitem o redirecionamento de E/S, apenas **stdin** e **stdout** serão redirecionados, com **stderr** continuando a escrever no console. (Isso varia com cada sistema operacional e implementação C. Portanto, veja seu manual de usuário.)

Você pode utilizar **stdin**, **stdout** e **stderr** como ponteiros de arquivo em qualquer função que utiliza uma variável do tipo ***FILE**.

Lembre-se de que **stdin**, **stdout** e **stderr** não são variáveis, mas constantes, e como tais não podem ser atribuídas ou alteradas. Da mesma maneira que esses ponteiros de arquivo são gerados automaticamente no início do programa, são fechados automaticamente no fim do mesmo; você não deve tentar utilizar **fclose( )** para fechá-los.

AS FUNÇÕES fprintf( ) e fscanf( )

Em adição às funções de E/S básicas de **getc( )** e **putc( )**, as bibliotecas da maioria das implementações C têm **fprintf( )** e **fscanf( )**, que podem ser utilizadas para escrever diversos formatos de dados em um arquivo aberto por **fopen( )**. A forma geral de **fprintf( )** é

```
fprintf(fp,série de controle,lista de argumentos);
```

e a forma geral de **fscanf( )** é

```
fscanf(fp,série de controle,lista de argumentos);
```

onde **fpe** é um ponteiro de arquivo devolvido com uma chamada para **fopen( )**. Exceto pelo direcionamento de suas saídas para o arquivo definido por **fp**, **fprintf( )** e **fscanf( )**, operam exatamente como **printf( )** e **scanf( )**, respectivamente.

Para ilustrar a utilidade dessas funções, o programa da Figura 6.9 vai ler informação do teclado, escrevê-la em um arquivo em disco, para em seguida ler a informação e recolocá-la no vídeo.

Uma nota de aviso: apesar de **fprintf( )** e **fscanf( )** serem com freqüência a maneira mais fácil de escrever e ler arquivos em disco, não são os mais eficientes, porque os dados ASCII são escritos da maneira em que aparecem em seu vídeo, em vez de em binário. Se você estiver preocupado com velocidade ou tamanho de arquivo, provavelmente deve escrever rotinas de arquivos próprias semelhantes ao **putw( )** e **getw( )**.

E/S DE BAIXO NÍVEL: ROTINAS DE ARQUIVO TIPO UNIX

Pelo fato de C ter sido desenvolvido originalmente no sistema operacional UNIX e por certas aplicações requererem a habilidade de dirigir as operações de disco no nível do sistema operacional, foi criado um segundo subsistema de E/S de arquivo em disco. Utiliza funções diferentes daquelas aprendidas nas seções anteriores. Estas são as funções de baixo nível, do tipo UNIX para E/S de disco:

read( )	**write( )**	**open( )**
close( )	**creat( )**	**unlink( )**
lseek( )		

```
/*
** Exemplo de fscanf - fprintf
*/
#include <stdio.h>

main()
{
FILE *fp;
char  s[80];
int   t;

if((fp = fopen("teste","w")) == NULL)
  {
  printf("nao pode abrir arquivo.\n");
  exit(0);
  }
fscanf(stdin,"%s%d",s,&t);    /* leitura do teclado */
fprintf(fp,"%s %d",s,t);      /* escrita no arquivo */
fclose(fp);
if((fp = fopen("teste","r")) == NULL)
  {
  printf("nao pode abrir arquivo.\n");
  exit(0);
  }
fscanf(fp,"%s%d",s,&t);       /* leitura do arquivo */
fprintf(stdout,"%s %d",s,t); /* escrita no video */
}
```

Figura 6.9 Um programa que utiliza **fprintf( )** e **fscanf( )**

A razão pela qual o subsistema de E/S de disco construído com essas funções denomina-se "baixo nível" é que, como programador, você terá de fornecer e manter *todas* as áreas intermediárias de disco. Ao contrário das funções **getc( )** e **putc( )**, que escrevem ou lêem caracteres de ou para uma área de dados, que foi automaticamente escrita em ou lida de um arquivo de disco, as funções **read( )** e **write( )** lêem ou escrevem uma área intermediária completa de informação em cada chamada. Você terá de colocar a informação que será utilizada nesta área, e deve saber quando a área está cheia. Você define os comprimentos da área e do registro para seu arquivo.

Os iniciantes em C normalmente acham que o sistema de E/S de alto nível é mais fácil de trabalhar e menos sujeito a erros. Porém, à medida que você progredir, o sistema de E/S de baixo nível poderá oferecer maior flexibilidade e velocidade, para determinadas aplicações.

AS FUNÇÕES open( ), close( ) E creat( )

Ao contrário do sistema de E/S de alto nível, os sistemas de baixo nível não utilizam ponteiros de arquivo do tipo **FILE**, mas descritores de arquivo do tipo **int**. A forma geral de uma chamada para **open( )** é

```
int fd;
fd = open(nomearquivo,modo);
```

onde nomearquivo é qualquer nome de arquivo válido, e modo é um dos seguintes inteiros:

Modo	Efeito
0	ler
1	escrever
2	ler/escrever

A função **open( )** devolve −1 quando o arquivo não pode ser aberto. Você verá, normalmente, a chamada para **open( )** como:

```
int fd;

if((fp = open(filename,mode)) == -1)
  {
  printf("nao pode abrir arquivo\n");
  exit(0);
  }
```

Dependendo da implementação de seu compilador C, você será capaz de utilizar **open( )** para criar um arquivo que atualmente não exista. O manual do usuário deverá ser consultado. A forma geral de **close( )** é

```
int fd;
close(fd);
```

A função **close( )** devolve −1 se não conseguir fechar o arquivo. Isto poderia ocorrer, por exemplo, se o disquete fosse removido do acionador.

A função **close( )** libera o descritor do arquivo, de modo a poder ser utilizado com outro arquivo. Existe sempre um limite para o número de arquivos que podem estar abertos simultaneamente, de modo que sempre que um arquivo não é mais necessário, você deve utilizar **close( )**. Apesar da programação normal fechar todos os arquivos automaticamente, isto não é considerado uma boa prática de programação.

Caso seu compilador não permita a criação de um arquivo novo utilizando **open( )**, ou caso você queira garantir a portabilidade, deve utilizar **creat( )**. A função **creat( )**, basicamente, abre um novo arquivo para operações de escrita. A forma geral de **creat( )** é

```
int fd;
fd = creat(nomearquivo,pmodo);
```

onde nomearquivo é qualquer nome de arquivo válido. O pmodo não é utilizado com a maioria dos compiladores C para microcomputadores, apesar de continuar fazendo parte da chamada **creat( )**. (Consulte seu manual de usuário C para detalhes específicos.) Basicamente, pmodo tem a ver com os modos de proteção no UNIX. Um **creat( )** devolve −1 em caso de falha.

AS FUNÇÕES write( ) E read( )

Uma vez que um arquivo tenha sido aberto para escrita e você tenha declarado um vetor de caracteres para atuar como área intermediária, esse arquivo pode ser acessado por **write( )**. Em cada operação de escrita, a área será escrita no disco. Você deve ainda especificar o tamanho da área auxiliar: este será o número de bytes realmente escritos no arquivo em disco. Normalmente, deveria ser do mesmo tamanho de sua área intermediária. Em alguns sistemas deverá ser um múltiplo par de 128. Em outros, poderá ser qualquer número. Assegure-se de olhar seu manual de usuário C.

A forma geral da função **write( )** é

```
#define BUF_SIZE        128
int fd;
char buf[BUF_SIZE];
write(fd,buf,BUF_SIZE);
```

Cada vez que se executa uma chamada para **write( )**, os caracteres designados por **BUF_SIZE** são escritos no arquivo em disco, especificados por **fd** do vetor de caracteres **buf**. O vetor de caracteres **buf** não precisa terminar com um nulo, pois não se trata de uma série.

Você deve estar pensando porque é que todo o conteúdo da área auxiliar não é escrito automaticamente no disco. A resposta é: a área poderá não terminar com um nulo. Não existe maneira pela qual a função **write( )** possa saber o comprimento da área sem informação explícita. Você pode ainda desejar escrever menos que uma área completamente cheia.

A função **write( )** informa o tamanho da área após uma operação de escrita bem-sucedida. Em caso de falha, a maioria das implementações devolve −1; mas, verifique o seu manual de usuário.

A função **read( )** é o complemento de **write( )**. A forma geral de **read( )** é

```
read(fd,buf,BUF_SIZE);
```

onde **fd, buf** e **BUF_SIZE** são os mesmos que em **write( )**. Quando **read( )** é bem-sucedido, informa o número de caracteres realmente lidos. Devolve 0 com a terminação física do arquivo, e −1 em caso de erro.

O programa apresentado na Figura 6.10 mostra alguns aspectos de E/S de baixo nível. Vai ler linhas de texto do teclado e escrevê-las no arquivo em disco. Após terem sido escritas, o programa as relê.

```
#include "stdio.h"

#define BUF_SIZE        128

/*
** Leitura e escrita usando E/S de nivel baixo
*/
main()
{
char buf[BUF_SIZE];
int  fd1,fd2,t;

if((fd1 = open("test",1)) == -1)
      /* abertura para escrita */
  {
  printf("nao posso abrir arquivo.\n");
  exit(0);
  }
input(buf,fd1);
                        /* agora fecha e le a escrita */
close(fd1);
if((fd2 = open("test",0)) == -1)
      /* abertura para leitura */
  {
  printf("nao posso abrir arquivo.\n");
  exit(0);
  }
display(buf,fd2);
close(fd2);
}

input(buf,fd1)
char *buf;
int   fd1;
{
do
  {
```

Figura 6.10 Um programa que lê linhas de texto do teclado e escreve-as em disco

```
    gets(buf); /* le caracteres do teclado */
    if(write(fd1,buf,BUF_SIZE))
      {
      printf("Erro na leitura.\n");
      exit(0);
      }
    printf(buf);
    } while (!strcmp(buf,"quit"));
}
```

Figura 6.10 *(continuação)*

A FUNÇÃO unlink( )

Caso você deseje remover um arquivo do diretório, vai utilizar **unlink( )**. Apesar de **unlink( )** ser considerado parte do sistema de E/S de baixo nível, removerá qualquer arquivo do diretório. A forma geral de **unlink( )** é

```
unlink(nomearquivo);
```

onde nomearquivo é um ponteiro de caracteres para qualquer nome de arquivo válido. A função **unlink( )** devolve um erro (normalmente −1) caso não consiga cancelar o arquivo. Isso ocorreria se o arquivo não estivesse presente no disquete, ou que o disquete tivesse proteção contra escrita.

Tabela 6.5 O Efeito da Origem sobre **lseek**

Origem	Efeito de chamada para lseek( )
0	Conta o deslocamento do início do arquivo
1	Conta o deslocamento da posição atual
2	Conta o deslocamento do fim do arquivo.

ARQUIVOS DE ACESSO RANDÔMICO E lseek( )

C sustenta E/S com arquivos de acesso randômico no sistema de E/S de baixo nível, utilizando chamadas para **lseek( )**. A forma geral de **lseek( )** é

```
int fd, origem;
long deslocamento;
lseek(fd, deslocamento, origem);
```

onde **fd** é um descritor de arquivo devolvido por uma chamada à **creat( )** ou à **open( )**.

```
#include "stdio.h"

#define BUF_SIZE 128

main(argc,argv) /* leitura e escrita
                   usando E/S baixo nivel */
int   argc;
char *argv[];
{
char buf[BUF_SIZE + 1],s[10];
int  fd,sector;

buf[BUF_SIZE + 1] = '\0'; /* terminador nulo */
if((fd = open(argv[1],0)) == -1)
      /* abertura para leitura */
  {
  printf("nao pode abrir arquivo\n");
  exit(0);
  }
do
  {
  gets(s);
  sector = atoi(s); /* numero do setor a ler */
  if(lseek(fd,sector * BUF_SIZE,0) == -1)
    printf("Erro de seek\n");
  if(read(fd,buf,BUF_SIZE) == 0)
    printf("Setor fora dos limites\n");
  else printf(buf);
  } while(sector > 0);
close(fd);
}
```

Figura 6.11 O programa **read_file**, que utiliza **lseek( )**

A maneira com que **lseek( )** opera depende do valor da origem e do deslocamento. A origem pode ser 0, 1 ou 2. A Tabela 6.5 explica como o deslocamento é interpretado para cada valor de origem.

Um exemplo simples da utilização de **lseek( )** é o programa **read file** apresentado na Figura 6.11. Para processá-lo, deverá ser especificado um arquivo na linha de comando. Em seguida, podemos especificar a área auxiliar que desejamos ler. A entrada com um número negativo permite que saiamos. Apesar de não ser necessário, você poderá desejar alterar o tamanho da área auxiliar para que se iguale ao tamanho do setor de seu sistema.

UTILIZANDO ARQUIVOS EM DISCO

Nesta seção, rotinas de escrita e leitura em arquivos de disco serão acrescentadas ao programa de banco de dados de cliente da Figura 6.2. Isso permitirá que você salve o banco de dados em um arquivo em disco, e o leia posteriormente.

A primeira rotina que deve ser escrita é denominada **wr_data( )**, e escreve todo o arquivo de banco de dados em um arquivo denominado **client** e utiliza um sistema de E/S de alto nível:

```
wr_data()
{
register int   t;
         FILE *fp;

if((fp = fopen("cliente","w")) == 0)
  {
  printf("nao pode abrir arquivo cliente.\n");
  exit(0);
  }
for(t = 0;t < 1000;t++) putc(name[t],fp);
   /* salva os nomes dos clientes */
for(t = 0;t < 50;++t) putw(balance[t],fp);
   /* salva saldos */
fclose(fp);
}
```

A segunda rotina que você necessita é **rd_data( )** que vai ler todo o banco de dados do arquivo **client** e que também utiliza o sistema de E/S de alto nível:

```
rd_data()
{
register int   t;
         FILE *fp;

if((fp = fopen("cliente","r")) == 0)
  {
  printf("nao pode abrir arquivo cliente.\n");
  exit(0);
  }
for(t = 0;t < 1000;t++) name[t] = getc(fp);
   /* restaura nomes de clientes */
for(t = 0;t < 50;++t) balance[t] = getw(fp);
   /* restaura saldos */
fclose(fp);
}
```

Finalmente, a Figura 6.12 mostra a função **main( )** modificada para a inclusão de duas novas opções.

```
#include "stdio.h"

char name[1000];  /* 50 nomes de cliente,
                     20 caracteres cada */
int  balance[50];

main()  /* programa de base de dados
           de clientes simples */
{
char ch;
int  client;

init();  /* inicializa a base de dados com 0 */
do
  {                          /* malha principal */
  printf("Suas opcoes:\n");
  printf("\n\n   1.Inserir um novo cliente.\n");
  printf("   2.Procurar um cliente.\n");
  printf("   3.Atualizar um cliente.\n");
  printf("   4.Retirar um cliente.\n");
  printf("   5.Salvar a base de dados.\n");
  printf("   6.Restaurar a base de dados.\n");
  printf("   7.Sair.\n");
  do
    {
    printf("Selecione opcao: ");
    ch = getchar();
    fflush(stdin);
    } while(ch < '1' || ch > '7');
  printf("\n");
  switch(ch)
    {
    case '1' : enter();
               break;
    case '2' : client = find();
               if(client != -1) display(client);
               break;
    case '3' : client = find();
               if(client != -1) update(client);
               break;
```

Figura 6.12 A nova versão do programa de banco de dados do cliente, dado na Figura 6.2

```
        case '4' : client = find();
                   if(client != -1) delete(client);
                   break;
        case '5' : wr_data();
                   break;
        case '6' : rd_data();
                   break;
        case '7' : break;
        }
    } while(ch != '7');
}
```

Figura 6.12 *(continuação)*

Em resumo, a biblioteca padrão C em verdade contém dois sistemas separados de E/S de arquivos: o de alto nível e o de baixo nível. Apesar dos dois sistemas poderem ser utilizados para qualquer tarefa, cada um tem seus próprios méritos. O programador C capaz vai escolher o sistema certo para uma determinada tarefa. Em geral, as rotinas de baixo nível são mais rápidas mas requerem um trabalho maior de sua parte. Primeiro você deve consultar seu manual de usuário C quanto aos detalhes, antes de utilizar qualquer das funções de arquivo em disco.

EXERCÍCIOS

1. Escreva um programa simples que emule uma calculadora de quatro operações. Para cada cálculo, insira dois números e um operador. Utilizando uma declaração **switch**, baseada no operador, imprima a saída da operação. Por exemplo, caso você tivesse digitado 100 100 +, imprimiria 200 na tela.

2. Escreva uma chamada **printf( )** que apresente ao menos cinco caracteres, mas não mais que 10 caracteres, para cada um dos seguintes tipos de dados: série, inteiro e ponto flutuante.

3. Escreva uma chamada **scanf( )** que insira uma série e dois números inteiros.

4. Escreva um programa que utilize E/S de alto nível, que copie um arquivo em disco para outro. Utilize argumentos de linhas-de-comando para especificar os arquivos.

5. Modifique o exemplo do banco de dados de cliente, de modo que armazene o primeiro e o último nome.

6. Modifique as rotinas de arquivo do banco de dados de cliente, para que seja feito o armazenamento e recuperação do primeiro e último nome acrescentados no exercício anterior.

RESPOSTAS

1.
```
#include "stdio.h"

main()  /* programa calculadora */
{
float num1, num2;
char  op;

do
  {
  printf("entre dois numeros e operador:");
  scanf("%f%f %c",&num1,&num2,&op);
  switch(op)
    {
    case '+' : printf("%f\n",num1 + num2);
               break;
    case '-' : printf("%f\n",num1 - num2);
               break;
    case '/' : printf("%f\n",num1 / num2);
               break;
    case '*' : printf("%f\n",num1 * num2);
               break;
    default  : printf("Operador desconhecido.\n");
    }
  } while(op != 'q');
}
```

2.
```
printf("%5.10d %5.10s %5.10f\n",dec,string,flt);
```

3.
```
scanf("%s%d%d",s,&d1,&d2);
```

4.
```
#include "stdio.h"

main(argc,argv) /* programa de copia */
int   argc;
char *argv[];
{
int   ch;
FILE *fp1,*fp2;

if((fp1 = fopen(argv[1],"r")) == 0)
  {
  printf("nao pode abrir arquivo %s\n",argv[1]);
  exit(0);
```

```
  }
if((fp2 = fopen(argv[2],"w")) == 0)
  {
  printf("nao pode abrir arquivo %s\n",argv[2]);
  exit(0);
  }
while((ch = getc(fp1)) != EOF) putc(ch,fp2);
fclose(fp1);
fclose(fp2);
}
```

5.
```
#include "stdio.h"

char name[1000];  /* 50 nomes de cliente,
                     20 caracteres cada */
char fname[1000]; /* guarda os sobrenomes */
int  balance[50];

main()  /* programa de base de dados
           de clientes simples */
{
char ch;
int  client;

init();  /* inicializa a base de dados com 0 */
do
  {                          /* malha principal */
  printf("Suas opcoes:\n");
  printf("\n\n   1.Inserir um novo cliente.\n");
  printf("   2.Procurar um cliente.\n");
  printf("   3.Atualizar um cliente.\n");
  printf("   4.Retirar um cliente.\n");
  printf("   5.Salvar a base de dados.\n");
  printf("   6.Restaurar a base de dados.\n");
  printf("   7.Sair.\n");
  do
    {
    printf("Selecione opcao: ");
    ch = getchar();
    fflush(stdin);
    } while(ch < '1' || ch > '7');
  printf("\n");
  switch(ch)
    {
    case '1' : enter();
               break;
```

```
    case '2' : client = find();
               if(client != -1) display(client);
               break;
    case '3' : client = find();
               if(client != -1) update(client);
               break;
    case '4' : client = find();
               if(client != -1) delete(client);
               break;
    case '5' : wr_data();
               break;
    case '6' : rd_data();
               break;
    case '7' : break;
    }
  } while(ch != '7');
}

init()
{
register int t;

for(t = 0;t < 1000;++t) name[t] = '\0';
}

enter()
{
char s[20];
int  slot;

for(slot = 0;slot < 50;slot++)
   if(!name[slot * 20]) break;
      /* encontrou aberto */
if(slot == 50)
  {
  printf("Lista de clientes cheia.\n");
  return 0;
  }
printf("Entre sobrenome, nome e saldo:\n");
scanf("%19s%19s%d",&fname[slot * 20],
            &name[slot * 20],&balance[slot]);
fflush(stdin);
}

find()
{
```

```
char s[20];
int  slot;

printf("Entre o nome do cliente:\n");
scanf("%19s",s);
fflush(stdin);
for(slot = 0;slot < 50;slot++)
   if(!strcmp(s,&name[slot * 20])) break;
     /* encontrou, abandone a malha */
if(slot == 50)
  {
  printf("Cliente nao esta na lista.\n");
  return(-1);
  }
return slot;
}

display(slot)
int slot;
{
printf("\nO saldo de %s %s e' %d\n",&fname[slot * 20],
        &name[slot * 20],balance[slot]);
}

update(slot)
int slot;
{
printf("Entre novo sobrenome, nome e saldo:");
scanf("%19s%19s%d",&fname[slot * 20],
            &name[20 * slot],&balance[slot]);
}

delete(slot)
int slot;
{
printf("Entrada retirada.\n");   name[20 * slot] = '\0';
balance[slot] = 0;
}

wr_data()
{
register int   t;
         FILE *fp;

if((fp = fopen("cliente","w")) == 0)
  {
```

```
  printf("nao pode abrir arquivo cliente.\n");
  exit(0);
  }
for(t = 0;t < 1000;t++) putc(name[t],fp);
   /* salva os nomes dos clientes */
for(t = 0;t < 1000;t++) putc(fname[t],fp);
   /* salva os sobrenomes */
for(t = 0;t < 50;++t) putw(balance[t],fp);
   /* salva saldos */
fclose(fp);
}

rd_data()
   {
   register int   t;
            FILE *fp;

   if((fp = fopen("cliente","r")) == 0)
     {
     printf("nao pode abrir arquivo cliente.\n");
     exit(0);
     }
   for(t = 0;t < 1000;t++) name[t] = getc(fp);
       /* restaura nomes de clientes */
   for(t = 0;t < 1000;t++) fname[t] = getc(fp);
       /* restaura sobrenomes */
   for(t = 0;t < 50;++t) balance[t] = getw(fp);
       /* restaura saldos */
   fclose(fp);
   }
```

CAPÍTULO **7**

PONTEIROS

Em capítulos anteriores você já foi apresentado aos ponteiros. Neste capítulo você aprenderá com maiores detalhes como utilizá-los, bem como alguns "macetes" que permitirão que seus programas sejam processados com maior rapidez. A utilização de ponteiros tem grande importância para que a programação C seja bem-sucedida, de modo que este capítulo deve ser lido com muito cuidado.

Os ponteiros em C não se igualam a nenhuma estrutura do BASIC. É possível fazer uma simulação, utilizando **PEEK** e **POKE**, mas exemplos desta natureza provavelmente não vão auxiliá-lo na compreensão dos ponteiros. Portanto, nenhuma comparação ao BASIC será feita neste capítulo.

PONTEIROS COMO ENDEREÇOS

Um *ponteiro* é uma variável que contém um endereço. Lembre-se, este endereço é a localização de outra variável na memória. O valor de um ponteiro "aponta" para uma variável, que pode ser acessada indiretamente com os operadores especiais de ponteiros * e &. O operador * acessa o conteúdo de uma variável cujo endereço é o valor de um ponteiro. O * pode ser memorizado como "no endereço". O operador & devolve o endereço de uma variável e pode ser memorizado como "o endereço de".

Por exemplo, se **xyz** e **k** são variáveis inteiras e **h** um ponteiro para inteiro, então

```
h = &xyz;
k = *h;
```

atribui o valor de **xyz** para **k**.

Os ponteiros devem ser declarados. Para declarar um ponteiro para um inteiro **x**, você utilizaria

```
int *x;
```

Para um ponteiro **float** de **x** utilizaríamos

```
float *x;
```

Você deve se certificar de que as variáveis ponteiros sempre apontam para o tipo de dado correto. Quando você declara um ponteiro para que seja do tipo **int**, o compilador assume que qualquer endereço contido pelo ponteiro aponta para uma variável inteira. Uma vez que C permite que se atribua qualquer endereço para uma variável ponteiro, o seguinte trecho de codificação compila sem erro, mas não fornece o resultado desejado:

```
float  x,y;
int    *p;

p = &x;
y = *p;
```

Este código *não irá* atribuir o valor de **x** para **y**. Uma vez que **p** foi declarado como sendo um ponteiro inteiro, apenas dois bytes de informação serão transferidos para **y**, e não os oito bytes que normalmente formam um número em ponto flutuante.

Como foi visto no Capítulo 6, essa habilidade em estabelecer qualquer endereço para um ponteiro de qualquer tipo é uma vantagem quando lemos e escrevemos arquivos. Quando fizer isto, você deve ter certeza absoluta do que está fazendo. Caso seu programa lhe forneça resultados inesperados, você deve verificar as atribuições de seus ponteiros.

ATRIBUIÇÃO DE PONTEIROS

Como com qualquer variável, um ponteiro poderá ser utilizado do lado direito de uma declaração de atribuição, para estabelecer o seu valor para outro ponteiro ou para um não-ponteiro, conforme indicado aqui:

```
int x;
unsigned y;
int *p1,*p2;

p1 = &x;
p2 = p1; /* atribui a p2 o endereco de x */
y = p2;
printf(" %u",y); /* imprime o valor decimal
                    do endereco de x
                    nao e' o valor de x! */
```

Você poderia ter impresso diretamente o valor decimal de **p2**, utilizando **p2** na chamada **printf( )**. Este trecho de codificação não tem a pretensão de indicar que inteiros sem sinal sejam ponteiros ou que possam ser utilizados para manter valores de ponteiros em geral. Ao contrário, mostra que o valor de um ponteiro é apenas um valor como outro qualquer e pode ser atribuído para qualquer outro tipo de variável, dependendo das regras de conversão. Inteiros e ponteiros de qualquer tipo não são intercambiáveis.

EXPRESSÕES DE PONTEIROS

Os ponteiros podem ser utilizados na maioria das expressões C válidas. Devem ser aplicadas determinadas regras, e você poderá ser obrigado a aplicar parênteses adicionais para obter o resultado desejado.

ARITMÉTICA DE PONTEIROS

Existem apenas dois operadores aritméticos que podem ser utilizados nos ponteiros, + e −. Para entender o que acontece na aritmética de ponteiros, seja **p1** um ponteiro para um inteiro com um valor atual de 2000. Após a expressão

```
p1++;
```

o conteúdo de **p1** será 2002 e não 2001. Para cada incremento de **p1** ele apontará para o *inteiro seguinte*, que na maioria dos computadores tem dois bytes de comprimento. O mesmo é válido para decréscimos. Por exemplo,

```
p1--;
```

fará com que **p1** tenha o valor 1998, presumindo que o valor anterior era 2000.

Lembre-se de que cada vez que um ponteiro é incrementado, apontará para o próximo local de memória que contém um elemento de seu tipo. Cada vez que é decrementado, apontará para o local anterior com elemento de seu tipo.

Para ponteiros de caracteres isso freqüentemente se parece com a aritmética "normal". Entretanto, todos os outros ponteiros aumentam ou diminuem de acordo com o comprimento do tipo de dado que apontam. Para os inteiros, este comprimento normalmente é dois bytes; para flutuantes normalmente é oito bytes. Apesar de ter sido utilizado endereço de memória na explicação sobre as diferenças dos diversos ponteiros, lembre-se de que você não precisa conhecer os valores dos ponteiros porque estes não estão necessariamente na condição de serem utilizados.

Porém, você não está limitado a incrementar e decrementar ponteiros. Você pode adicionar ou subtrair inteiros dos ponteiros. A expressão

```
p1 = p1 + 9;
```

fará com que **p1** aponte para o nono elemento de seu tipo, além daquele que estiver apontando no momento.

Não podem ser executadas outras operações aritméticas sobre ponteiros além da subtração e adição de um ponteiro e um inteiro: você não pode multiplicar ou dividir ponteiros; não pode somar ou subtrair dois ponteiros; não pode aplicar os operadores de deslocamento de bits e mascaramento; não pode somar ou subtrair tipo **float** ou **double** e ponteiros.

COMPARAÇÃO DE PONTEIROS

Dois ponteiros que se referem a tipos de variáveis separadas, não apresentam relacionamento entre si. Por exemplo, caso **p1** e **p2** sejam ponteiros que apontem para duas variáveis separadas e não relacionadas, qualquer comparação entre **p1** e **p2** não tem sentido, porque sem dúvida **p1** e **p2** serão diferentes. Entretanto, se **p1** e **p2** apontam para variáveis relacionadas entre si — como os elementos do mesmo vetor — **p1** e **p2** poderão ser comparados. Assim, ponteiros que têm esse tipo de relacionamento poderão ser utilizados em operações lógicas e relacionais.

Por exemplo, imagine que você esteja construindo rotinas de pilha para conter valores inteiros. Uma *pilha* é uma lista que utiliza um acesso do tipo "primeiro-que-entra, último-que-sai". (Freqüentemente é comparado com uma pilha de pratos sobre uma mesa, porque o primeiro colocado é o último que será utilizado.) As pilhas são freqüentemente utilizadas em compiladores, interpretadores, planilhas e outros tipos de software de sistema.

Para criar uma pilha, você precisa de duas rotinas, **push( )** e **pop( )**, e alguma área de memória livre para sua pilha. Para reservar a área de memória você precisa inicialmente utilizar a função padrão **malloc( )**. Um **malloc( )** devolve um ponteiro de caractere. Caso **p1** represente o ponteiro da pilha, então o código que segue vai inicializá-lo com 50 bytes de memória livre. Para este exemplo, pressuponha que sua máquina utilize dois bytes para inteiros; esse código não será aplicável em qualquer computador:

```
int  *p1,*tos;
char *malloc(); /* declara malloc() retorna um ponteiro
                   para caracteres */

p1 = malloc(50);
tos = p1; /* tos guarda o topo da pilha */
```

A variável **tos** conterá o endereço de memória do topo da pilha e será utilizada para impedir "underflow" da pilha. Após a inicialização da pilha, **push( )** e **pop( )** podem ser utilizados para uma pilha de inteiros, como é indicado aqui:

```
push(i)
int i;
{
p1++;
```

```
if(p1 == (tos + 50))
  {
  printf("Estouro da pilha\n");
  exit();
  }
*p1 = i;
}

pop()
{
if((p1 - 1) == tos)
  {
  printf("Estouro inferior da pilha\n");
  exit();
  }
p1--;
return *(p1 + 1);
}
```

As funções **push( )** e **pop( )** executam um teste relacional no ponteiro **p1** para detectar erros de limite. Em **push( ), p1** é testado contra o fim da pilha pela adição de 25 ao **tos**. Lembre-se de que neste exemplo, presume-se que os inteiros têm dois bytes de comprimento. Em **pop( ), p1** é comparado com **tos** para haver a certeza de que não tenha ocorrido um extravasamento "por baixo".

Em **pop( )**, sem os parênteses, a declaração **return** ficaria

```
return *p1 + 1;
```

que devolveria o valor da posição **p1** mais um, e não o valor da posição **p1 + 1**. Você deve ter cuidado em utilizar os parênteses para assegurar que foi feita a avaliação correta, ao utilizar ponteiros.

Esta implementação armazena apenas 24 (e não 25) valores, deixando sem utilização os dois primeiros bytes da memória. (Mais tarde vai reescrevê-lo para que armazene todos os 25.)

PONTEIROS E VETORES

Como você já deve ter adivinhado, existe um relacionamento muito íntimo entre ponteiros e vetores. Por exemplo, em

```
char str[80];
char *p1;

p1 = str;
```

p1 foi definido como o endereço do primeiro elemento do vetor **str**. Se você quisesse acessar o quinto elemento de **str**, poderia escrever

```
str[4]
```

ou

```
*(p1 + 4)
```

As duas declarações fornecem o quinto elemento. Lembre-se de que uma vez que os vetores iniciam em zero, utilizamos um quatro para indexar **str**. Você poderia também somar quatro ao ponteiro **p1**, para a obtenção do quinto elemento, porque normalmente **p1** aponta para o primeiro elemento de **str**.

Em resumo, C permite dois métodos de acesso aos elementos de um vetor. Isso é importante porque a aritmética dos ponteiros poderá ser *mais rápida* que a indexação do vetor. Uma vez que a velocidade tem importância em programação, a utilização de ponteiros para acessar elementos de um vetor é muito comum nos programas C.

Por exemplo, estas são duas versões de **puts( )**, – uma com indexação de vetor,

```
puts(s) /* com vetores */
char *s;
{
register int t;

for(t = 0;s[t];++t) putchar(s[t]);
}
```

e uma com ponteiros,

```
puts(s) /* com ponteiros */
char *s;
{
while(*s) putchar(*s++);
}
```

A maioria dos programadores profissionais em C acharia a segunda versão mais fácil de ser lida e compreendida, de modo que esta é a maneira em que as rotinas desse tipo normalmente são escritas em C.

Não pense que a indexação de vetor está errada, ela tem seu lugar. Lembre-se apenas de que caso o vetor deva ser acessado estritamente na ordem ascendente ou descendente, os ponteiros são mais rápidos e mais fáceis de serem utilizados. Caso o vetor deva ser acessado randomicamente, a indexação do vetor provavelmente será melhor, por ser mais fácil de ser lida e codificada, apesar de que os ponteiros continuem mais rápidos.

PONTEIROS PARA VETOR DE CARACTERES

Antes deste capítulo, todas as operações com séries eram feitas pela indexação do vetor de caracteres. Entretanto, a maioria das operações com séries em C são normalmente feitas com ponteiros de vetor e aritmética de ponteiros, porque os ponteiros são mais rápidos e mais fáceis de serem utilizados.

O que segue são duas maneiras de escrever a função **strcmp( )**, encontrada na biblioteca de seu compilador C:

```
strcmp(s1,s2) /* com vetores */
char *s1,*s2;
{
register int t;

for(t = 0;s1[t];++t) if(s1[t] != s2[t]) return s1[t];
if(s2[t]) return (s2[t]);
return '\0';
}

strcmp(s1,s2) /* com ponteiros */
char *s1,*s2;
{
while(*s1) if(*s1++ != *s2++) return *(s1--);
if(*s2) return (*s2);
return '\0';
}
```

Caso você tenha esquecido, todas as séries em C são terminadas por um nulo, que é um valor falso. Portanto, uma declaração como

```
while(*s1)
```

é considerada válida até que se chegue ao fim da série.

A maioria das funçõs série se assemelham ao **strcmp( )** com ponteiros onde existe controle de malha. É mais rápido, mais eficiente e mais fácil de ser entendido do que os vetores.

Na função **strcmp( )**, tanto **s1** como **s2** eram variáveis locais. Poderiam ser alteradas sem efeito secundário nas variáveis de chamadas. Porém, tome cuidado. Você consegue encontrar o erro no programa da Figura 7.1?

Na Figura 7.1, **p1** recebe o endereço de **s** uma vez. Essa designação foi feita fora da malha. Durante a primeira passagem pela malha, **p1** aponta para o primeiro caractere em **s**. Entretanto, durante a segunda passagem continua do ponto em que parou porque não foi reinicializado para o início do vetor s. Esse novo caractere poderá ser parte da próxima série, outra variável ou outro trecho de programa. Eventualmente, teremos um aborto de programa.

```
main()  /* Este programa tem um erro */
{
char  s[80];
char *p1;

p1 = s;
do
  {
  gets(s);  /* le uma serie de caracteres */
  while(*p1) print("%d\n",*p1++);
/* imprime o equivalente decimal de cada caractere */
  } while(!strcmp(s,"done"));
}
```

Figura 7.1 Um programa com erro

A maneira correta de escrever este programa é dado na Figura 7.2. Para cada iteração da malha, **p1** é colocado no início da série. Lembre-se, você precisa saber para onde está apontando o ponteiro em cada instante.

```
main()  /* Este programa esta correto */
{
char  s[80];
char *p1;
do
  {
  p1 = s;
  gets(s);  /* le uma serie de caracteres */
  while(*p1) print("%d\n",*p1++);
/* imprime o equivalente decimal de cada caractere */
  } while(!strcmp(s,"done"));
}
```

Figura 7.2 A versão correta do programa da Figura 7.1

VETORES DE PONTEIROS

Os vetores de ponteiros poderão ser colocados em forma de vetor da mesma maneira que outros tipos de dados. A declaração para um vetor de ponteiros **int** com 10 elementos é

```
int *x[10];
```

Para atribuir o endereço de uma variável inteira chamada de **var** ao terceiro elemento do vetor de ponteiros, você escreveria

```
int var;

x[2] = &var;
```

Lembre-se de que você está operando com um vetor de ponteiros. Os únicos valores que os elementos do vetor podem conter são os endereços de variáveis inteiras. Para encontrar o valor de **var**, você escreveria

```
*x[2]
```

Caso você queira passar um vetor de ponteiros para uma função, poderá utilizar o mesmo método utilizado para outros vetores – basta chamar a função pelo nome do vetor, sem nenhuma indexação. Por exemplo, a função que vai receber o vetor **x**, que foi declarada anteriormente, fica parecida com:

```
func1(q)
int *q[];
{
int t;

t = *q[2]; /* atribui o valor de um inteiro apontado
              pelo terceiro elemento do vetor q */
}
```

Você deve utilizar [ ] para representar o vetor. **q** não é um ponteiro de inteiros, mas um ponteiro para um vetor de ponteiros para inteiros. Nunca esqueça o tipo de ponteiros que você está utilizando.

Uma utilização comum do vetor de ponteiros é manter ponteiros para mensagens de erro. Você pode criar uma função que vai emitir uma mensagem baseada no número inteiro dessa mensagem. Por exemplo, **serror( )** utiliza **init_messages( )** para inicializar **err**, um vetor de ponteiros de caracteres que é global.

```
char *err[5];
init messages()
{
err[0] = "erro de sintaxe";
err[1] = "parentesis esperado";
err[2] = "variavel indefinida";
err[3] = "rotulo duplicado";
}
```

```
serror(i) /* imprime erro i */
int i;
{
printf(err[i]);
}
```

Como podemos ver, **printf( )** no interior de **serror( )** é chamado com um ponteiro de caracteres, que aponta para uma das várias mensagens de erro.

Agora você deve observar a semelhança entre esse código e a maneira pela qual os argumentos de linha-de-comando são acessados. Lembre-se, os argumentos de linha-de-comando estão contidos em **argv**, um vetor de ponteiros de caracteres.

PONTEIROS DE VETORES EM GERAL

Os ponteiros para qualquer tipo de vetor operam como uma forma alternativa de indexação. Existem alguns conceitos que você deve guardar quando utiliza ponteiros para vetores.

Comparações entre ponteiros que não acessam o mesmo vetor são inválidas e *causarão* erros. Você nunca saberá onde os seus dados serão colocados na memória, se serão colocados ali novamente da mesma maneira, ou se cada compilador vai tratá-los da mesma forma. Portanto, fazer qualquer comparação entre ponteiros para dois vetores diferentes dará resultados inesperados. Por exemplo,

```
char s[80];
char y[80];
char *p1, *p2;
p1=s;
p2=y;
if(p1 < p2)...
```

é intrinsecamente um conceito errado. Não faça esse tipo de programação a não ser que sua aplicação seja muito insólita e necessite do conhecimento da localização da memória de certas variáveis.

Um erro relacionado presume que dois vetores próximos podem ser indexados como um único, simplesmente incrementando o ponteiro pelo limite do vetor. Por exemplo, este código

```
int first[10];
int second[10];
int *p,t;
p = first;
for(t = 0;t < 20;++t) *p++ = t;
```

não pode ser utilizado para inicializar os vetores **first** e **second** com os números de 0 até 19. Mesmo que possa funcionar em alguns compiladores, sob certas circunstâncias, presume que os dois vetores serão colocados justapostos na memória. Pode não ser sempre o caso, e normalmente vai resultar em problemas.

PONTEIROS DE PONTEIROS

Quando você operava com vetor de ponteiros, na verdade estava operando com *ponteiro de ponteiros.* O conceito de vetor de ponteiros é direto, porque os índices tornam o significado claro. Entretanto, ponteiros de ponteiros poderá ser bem confuso.

O ponteiro de um ponteiro é uma forma de endereçamento múltiplo, ou uma corrente de ponteiro. Como você pode ver na Figura 7.3, no caso de um ponteiro normal, o valor do ponteiro é o endereço da variável que contém o valor desejado. No caso do ponteiro de um ponteiro, o primeiro ponteiro contém o endereço do segundo ponteiro, que aponta para a variável que contém o valor desejado.

Endereçamento múltiplo poderá ter quantos ponteiros forem necessários, mas são poucos os casos em que se precisa mais do que o ponteiro de um ponteiro – ou mesmo recomendável de se utilizar. Um endereçamento indireto excessivo é difícil de ser seguido e sujeito a erros conceituais. (Não confunda endereçamento indireto com o conceito mais simples de *listas ligadas*, como aquelas dos bancos de dados.)

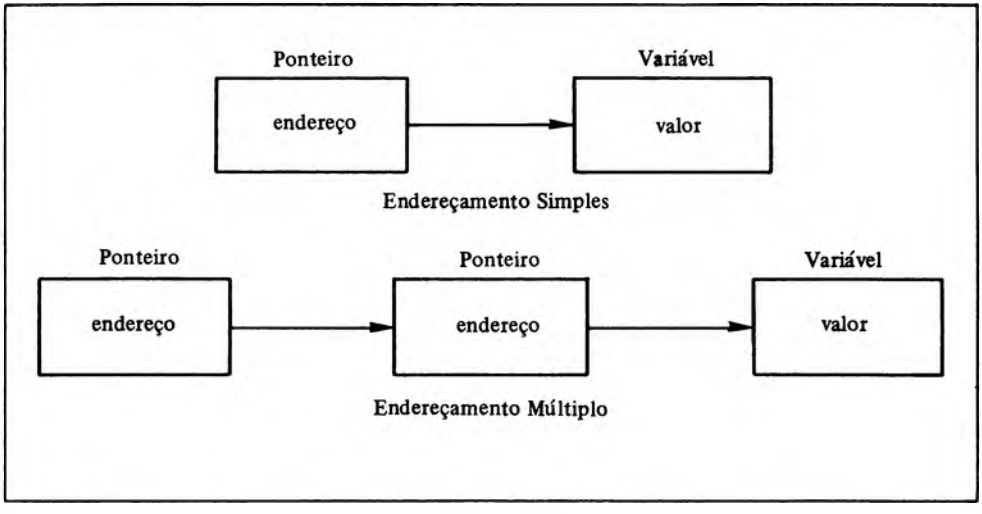

Figura 7.3 Endereçamento simples e múltiplo

Um exemplo de como utilizar o ponteiro do ponteiro é o programa da Figura 7.4.

O programa declara **p** como o ponteiro de um inteiro e **q** como o ponteiro do ponteiro de um inteiro. A chamada ao **printf( )** imprimirá o número 10 na tela. Consiste em uma utilização fictícia do endereçamento múltiplo, mas serve para mostrar de forma clara o procedimento.

```
main()
{
int x,*p,**q;

x = 10;
p = &x;
q = &p;
printf("%d\n",**q); /* imprime o valor de x */
}
```

Figura 7.4 Um programa que mostra a utilização do ponteiro de um ponteiro.

INICIALIZANDO PONTEIROS

Após a declaração de um ponteiro, mas antes de sua atribuição, seu conteúdo será sem nexo. Caso você tente utilizar o ponteiro antes de dar-lhe um determinado valor, provavelmente vai destruir seu programa e também o sistema operacional de seu computador.

Como norma, um ponteiro poderá ser inicializado em nulo, normalmente zero, significando que não está apontando para nada. A idéia do ponteiro que tem um valor nulo é a de que não está apontando para nada, está livre para ser utilizado. É um procedimento que deve ser seguido. Porém, mesmo atribuindo a ele um valor nulo, ainda poderá destruir seu programa e o sistema operacional de seu computador.

Por exemplo, a função **malloc( )**, encontrada na biblioteca padrão, devolve o endereço da memória solicitada ou um nulo. Um nulo significa que sua solicitação de memória foi negada devido à falta de memória livre.

Você poderá utilizar o ponteiro nulo para facilitar diversas rotinas de ponteiros. Em uma função que procura um vetor de ponteiros, se você assumir que a procura pode parar quando um ponteiro nulo for encontrado, não haverá necessidade de procurar todo o vetor. Neste exemplo,

```
search(p)
int *p[];
{
register int t;

for(t = 0;t < array_size && p[t];++t)
  if(!pt[t]) return t;
return -1; /* nao encontrado */
}
```

search( ) presume que o vetor de ponteiros utilizado na chamada conterá endereços válidos ou nulos. Isso é um pressuposto de que foi inicializado em outro local.

Como com qualquer variável, os vetores de ponteiros podem ser inicializados quando são declarados. Caso você se lembre da função **serror( )** do início deste capítulo, o vetor **err**, que continha ponteiros para as mensagens de erro, foi inicializado explicitamente em uma rotina separada. O vetor **err** também era global. É possível eliminar a função de inicialização fazendo de **err** um vetor local **estática**. A seguinte versão de **serror( )** não exige inicialização ou variáveis globais.

```
serror(i) /* versao melhorada */
int i;
{
static char *err[] =
  {
  "erro de sintaxe",
  "parentesis esperado",
  "variavel indefinida",
  "rotulo duplicado"
  };
printf(err[i]);
}
```

O par **[ ]** após **err** indica ao compilador que ele deve contar os elementos e criar um vetor com tamanho suficiente para mantê-los. Esta versão é autônoma e não reinicializa o vetor a cada chamada.

PONTEIROS DE FUNÇÕES

Uma utilização confusa, porém bastante poderosa dos ponteiros é como *ponteiros de função*. Apesar da função não ser uma variável, tem uma posição na memória que pode ser atribuída a um ponteiro. Este ponteiro poderá, em seguida, ser utilizado para manipular chamadas de funções.

Para compreender o que significa o ponteiro de uma função, você deve entender um pouco sobre a compilação de uma função e a sua chamada em C. Primeiramente, à medida que cada função é compilada, código-fonte é transformado em código-objeto, que executa as atividades da função. Segundo, no momento do encadeamento, o endereço em que o código de função inicia é conhecido. Quando uma chamada a uma função é feita durante o processamento do programa, uma "chamada" em linguagem de máquina é feita para o endereço de memória da função. Portanto, o ponteiro de uma função de fato contém o endereço de memória do início do código da função.

Para entender o conceito do ponteiro de uma função, considere o programa da Figura 7.5, prestando bastante atenção nas declarações.

strcmp( ) é a função padrão de comparação de série, encontrada na biblioteca padrão. É declarada em **main( )** para que o programa saiba que tipo de valor está devolvendo – neste caso, um inteiro – e que se trata de uma função, e não de uma variável. Com a chamada da função

check( ), são passados como parâmetros dois ponteiros de caracteres e um ponteiro da função. Dentro da função **check( )**, os argumentos são declarados como ponteiros de caracteres e um ponteiro de função. Ao declarar um ponteiro de função você deve utilizar exatamente o mesmo método da Figura 7.5. Os parênteses são necessários para que o compilador possa interpretar corretamente esta declaração. Sem a colocação dos parênteses em torno de ***cmp**, o compilador presume que você simplesmente está declarando uma função, o que não é o caso aqui.

```
main()
{
int  strcmp(); /* declara uma funcao */
char s1[80],s2[80];

gets(s1);
gets(s2);
check(s1,s2,strcmp);
}

check(a,b,cmp);
char *a,*b;
int (*cmp)();
{
printf("testando igualdade\n");
if(!(*cmp)(a,b)) printf("igual\n");
else printf("desigual\n");
}
```

Figura 7.5 Um programa que utiliza um ponteiro para uma função

Uma vez dentro de **check( )**, você pode ver como a função **strcmp( )** é chamada.

A declaração

```
(*cmp)(a,b)
```

vai executar a chamada a uma função – nesse caso, **strcmp( )**, apontado por **cmp** com os argumentos **a** e **b**.

Você poderia perguntar por que alguém desejaria escrever um programa desta maneira. Obviamente nada se ganha e ocorre muita confusão. Porém, existem situações em que é vantajoso passar-se funções arbitrárias para procedimentos ou manter um vetor de funções.

Apesar de não ser parte deste livro o desenvolvimento de um exemplo detalhado, a discussão que segue pode ajudar na ilustração da utilização de ponteiros de função. Ao ser

escrito um compilador, é comum para aquela parte dele que avalia expressões aritméticas executar chamadas de funções para diversas rotinas de suporte; por exemplo, as funções seno, co-seno e tangente. Em lugar de ter uma declaração **switch** listando todas estas funções, um vetor de ponteiros de função pode ser mantido com a geração direta de um índice. Uma vez conhecido o índice, pode ser chamada a função apropriada.

A função **check** da Figura 7.6 poderá verificar igualdade alfabética ou igualdade numérica, bastando uma pequena alteração.

```
main()
{
int strcmp(); /* declara uma funcao */
int numcmp();
char s1[80],s2[80];

gets(s1);
gets(s2);
printf("numerico (n) ou alfabetico (a)? ");
if(getchar() == 'a') check(s1,s2,srtcmp);
else check(s1,s2,numcmp);
}

check(a,b,cmp)
char *a,*b;
int (*cmp)();
{
printf("testando igualdade\n");
if((*cmp)(a,b)) printf("igual\n");
else printf("desigual\n");
}

numcmp(a,b)
char *a,*b;
{
if(atoi(a) == atoi(b)) return 0;
else return 1;
}
```

Figura 7.6 Um programa, que mostra a utilização dos ponteiros de função

Os dois tipos de igualdade podem ser testados com uma chamada para a mesma função.

Talvez você nunca tenha necessidade de utilizar ponteiros para funções. Eles existem porque C substitui a linguagem de programação assembly. No assembly é possível carregar um registro de endereçamento e executar uma chamada para este endereço. Os ponteiros de função permitem o mesmo procedimento.

PONTEIROS NÃO SÃO INTEIROS

Ponteiros não são inteiros – também não são inteiros sem sinal. Por exemplo, o programa da Figura 7.7 aloca uma região de memória e a atribui para um ponteiro de caractere.

Apesar da aparência correta do programa, e de ele ser corretamente compilado e processado em todos os microprocessadores existentes e implementações C, não está *tecnicamente* correto. A razão é que o compilador presume que **malloc( )** devolve um inteiro, quando em verdade está devolvendo um ponteiro de caracteres. Na maioria das vezes isso não causa problema, mas para garantir a portabilidade, o programa deverá ser reescrito de acordo com a Figura 7.8.

A razão pela qual os ponteiros, de uma maneira geral, não podem ser tratados como inteiros é a de que alguns processadores podem executar alguns acertos de sinal neles, ou que o tamanho de um inteiro pode não ser o tamanho do endereço. Seja qual for a razão, deve-se tomar cuidado.

```
main()
{
char *p;

p = malloc(100); /* aloca 100 bytes */
}
```

Figura 7.7 Um programa tecnicamente incorreto que aloca uma região de memória e a atribui para um ponteiro de caractere

```
main()
{
char *p, *malloc();

p = malloc(100); /* aloco 100 bytes */
}
```

Figura 7.8 A versão correta do programa da Figura 7.7

PROBLEMAS COM PONTEIROS

Nada pode causar tanto problema como um ponteiro "perdido". Os ponteiros são abençoados, porém podem ser confusos. Fornecem-lhe um tremendo potencial e são necessários em diversos programas de sistema, mas quando um ponteiro acidentalmente contém um valor errado, poderá ser difícil a sua localização.

Um ponteiro com erro é difícil de ser encontrado, porque o problema não está no ponteiro em si; o problema consiste em que a cada operação feita com ele, você esteja lendo ou escrevendo em algum local desconhecido da memória. Se você estiver lendo, o pior que pode acontecer é você receber "lixo". Entretanto, caso você escreva nela, estará escrevendo sobre outros elementos de código ou dados. Isso, por sua vez, poderá se manifestar bem mais tarde, durante a execução de seu programa, fazendo com que você procure o erro em local errado. Poderá haver pouca ou nenhuma evidência de que o ponteiro é o problema. Esse tipo de problema já causou muita insônia para os programadores.

Pelo fato dos erros com ponteiros causarem tantos pesadelos, você deve fazer o melhor que puder para evitá-los. Para evitar erros com ponteiros, você deve saber como criá-los, para nunca errar. O exemplo clássico de um ponteiro com erro é o *ponteiro não inicializado*, mostrado na Figura 7.9.

O **p** da Figura 7.9 contém um endereço desconhecido, pois nunca foi definido. Você não tem meios de saber onde é que o valor de **x** foi escrito. Quando o programa é pequeno, como no caso desta figura, a possibilidade é de que **p** contenha um endereço que não está em sua área de código ou de dados. Na maioria das vezes o programa parece operar corretamente. Entretanto, à medida que seu programa aumenta, a probabilidade de que **p** tenha um ponteiro para o seu código de programa ou área de dados *aumenta*. Eventualmente o programa pára de operar. A solução consiste em se ter certeza de que o ponteiro esteja apontando algo válido, antes que seja utilizado.

```
main() /* este programa esta incorreto */
{
int x,*p;

x = 10;
*p = x;
}
```

Figura 7.9 Um programa incorreto que utiliza um ponteiro não inicializado

Um segundo erro, muito comum, é provocado pela simples falta de compreensão de como utilizar um ponteiro. Na Figura 7.10, a chamada de **printf( )** não vai imprimir o valor de **x**, que é 10, na tela. Vai imprimir algum valor desconhecido, porque a declaração

```
p = x;
```

está errada. Esta declaração atribui o valor 10 ao ponteiro **p**, que deveria conter um endereço e não um valor. Para corrigir o programa você deve escrever

```
p = & x;
```

```
main() /* este programa esta incorreto */
{
int x,*p;

x = 10;
p = x;
printf("%d\n",*p);
}
```

Figura 7.10 Um programa que utiliza erroneamente um ponteiro

O fato de os ponteiros utilizados incorretamente causarem erros complicados não é motivo para que não sejam utilizados. Basta que se tome cuidado e que se tenha certeza para onde o ponteiro está apontando, antes de utilizá-lo.

EXERCÍCIOS

1. Seja **x** um inteiro e **p** um ponteiro de inteiro. Escreva o trecho de código que determina para **x** o valor 10, utilizando o ponteiro **p1**.
2. Escreva uma função denominada **swapg( )**, que mude o valor de dois inteiros se, e apenas se, o primeiro argumento for maior que o segundo.
3. Reescreva as rotinas **push( )** e **pop( )** de modo que todos os 25 elementos possam ser utilizados.
4. Utilizando ponteiros, em lugar de indexação de vetor, reescreva **putw( )** do Capítulo 6.
5. O que há de errado com esta função?

```
func1()
{
char *p;
char s[80];

p = s[0];
gets(s);
printf("%s",p);
}
```

6. Escreva uma função curta que imprima o valor de cada elemento apontado por um vetor de ponteiros **float**. Assuma que o vetor tem dez elementos.

RESPOSTAS

```
1. p  = &x;
   *p = 10;

2. swapg(a,b)
   int *a,*b;
   {
   register int t;

   if(*a <= *b) return; /* sem troca */
   t = *a;
   *a = *b;
   *b = t;
   }

3. push(i)
   int i;
   {
   if(p1 == (tos + 25))
     {
     printf("estouro da pilha\n");
     exit();
     }
   *p1 = i;
   p1++;
   }

   pop()
   {
   if((p1 == tos)
     {
     printf("estouro inferior da pilha\n");
     exit();
     }
   p1--;
   return *p1;
   }

4. putw(i,fp)
   int i;
   FILE *fp;
   {
   char *t;
```

```
t = &i; /* atribui o endereco de i a *t */
putc(*t++,fp);
putc(*t,fp);
}
```

5. O **p** recebe o valor do primeiro elemento do vetor s, e não o seu endereço. A intenção foi atribuir o endereço do primeiro elemento da matriz.

6.
```
func1(f)
float *f[];
{
register int t;

for(t = 0;t < 10;++t) printf("%f\n",*f[t]);
}
```

CAPÍTULO 8

VETORES

Os capítulos anteriores apresentaram os vetores de uma dimensão. Neste capítulo, serão cobertos diversos tópicos sobre vetores, com ênfase nos vetores de várias dimensões.

Em C todos os vetores consistem em posições de memória contíguas. O endereço mais baixo corresponde ao primeiro elemento, e o endereço mais alto ao último elemento.

A declaração geral de um vetor unidimensional é

tipo var_nome[dimensão];

Esta declaração é similar à declaração **DIM** do BASIC. Enquanto algumas versões do BASIC apresentam um comprimento padrão para vetores, em C é exigida a informação do tamanho do vetor para cada declaração.

Os vetores em C iniciam em zero. Quando você escreve

```
char p[10];
```

você está declarando um vetor de caracteres formado por 10 elementos de vetor, p [0] até p [9]. Os seguintes exemplos, em C e em BASIC, ilustram este ponto importante:

```
int x[10]; /* reserva 10 elementos inteiros */
int t;

for(t = 0;t < 10;++t) x[t] = t;
```

```
10 DIM X(10)
20 FOR T=1 TO 10
30    X(T) = T
40 NEXT
```

A versão em BASIC indexa o vetor X de 1 até 10, enquanto a versão C indexa de 0 até 9. Apesar da maioria das versões BASIC permitirem que se inicie um vetor em 0, a maioria dos programadores BASIC não o faz. Caso você seja um programador BASIC, levará um certo tempo para se acostumar a iniciar com 0. A diferença fundamental é que, em BASIC, o número na declaração **DIM** normalmente especifica o último elemento, enquanto em C, o número na declaração especifica o número de elementos.

Em C não há verificação de limites; você pode ultrapassar qualquer uma das duas extremidades de um vetor e escrever nos dados de outra variável ou mesmo em um trecho do programa. Em programas escritos corretamente, isso normalmente não representa nenhum problema, mas pode sê-lo enquanto você aprende a programar em C. Portanto, tenha um certo cuidado. Por exemplo, tenha certeza de que os vetores de caracteres que aceitam a entrada de caracteres utilizando **gets( )** tenham o comprimento suficiente para aceitar a maior entrada.

Quando você passa um vetor como sendo o argumento de uma função, o endereço do primeiro elemento do vetor é passado para a função. Não é feita uma cópia completa do vetor. Isso significa que qualquer operação nele executada, enquanto estiver dentro da função, afeta-o fora da função. Em outras palavras, você pode alterar os argumentos de um vetor de função.

VETORES UNIDIMENSIONAIS

Vetores unidimensionais são basicamente listas de informação de um determinado tipo. O exemplo mais comum de vetor unidimensional é uma série de caracteres. Quando um vetor de caracteres é declarado para guardar uma série, você deve permitir um byte extra para o terminador nulo, que é parte de toda série. Por exemplo, se você quisesse declarar um vetor s para uma série de 10 caracteres, escreveria

```
char s[11];
```

Isso fornece o espaço para o nulo no final da série.

Ao passar vetores unidimensionais para funções, você chama a função com o nome do vetor. Por exemplo, para passar o vetor s para a função **gets( )**, você escreveria

```
gets(s);
```

Isso faz com que o endereço do primeiro elemento de s seja passado para **gets( )**. Em verdade, você está passando um ponteiro de caracteres.

Se uma função vai receber um vetor unidimensional, você pode declarar o parâmetro formal em uma das duas formas: como um ponteiro ou como um vetor. Por exemplo, para receber s em uma função denominada **func1( )**, você pode declarar **func1( )** como

```
func1(str)
char *str;
{
.
.
.
```

ou

```
func1(str)
char str[ ];
{
.
.
.
```

Os dois métodos de declaração são idênticos porque os dois informam ao compilador que um ponteiro será recebido.

Realmente, no que diz respeito ao compilador,

```
func1(str)
char str[11];
{
.
.
.
```

também funciona, porque C vai gerar um código que instrui **func1( )** a receber um ponteiro de caracteres. Como não há verificação de limites, um vetor de qualquer dimensão poderá ser passado para a rotina chamada, mesmo que tenha sido declarada apenas uma dimensão de 11.

VETORES BIDIMENSIONAIS

C permite vetores multidimensionais. A forma mais simples de vetor multidimensional é o vetor bidimensional. Um vetor bidimensional, é, em resumo, um vetor de vetores unidimensionais. Para declarar um vetor de inteiros bidimensional **d** com tamanho 10, 20, você escreveria

```
int d[10][20];
```

Preste atenção na declaração: ao contrário da maioria das outras linguagens de computador, que utilizam a vírgula para separar as dimensões do vetor, C coloca cada dimensão em colchetes próprios.

Igualmente, para acessar o ponto 3,5 do vetor **d**, você utilizaria

```
d[3][5]
```

Caso você conheça BASIC os programas da Figura 8.1 serão úteis. Tanto a versão em BASIC como aquela em C vão carregar um vetor bidimensional com os números de 1 até 12.

Na versão C, num[0] [0] terá o valor 1, num[0] [1] o valor 2, num[0] [3] o valor 3, e assim em diante. O valor de num[2] [3] será 12.

Vetores bidimensionais são armazenados em um vetor de linhas e colunas, onde o primeiro índice indica a linha e o segundo indica a coluna. Isso significa que o índice à direita muda mais rapidamente que o índice à esquerda, ao passarmos seqüencialmente pelo vetor. A Figura 8.2 mostra um vetor bidimensional na memória. Em resumo, o primeiro índice pode ser encarado como um ponteiro para a linha desejada.

Lembre-se de que o armazenamento para todos os elementos do vetor é alocado permanentemente. No caso de um vetor bidimensional, a fórmula que segue encontra o número de bytes de memória:

bytes = linha * coluna * número de bytes do tipo de dado

```
main()
{
int t,i,num[3][4];

for(t = 0;t < 3;++t)
  for(i = 0;i < 4;++i)
    num[t][i] = (t * 4) + i + 1;
}
```

```
10 DIM N(3,4)
20 FOR T=1 TO 3
30   FOR I=1 TO 4
40     N(T,I) = ((T-1)*4)+I
50   NEXT
60 NEXT
```

Figura 8.1 As versões C e BASIC de um programa que carrega um vetor bidimensional

Portanto, um vetor inteiro com dimensão de 10,5 deveria ter $10 \times 5 \times 2$, ou seja, 100 bytes alocados de memória. Lembre-se de que vetores grandes utilizam muita memória, de modo que você deve tomar o devido cuidado em declarar um vetor com o tamanho suficiente para a sua finalidade.

Ao passar vetores bidimensionais para funções, você apenas passa o vetor do primeiro elemento. Você pode fazer isso, utilizando o nome do vetor sem indexação. Porém, uma função, recebendo um vetor bidimensional como parâmetro, deverá definir o comprimento da segunda dimensão. Por exemplo, uma função que vai receber um vetor inteiro bidimensional com dimensões 10,10 seria declarada como a seguir:

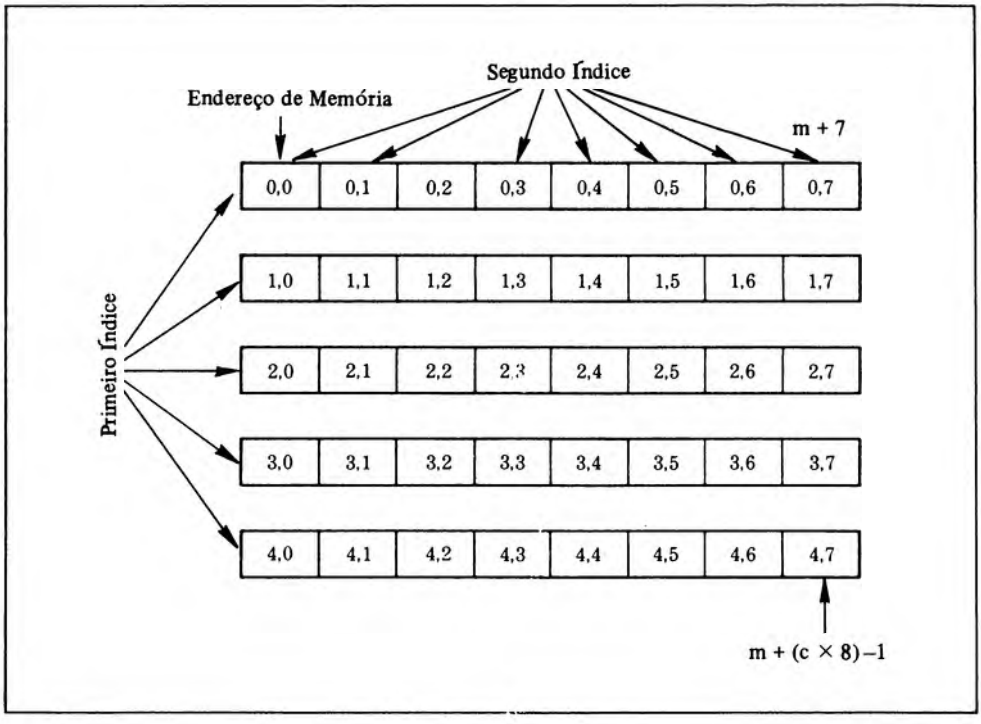

Figura 8.2 Vetor bidimensional na memória

```
func1(x)
int x[ ][10];
{



}
```

É possível fornecer também a primeira dimensão, mas não é necessário.

C precisa conhecer a segunda dimensão para poder operar com declarações do tipo

```
x[2] [4]
```

dentro da função. Caso o comprimento das linhas não seja conhecido, será impossível saber onde inicia a terceira linha.

A Figura 8.3 mostra o programa que utiliza um vetor bidimensional para armazenar a nota para cada estudante nas classes de um determinado professor. O programa presume que o professor tenha um máximo de 3 classes, com um máximo de 30 alunos por classe. Estude este exemplo para ver como os elementos do vetor são acessados.

```
#include <stdio.h>

#define CLASSES 3
#define GRADES  30

int grade[CLASSES][GRADES];

main()  /* programa de notas de classe */
{
char ch;

do
  {
  do
    {
    printf("(E)ntre notas\n");
    printf("(R)eporta notas\n");
    printf("(F)im\n");
    ch = toupper(getchar());
    fflush(stdin);
    } while(ch != 'E' && ch != 'R' && ch != 'Q');
  switch(ch)
    {
    case 'E' : enter_grades();
               break;
    case 'R' : disp_grades(grade);
               break;
    case 'Q' : exit(0);
    }
  } while(1);
}

enter_grades()
{
int t,i;

for(t = 0;t < CLASSES;t++)
```

Figura 8.3 Um programa que utiliza um vetor bidimensional para armazenar notas escolares

```
  {
  printf("Classe # %d:\n",t + 1);
  for(i = 0;i < GRADES;++i) grade[t][i] = get_grade(i);
  }
}

get grade(num)
int num;
{
char s[80];

printf("entre nota para o estudante # %d:\n",num + 1);
gets(s);
return(atoi(s));
}

disp grades(g)
int g[][CLASSES];
{
int t,i;

for(t = 0;t < CLASSES;++t)
  {
  printf("Classe # %d:\n",t + 1);
  for(i = 0;i<GRADES;++i)
     printf("nota para o estudante #%d e' %d\n",
              i + 1,g[t][i]);
  }
}
```

Figura 8.3 *(continuação)*

A função **disp_grades( )** foi chamada com o vetor **grades** passado como argumento. Observe que a segunda dimensão do parâmetro g foi especificada explicitamente, conforme é requerido ao se passar vetores bidimensionais.

VETORES MULTIDIMENSIONAIS

C permite vetores com mais de duas dimensões. O limite exato, caso exista, será determinado pelo seu compilador. A forma geral da declaração de um vetor multidimensional é

nome tipo [a] [b] [c] ... [z];

Vetores de três ou mais dimensões não são utilizados com freqüência, devido à quantidade de memória necessária para mantê-las. Como já foi dito, o armazenamento de todos os elementos do vetor é alocado permanentemente durante a execução de seu programa. Por exemplo, um vetor de caracteres de quatro dimensões com dimensões 10, 6, 9, 4 exigiria $10 \times 6 \times 9 \times 4$, ou seja 2160 bytes. Caso o vetor fosse inteiro, seriam alocados 4320 bytes. Se o vetor fosse de duplo flutuante, seriam necessários 34.560 bytes. A área necessária cresce exponencialmente com o número de dimensões. O computador leva tempo para gerar cada índice e isto poderá fazer com que vetores multidimensionais acessem mais lentamente que vetores de uma dimensão com o mesmo número de elementos. Por essas e outras razões, quando forem necessários vetores multidimensionais, você deverá alocar dinamicamente partes do vetor, conforme a necessidade, e utilizar ponteiros. Entretanto, esse procedimento, *processamento por economia de vetor* não faz parte do escopo deste livro.

Ao passar vetores multidimensionais para funções, você deve declarar tudo menos a primeira dimensão. Por exemplo, caso você declare o vetor **m** como

```
int m[4] [3] [6] [5];
```

então, a função **func1(1)**, que vai receber **m**, fica assim

```
func1(d)
int d[ ][3][6][5];
{
.
.
.
```

VETORES *VERSUS* PONTEIROS EM VETORES MULTIDIMENSIONAIS

No último capítulo você aprende que ponteiros e vetores unidimensionais podem facilmente ser intercambiáveis. Isso permanece verdadeiro para vetores com duas ou mais dimensões. Por exemplo, considere o vetor:

```
char p[10];
```

As seguintes declarações são idênticas:

```
P
&p[0]
```

Esta declaração

```
p == &p[0]
```

avalia corretamente porque o endereço do primeiro elemento do vetor é o mesmo que o endereço do vetor, que é derivado pela utilização de seu nome, sem índice.

Ao considerar o vetor de caracteres a, que tem duas dimensões 10,10 você pode ver que essas duas declarações são idênticas:

```
a
&a[0][0]
```

O elemento 0,4 pode ser referenciado de duas maneiras: ou pela indexação esperada de vetor, **a[0][4]**, ou por ponteiros, ***(a+4)**. Portanto, o elemento 1,2 é **a[1][2]** ou ***(a+12)**. Para um vetor bidimensional, **a[j][k]** é equivalente a ***(a+(j*comprimento da linha)+k)**.

De um certo modo, um vetor bidimensional é como um vetor de ponteiros de linha e vetores unidimensionais de linhas. Assim, uma maneira simples de utilizar ponteiros para acessar vetores bidimensionais é pela utilização de uma variável apontadora separada. A razão pela qual utilizaríamos ponteiros em lugar da indexação de vetor padrão seria a velocidade e a eficiência. A função que segue imprimirá o conteúdo da linha especificada para o vetor de inteiros global **num**:

```
num[10][10];
pr_row(j)
int j;
{
int *p,t;
p = &num[j][10]; /* endereco do primeiro elemento
                    na linha j */
for(t = 0;t < 10;++t) printf("%d ",*(p + t));
}
```

A malha que imprime os elementos da linha é processada com maior rapidez, com a utilização de aritmética de ponteiros do que com a aplicação de indexação de vetor. Isso se deve ao fato de que os ponteiros são incrementados com um simples conjunto de instruções de máquina, mas indexar um vetor exige um maior processamento, que leva mais tempo.

Esta rotina pode ser completamente generalizada, fazendo com que os argumentos de chamada sejam a linha, o comprimento da linha e um ponteiro para o primeiro elemento do vetor, conforme indicado abaixo:

```
pr_row(j,row_dimension,p) /* generalizado */
int j,row_dimension;
int *p;
{
int t;
p = p + (j * row_dimension);
for(t = 0;t < row_dimension;++t)
  printf("%d ",*(p + t));
}
```

Vetores com mais de duas dimensões podem ser focalizados da mesma maneira. Por exemplo, um vetor tridimensional pode ser reduzido a um ponteiro para um vetor bidimensional, que pode ser reduzido a um ponteiro para um vetor unidimensional.

Generalizando, um vetor de **n** dimensões pode ser reduzido a um ponteiro para um vetor de n−1 dimensões. Este novo vetor pode ser novamente reduzido pelo mesmo procedimento. Esse processo termina quando um vetor unidimensional é obtido.

VETORES ALOCADOS

Em C você pode liberar e alocar memória dinamicamente, utilizando as rotinas **malloc( )** e **free( )**, da biblioteca padrão. (Em algumas implementações **alloc( )** é a função de alocação.) Caso a memória seja limitada e você necessite de um vetor durante um curto período, você poderá alocá-lo utilizando **malloc( )**, e devolvê-lo para a memória livre utilizando **free( )** ao terminar o trabalho.

Este fragmento aloca 1000 bytes de memória.

```
char *p;

p = malloc(1000); /* aloca 1000 bytes */
```

p aponta para o primeiro dos 1000 bytes de memória livre. Caso você precise utilizar essa memória como um vetor bidimensional de dimensões 10,10 para executar o processamento do vetor, pode declarar uma função semelhante à seguinte:

```
process(s)
char s[][10];
{
/* vetor process */
}
```

Isso engana o compilador C, simulando um vetor de 10 por 10 caracteres. Na verdade, você tem um vetor de 10 por 10 caracteres dentro da função; a diferença é que você executou a alocação manualmente, utilizando **malloc( )**, em vez de automaticamente, pela utilização da declaração normal de declaração-de-vetor.

A seqüência principal de alocação, processamento e dealocação é apresentada no trecho de programa seguinte. Este vai alocar memória para um vetor de caracteres de 10 por 10, passar o ponteiro daquela memória para uma função denominada **process( )**, e liberar a memória quando o processo voltar:

```
main( )
{
        char *p;
        .
        .
        .
        p = malloc(1000;
        process(p);
        free(p);
        .
        .
        .
}
process(ax)
char ax[10] [10];
{
        .
        .
        .
        /* ax pode ser utilizado como um
          vetor de caracteres bidimensional normal */


}
```

Pelo fato de uma codificação dessa natureza poder ser causadora de enganos e confusão para alguém que lê o seu programa, isso deverá ser feito apenas quando houver pouca disponibilidade de memória. É melhor, sempre que possível, declarar os vetores explicitamente. Entretanto, a utilização de **malloc( )** e **free( )**, e alocação dinâmica em geral certamente é aceitável.

UM EXEMPLO MAIS LONGO

Os vetores bidirecionais são normalmente utilizados para simular jogos de mesa matriciais, como xadrez e dama. Como não é parte do escopo deste livro apresentar um programa de xadrez ou damas, observe a Figura 8.4, que lista um programa que joga uma partida simples de "jogo da velha". O campo do jogo será representado utilizando um vetor de 3-por-3 caracteres.

O computador joga uma partida muito simples. Você é sempre **X** e o computador é **0**. Quando é a vez do computador jogar, ele apenas varre o vetor e coloca seu **0** na primeira posição vaga. Caso não encontre uma posição livre, relata fim de jogo e termina.

Para simplificar a malha principal e a rotina de apresentação do vetor, presume-se que a rotina **check( )** devolve um espaço caso ainda não haja vencedor. **check( )** devolve um **X** se você ganhou ou um **0** se o vencedor foi o computador. Para tanto, o vetor deve ser inicializado para conter espaços.

A rotina **get_player_move( )** é recursiva quando uma posição inválida é colocada. O programa da Figura 8.4 é um exemplo em que a recursão pode ser utilizada para simplificar a rotina e reduzir o código.

A função **disp_matrix( )** apresenta a situação atual do jogo. Você deveria ter condição de ver como a inicialização do vetor com espaços simplificou esta função.

O **check( )** verifica o vetor após cada movimento, para ver se existe um vencedor. Varre as linhas, as colunas e em seguida as diagonais. Você pode ver como o programa foi simplificado, utilizando um espaço para determinar o vencedor. No início do jogo, essa rotina devolve um espaço rapidamente, pois o vetor está cheio de espaços.

Todas as rotinas deste exemplo acessam o vetor **matrix** de maneiras diferentes. Estude-as para ter certeza de ter entendido a operação com vetores.

```
#define SPACE  ' '

char matrix[3][3]; /* matriz jogo da velha */

main()
{
char done;

printf("Este e o Jogo da Velha.\n");
printf("Voce ira jogar contra o computador.\n");
done = SPACE;
init_matrix();
do
  {
  disp_matrix();  get_player_move();
  done = check();  /* veja se vencedor */
  if(done != SPACE) break;  /* vencedor! */
  get_computer_move();
  done = check();  /* veja se vencedor */
  } while(done == SPACE);
if(done == 'X') printf("Voce ganhou!\n");
else printf("Eu ganhei!!!!\n");
disp_matrix();   /* mostre as posicoes finais */
}

init_matrix()
{
register int   t;
         char *p;
```

Figura 8.4 Um programa que faz o jogo da velha

```
p = (char *)matrix;
for(t = 0;t < 9;++t) *(p+t) = SPACE;
}

get_player_move()
{
int x,y;

printf("Entre coordenadas para seu X: ");
scanf("%d%d",&x,&y);
x--;
y--;
if(matrix[x][y] != SPACE)
  {
  printf("Movimento Invalido. Tente de novo.\n");
  get_player_move();
  }
else matrix[x][y] = 'X';
}

get_computer_move()
{
register int   t;
         char *p;

p = (char *)matrix;
for(t = 0;*p != SPACE && t < 9;++t) p++;
if(t == 9)
  {
  printf("empate.\n");
  exit(0);                          /* fim do jogo */
  }
else *p = '0';
}

disp_matrix()
{
int t,i;

for(t = 0;t   3;t++)
  {
```

Figura 8.4 *(continuação)*

```
  printf(" %c | %c | %c ",matrix[t][0],
                          matrix[t][1],matrix [t][2]);
  if(t != 2) printf("\n---|---|---\n");
  }
printf("\n");
}

check()
{
int t;
char *p;

for(t = 0;t < 3;t++)
  {                                  /* verifica linhas */
  p = &matrix[t][0];
  if(*p == *(p + 1) && *(p + 1) == *(p + 2)) return *p;
  }
for(t = 0;t < 3;t++)
  {                                  /* verifica colunas */
  p = &matrix[0][t];
  if(*p == *(p + 3)  &&  *(p + 3) == *(p + 6))
     return *p;
  }
/* testa diagonais */
if(matrix[0][0] == matrix[1][1] && matrix[1][1]
                == matrix[2][2]) return matrix[0][0];
if(matrix[0][2] == matrix[1][1] && matrix[1][1]
                == matrix[2][0]) return matrix[0][2];
return SPACE;
}
```

Figura 8.4 *(continuação)*

EXERCÍCIOS

1. Escreva uma função denominada **load( )**, que carregue de duas maneiras um vetor de caracteres de 10 bytes denominado **a**, com as letras de a até j. A primeira maneira é utilizando indexação de vetor; a segunda é utilizando ponteiros.
2. Escreva a parte da declaração de uma função chamada de **func( )**, que receberá o vetor **num**, a qual é declarada aqui.

```
int num[100][1234];
```

3. Dado

```
char str[10][5][3];
```

escreva uma expressão de ponteiro que devolva o valor

str [2] [4] [1].

4. Quantos bytes de memória serão necessários pelos seguintes vetores? Presuma que os inteiros têm 2 bytes e os flutuantes 8 bytes.

```
a. char s[80];
b. char s[80][10];
c. int n[10];
d. float f[10][5];
e. char x[10][9][8][7];
```

5. O que há de errado com o seguinte trecho?

```
f(A)
int s[10][4][ ];
{
int t;
t = s[3][2][4];
.
.
.
}
```

6. Melhore o jogo da velha, para que jogue uma partida melhor.
7. Apenas como diversão, reescreva o jogo da velha, para que jogue contra si próprio. Tente fazer com que cada partida seja diferente.

RESPOSTAS

```
1. load()
   {
   int t;

   for(t = 0;t < 10;++t) a[t] = 'a' + t;
   }

   load()
   {
   int   t;   char *p;
```

```
    p = a;
    for(t = 0;t < 10;++t) *p++ = 'a' + t;
    }
```

2.
```
    func(n)
    int n[][1234];
    {
```

3.
```
*(str +(2 * 15) + (5 * 3) + 3);
```

4. a. 80 b. 800 c. 20 d. 400 e. 5040

5. A dimensão do vetor mais a direita está faltando. Essa dimensão é necessária para a função manipular corretamente a indexação do vetor.

CAPÍTULO **9**

ESTRUTURAS, UNIÕES E TIPOS DEFINIDOS PELO USUÁRIO

C permite que você crie novos tipos de dados, de duas maneiras: primeiro, combinando diversos tipos de variáveis em uma variável conglomerada denominada *estrutura*; e segundo, utilizando uma *união*, que permite a diversas variáveis compartilharem a mesma área de memória. tilizando **typedef** você pode também criar nomes novos para tipos de variáveis padrão. Essas características se combinam para fornecer ao C um conjunto muito rico de tipos de variáveis disponíveis ao programador.

ESTRUTURAS

A *estrutura* é uma coleção de variáveis referenciadas com um mesmo nome. Ao contrário do BASIC, que não tem como ligar variáveis, C utiliza estruturas para fornecer um meio conveniente de manter informações relacionadas em um mesmo local.

A *definição de uma estrutura* forma um meio que pode ser utilizado para criar variáveis estruturadas. Cada estrutura é formada por uma ou mais variáveis ligadas logicamente. Essas variáveis são denominadas *elementos da estrutura.*

As estruturas, como grupos de variáveis logicamente ligadas, podem ser facilmente passadas para funções. Com a utilização de estruturas, seu código-fonte poderá ser muito mais fácil de ser lido porque a conexão lógica entre os elementos da estrutura é óbvia.

Por exemplo, o agrupamento de nome e endereço em uma lista de endereçamento postal é um conjunto comum de informação relacionada. O trecho de código que segue declara uma estrutura para manter os campos de nome e endereço; a palavra-chave **struct** informa ao compilador que uma estrutura está sendo definida:

```
struct addr
{
char name[30];
char street[40];
char city[20];
char state[2];
unsigned long int zip;
};
```

Existem duas observações a respeito dessa definição. Primeiro, termina em ponto-e-vírgula, *porque uma definição de estrutura é uma declaração*. Segundo, o rótulo de estrutura **addr** identifica essa estrutura de dados em particular e é seu nome.

Até o momento *nenhuma variável foi realmente declarada*. Apenas a forma dos dados foi definida. Para realmente declarar uma variável com essa estrutura, você deveria escrever

```
struct addr ainfo;
```

Isso declara uma variável do tipo **addr**, chamada **ainfo**. Quando você define uma estrutura, na verdade está definindo uma variável complexa formada por elementos de estrutura.

Você pode ainda declarar uma ou mais variáveis ao mesmo tempo em que define a estrutura. Por exemplo,

```
struct addr
{
char name[30];
char street[40];
char city[20];
char state[2];
unsigned long int zip;
} ainfo,binfo,cinfo;
```

define uma estrutura denominada **addr** e declara as variáveis **ainfo**, **binfo** e **cinfo** do tipo **addr**.

Caso você necessite apenas uma variável com forma de estrutura, o nome da estrutura não será necessário. Significa que

```
struct
{
char name[30];
char street[40];
char city[20];
char state[2];
unsigned long int zip;
} ainfo;
```

declara uma variável, chamada **ainfo**, da estrutura que a precede.

A forma geral de uma definição de estrutura é

```
struct nome_estrutura{
        tipo nome_variável;
        tipo nome_variável;
        tipo nome_variável;

        .
} variáveis_estrutura;
```

onde nome_estrutura ou variável_estrutura pode ser omitido.

REFERENCIANDO OS ELEMENTOS DE UMA ESTRUTURA

O código que segue estabelece o código CEP 12345 à variável de estrutura **ainfo** anteriormente declarada.

```
ainfo.zip = 12345;
```

Como se vê, o nome da variável de estrutura **ainfo** seguida por um ponto e o nome do elemento referencia aquele elemento de estrutura específico. O ponto freqüentemente é chamado operador-de-ponto, *pelos programadores C*, mas em verdade significa que segue um *elemento de estrutura*. Todos os elementos de estrutura são acessados da mesma forma. A forma geral é

```
nome_estrutura.nome_elemento
```

Portanto, para imprimir o código do CEP na tela, você poderia escrever

```
printf("%d\n",ainfo.zip);
```

Isso vai imprimir o código do CEP contido na variável **zip** da variável de estrutura **ainfo**.

Por exemplo, considere **ainfo.name**. Esse elemento é um vetor de caracteres. Utilizando **gets( )** para inserir um nome, você poderia escrever

```
gets(ainfo.name);
```

Isso passa um ponteiro de caractere para o início de **name**.

Se você quisesse acessar os elementos individuais de **ainfo.name**, poderia indexar **name**. Por exemplo, você poderia imprimir os conteúdos de **ainfo.name**, utilizando

```
register int t;
for(t = 0;ainfo.name[t];++t) putchar(ainfo.name[t]);
```

VETORES DE ESTRUTURAS

Talvez a utilização mais comum de estruturas seja em *vetores de estruturas*. Para declarar um vetor de estruturas, você deve primeiro definir a estrutura e em seguida declarar uma variável de vetor daquele tipo. Por exemplo, para declarar um vetor de estrutura de 100 elementos, do tipo **addr** que foi definida anteriormente neste capítulo, você escreveria

```
struct addr ainfo[100];
```

Isso gera 100 conjuntos de variáveis organizados conforme definido na estrutura **addr**.

Para imprimir o código **CEP** da estrutura 3, você escreveria

```
printf("%d\n",ainfo[2].zip);
```

Como todas as variáveis de vetor, o vetor de estruturas inicia sua indexação em zero.

EXEMPLO DE UMA LISTAGEM DE ENDEREÇAMENTO POSTAL

Nesta seção, será desenvolvido um programa de Lista de Endereçamento Postal que utiliza um vetor de estruturas para guardar as informações de endereço. As funções deste programa interagem com as estruturas e seus elementos, para ilustrar a utilização de estrutura.

Neste exemplo, a informação que será armazenada inclui

- nome
- rua
- cidade
- estado
- código CEP

Para definir a estrutura de dados básica, **addr**, que guardará esta informação, você escreveria

```
struct addr
{
char name[30];
char street[40];
char city[20];
char state[3];
unsigned long int zip;
};
```

Observe que o campo do código CEP é um inteiro longo sem sinal. Isso é feito porque códigos CEP maiores do que 64000 — como 94564 — não podem ser representados em um inteiro de dois bytes. Neste exemplo, um inteiro é utilizado para manter o código CEP, ilustrando um elemento

de estrutura numérico; porém, a prática mais comum é utilizar uma série de caracteres que acomoda os códigos CEP com letras, bem como com números.

Uma vez que a estrutura de dados tenha sido definida você poderá declarar um vetor de estruturas com a seguinte declaração:

```
struct addr ainfo[100];
```

Isto declara um vetor denominado *ainfo*, que contém 100 estruturas do tipo **addr**.

A primeira função necessária pelo programa é **main( )**:

```
main() /* lista de enderecos simples
          exemplo empregando estruturas */
{
char s[80],choice;

init_list();  /* inicializa vetor de estruturas */
do
  {
  choice = menu_select();
  switch(choice)
    {
    case 1 : enter();
             break;
    case 2 : delete();
             break;
    case 3 : list();
             break;
    case 4 : exit(0);
    }
  } while(1);
}
```

Aqui a função **init_list( )** prepara o vetor de estrutura para ser utilizado, colocando um caractere nulo no primeiro byte do campo "nome". O programa assume que se o campo "nome" estiver vazio, a variável de estrutura não está em uso. A função **init_list( )** é escrita como

```
init_list(){
register int t;

for(t = 0;t < 100;++t) ainfo[t].name[0]    '\0';
}
```

Em seguida a função **menu_select( )** vai apresentar as opções e devolver a opção do usuário:

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

printf("1.Entra um nome\n");
printf("2.Retira um nome\n");
printf("3.Lista o arquivo\n");
printf("4.Fim\n");
do
  {
  printf("\nEntre sua opcao: ");
  gets(s);
  fflush(stdin);
  c = atoi(s);
  } while(c < 0 || c > 4);
return c;
}
```

A função **enter( )** solicita ao usuário que digite informação, e coloca esta informação na próxima estrutura livre. Caso o vetor esteja cheio, será impresso na tela a mensagem **lista cheia**. A função **find_free( )** procura, no vetor de estrutura, por um elemento não utilizado. As duas funções são escritas como segue

```
enter()
{
int slot;

slot = find_free();
if(slot == -1)
  {
  printf("\nlista cheia");
  return;
  }
printf("entre nome: ");
gets(ainfo[slot].name);
printf("entre endereco: ");
gets(ainfo[slot].street);
printf("entre cidade: ");
gets(ainfo[slot].city);
printf("entre estado: ");
```

```
gets(ainfo[slot].state);
printf("entre codigo postal: ");
scanf("%d",&ainfo[slot].zip);
fflush(stdin);
}

find_free()
{
register int t;

for(t = 0;ainfo[t].name[0] && t < 100;++t) ;
if(t == 100) return -1;  /* sem entradas livres */
return t;
}
```

Note que **find_free( )** devolverá −1 se todas as variáveis do vetor da estrutura estiverem em uso. Esse número não apresenta problemas em sua utilização, pois não pode existir um elemento −1.

A função **delete( )** apenas pede ao usuário para especificar o número da casa que deve ser apagado. Em seguida, a função coloca um caractere nulo na posição do primeiro caractere do campo **name**.

```
delete()
{
register int slot;
char s[80];

printf("entre numero da entrada: ");
gets(s);
slot = atoi(s);
if(slot >= 0 && slot < 100) ainfo[slot].name[0] = '\0';
}
```

A última função que o programa necessita é **list( )**, que imprime toda a lista de endereçamento na tela:

```
list()
{
register int t;

for(t = 0;t < 100;++t)
  {
  if(ainfo[t].name[0])
    {
```

```
    printf("%s\n",ainfo[t].name);
    printf("%s\n",ainfo[t].street);
    printf("%s\n",ainfo[t].city);
    printf("%s\n",ainfo[t].state);
    printf("%u\n",ainfo[t].zip);
    }
  }
printf("\n\n");
}
```

A Figura 9.1 fornece a listagem completa do Programa de Endereçamento. Compile e estude o programa de maneira a compreender totalmente as referências de estrutura.

```
/*
** Programa simples de correspondencia,
** exemplificando estruturas
*/
#include <stdio.h>

struct addr
{
        char     name[30];
        char     street[40];
        char     city[20];
        char     state[3];
unsigned long int zip;
} ;
struct addr ainfo[100];

main()
{
char s[80],choice;

init_list();  /* inicializa vetor de estruturas */
do
  {
  choice = menu_select();
  switch(choice)
    {
    case 1 : enter();
             break;
```

Figura 9.1 Um programa de endereçamento postal

```
      case 2 : delete();
               break;
      case 3 : list();
               break;
      case 4 : exit(0);
      }
    } while(1);
}

init_list()
{
register int t;

for(t = 0;t < 100;++t) ainfo[t].name[0] = '\0';
}

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

printf("1.Entra um nome\n");
printf("2.Retira um nome\n");
printf("3.Lista o arquivo\n");
printf("4.Fim\n");
do
   {
   printf("\nEntre sua opcao: ");
   gets(s);
   fflush(stdin);
   c = atoi(s);
   } while(c < 0 || c > 4);
return c;
}

enter()
{
int slot;

slot = find_free();
```

Figura 9.1 *(continuação)*

```
if(slot == -1)
  {
  printf("\nlista cheia");
  return;
  }
printf("entre nome: ");
gets(ainfo[slot].name);
printf("entre endereco: ");
gets(ainfo[slot].street);
printf("entre cidade: ");
gets(ainfo[slot].city);
printf("entre estado: ");
gets(ainfo[slot].state);
printf("entre codigo postal: ");
scanf("%d",&ainfo[slot].zip);
fflush(stdin);
}

find_free()
{
register int t;

for(t = 0;ainfo[t].name[0] && t < 100;++t) ;
if(t == 100) return -1;  /* sem entradas livres */
return t;
}

delete()
{
register int  slot;
         char s[80];

printf("entre numero da entrada: ");
gets(s);
slot = atoi(s);
if(slot >= 0 && slot < 100) ainfo[slot].name[0] = '\0';
}

list()
{
register int t;
```

Figura 9.1 *(continuação)*

```
for(t = 0;t < 100;++t)
  {
  if(ainfo[t].name[0])
    {
    printf("%s\n",ainfo[t].name);
    printf("%s\n",ainfo[t].street);
    printf("%s\n",ainfo[t].city);
    printf("%s\n",ainfo[t].state);
    printf("%u\n",ainfo[t].zip);
    }
  }
printf("\n\n");
}
```

Figura 9.1 *(continuação)*

PASSANDO ESTRUTURAS PARA FUNÇÕES

Até aqui todas as estruturas e vetores de estruturas utilizados nos exemplos eram globais. Agora você vai aprender como passar um elemento de estrutura para uma função. Quando uma estrutura é passada para uma função, ocorrem diversas mudanças na maneira pela qual um elemento de estrutura é referenciado.

PASSANDO ELEMENTOS DE ESTRUTURA PARA FUNÇÕES

Quando você passa um elemento de uma variável de estrutura para uma função, em verdade está passando o valor daquele elemento para a função. Portanto, você está passando uma variável simples. (Ao menos, é claro, que aquele elemento seja complexo, como um vetor de caracteres.) Por exemplo,

```
struct fred
{
char  x;
int   y;
float z;
char  s[10];
} mike;
```

Aqui temos exemplos de cada elemento sendo passado para uma função:

```
func(mike.x);      /* passa o caractere x */
func2(mike.y);     /* passa o inteiro y */
func3(mike.z);     /* passa o ponto flutuante z */
func4(mike.s);     /* passa o endereco do vetor s */
func(mike.s[2]); /* passa o caractere s[2] */
```

Entretanto, se você quisesse passar o endereço de elementos de estrutura individuais, colocaria o operador & antes do nome da estrutura. Por exemplo, para passar o endereço dos elementos na estrutura **mike**, você escreveria

```
func(&mike.x);      /* passa o endereco do caractere x */
func2(&mike.y);     /* passa o endereco do inteiro y */
func3(&mike.z);     /* passa o endereco do ponto
                       flutuante z */
func4(mike.s);      /* passa o endereco do vetor s */
func(&mike.s[2]); /* passa o endereco do caractere
                       s[2] */
```

Observe que o operador & precede o nome da estrutura, não o nome do elemento individual.

PASSANDO ESTRUTURAS INTEIRAS PARA FUNÇÕES

Quando uma estrutura é passada para uma função, apenas o endereço do primeiro byte da estrutura é passado. Isso se assemelha à maneira pela qual os vetores são passados para funções. Não é viável copiar toda a estrutura cada vez que ela é passada para uma função; portanto, apenas seu endereço é transferido. Pelo fato da função referenciar a estrutura em si e não uma cópia, você poderá modificar os conteúdos dos elementos da estrutura utilizada na chamada.

O conceito geral que existe por detrás da passagem de uma estrutura para uma função é a passagem de um endereço. Significa que você estará operando com um *ponteiro* de uma estrutura da mesma maneira que trabalha com um ponteiro de um vetor. Por exemplo, a Figura 9.2 fornece um programa simples que imprime horas, minutos e segundos em sua tela. O tempo real deste programa é ajustado variando a contagem da malha **delay( )**.

Como pode ser visto na Figura 9.2, foi definida uma estrutura global, chamada **tm**, sem declaração de variável. Dentro de **main( )**, a estrutura **time** foi declarada e inicializada em 0:0:0. Isso significa que **time** é conhecido apenas para a função **main( )**.

Para as duas funções, **update( )**, que altera o tempo, e **display( )**, que imprime o tempo, são passados o endereço de **time**. Nas duas funções você pode ver que o argumento é declarado como sendo do tipo estrutura **tm**. Isso é necessário para que o compilador possa saber como referenciar os elementos de estrutura.

```
struct tm
{
int hours;
int minutes;
int seconds;
};

main() /* versao 1 - referencias explicitas
                     de ponteiros */
{
struct tm time;

time.hours = 0;
time.minutes = 0;
time.seconds = 0;
do
  {
  update(&time);
  display(&time);
  } while(1);
}

update(t)struct tm *t;
{
(*t).seconds++;
if((*t).seconds == 60)
  {
  (*t).seconds = 0;
  (*t).minutes++;
  }
if((*t).minutes == 60)
  {
  (*t).minutes = 0;
  (*t).hours++;
  }
if((*t).hours == 24) (*t).hours = 0;
delay();
}
display(t)
struct tm *t;
{
```

Figura 9.2 Um programa que imprime horas, minutos e segundos na tela

```
printf("%d:",(*t).hours);
printf("%d:",(*t).minutes);
printf("%d\n",(*t).seconds);
}

delay()
{
int t;

for(t = 1;t < 1000;++t);
}
```

Figura 9.2 *(continuação)*

A referência em si é feita utilizando ponteiros. Por exemplo, se você quisesse retornar as horas para zero, poderia escrever

```
if((*t).hours == 24) (*t).hours = 0;
```

Essa linha de código informa ao compilador para tomar o endereço de **t**, que é **time** em **main( )**, e colocar zero no elemento denominado **hours**. Os parênteses de ***t** são necessários porque o operador de ponto tem precedência sobre *.

Na realidade, porém, dificilmente você verá referências a uma estrutura passada para uma função como no exemplo dado. O motivo é que esse tipo de acesso a elementos de estrutura é tão comum que um operador especial é definido em C para executar essa tarefa. É o –>. A maioria dos programadores o chama *operador-flecha.* É formado utilizando um sinal de menos seguido por um sinal de maior do que. O sinal –> é utilizado em lugar do operador de ponto, ao acessar um elemento de estrutura dentro de uma função. Por exemplo,

```
(*t).hours
```

é o mesmo que

```
t->hours
```

Portanto, **update( )** poderia ser reescrito como

```
update(t)
struct tm *t;
{
```

```
t->seconds++;
if(t->seconds == 60)
   {
   t->seconds = 0;
   t->minutes++;
   }
if(t->minutes == 60)
   {
   t->minutes = 0;
   t->hours++;
   }
if(t->hours == 24) t->hours = 0;
delay();
}
```

Você utiliza o operador ponto para acessar elementos de estrutura, quando a estrutura é global ou definida dentro da mesma função que o código que a referencia. Você utiliza -> para referenciar elementos de estrutura quando um ponteiro de estrutura foi passado para uma função.

Lembre-se também, de que você deve passar o endereço da estrutura para uma função utilizando o operador &. Estruturas *não* são como vetores, que podem ser passados apenas com o nome do vetor.

VETORES E ESTRUTURAS DENTRO DE ESTRUTURAS

Um elemento de estrutura pode ser simples ou complexo. Um elemento simples é qualquer um dos tipos de dados inerentes, como um inteiro ou um caractere. Você já viu um elemento complexo: o vetor de caracteres utilizado em **ainfo**. Outros tipos de dados complexos são vetores uni e multidimensionais dos outros tipos de dados e estruturas.

Um elemento de estrutura què é um vetor é tratado da maneira esperada com referência aos exemplos anteriores. Por exemplo, nesta listagem

```
struct x
{
int   a[10][10]; /* vetor de 10 x 10 inteiros */
float b;
} y;
```

para referenciar o inteiro 3,7 em a da estrutura y, você escreveria

```
y.a[3][7]
```

Uma estrutura poderá ser elemento de outra estrutura, como em

```
struct ralph
{
struct addr newaddr[2]; /* vetor de estruturas
                           tipo addr */
char ch;
} tom;
```

onde **addr** é a estrutura definida anteriormente. Aqui uma estrutura **ralph** foi definida como tendo dois elementos. O primeiro elemento é um vetor de estruturas do tipo **addr**; o segundo é um caractere, **ch**. Este trecho de código estabelece o código CEP 61853 para o segundo elemento de **newaddr**:

```
tom.newaddr[1].zip = 61853;
```

CAMPOS DE BIT

Ao contrário da maioria das outras linguagens de computador, C apresenta um método inerente para acessar um bit dentro de um byte. Isso poderá ser útil por diversos motivos: primeiro, se a memória é limitada, você pode armazenar diversas variáveis *Booleanas* (verdade/falso) em um byte; segundo, certas interfaces de dispositivos transmitem informação codificada em bits de um byte; e em terceiro lugar, certas rotinas de criptografia precisam acessar os bits de um byte. Apesar de todas estas funções poderem ser executadas utilizando bytes e operadores bit-a-bit, um campo-de-bit poderá acrescentar mais estrutura ao seu código e torná-lo mais portátil.

O método que C utiliza para acessar bits é baseado na estrutura. Esta estrutura define três variáveis de um bit cada:

```
struct device
{
unsigned active    : 1;
unsigned ready     : 1;
unsigned xmt_error : 1;
} dev_code;
```

Todas as três variáveis são declaradas **unsigned**, porque um bit não pode ter sinal. De fato, os únicos valores que um bit pode ter são 0 e 1. A variável de estrutura **dev_code** pode ser utilizada para decodificar informação da porta de um acionador de fita, por exemplo. A codificação que segue vai escrever um byte de informação na fita e verificar a existência de erros utilizando **dev_code**:

```
wr_tape(c)
char c;
{
while(!dev_code.ready) rd(&dev_code); /* espere */
wr_to_tape(c); /* escreva o byte */
while(dev_code.active) rd(&dev_code); /* espere ate
                        informacao ser escrita */
if(dev_code.xmt_error) printf("erro de escrita\n");
}
```

Aqui, **rd( )** vai retornar o estado do acionador de fita e **wr_to_tape( )** escreve os dados. A Figura 9.3 mostra como que a variável de campo-de-bit **dev_code** se apresenta na memória.

Como pôde ser visto no exemplo anterior, cada campo de bit é acessado utilizando o operador ponto de estrutura. Porém, caso a estrutura seja passada para uma função, você deve utilizar o operador –>.

Você não precisa dar nome para cada campo de bit. Isso facilita o encontro do bit desejado. Por exemplo, se o acionador de fita devolvesse também uma indicação de fim-de-fita no bit 5, você poderia alterar a estrutura **device** para permitir essa acomodação, utilizando

```
struct device
{
unsigned active     : 1;
unsigned ready      : 1;
unsigned xmt_error  : 1;
unsigned            : 1;
unsigned            : 1;
unsigned EOT        : 1;
} dev_code;
```

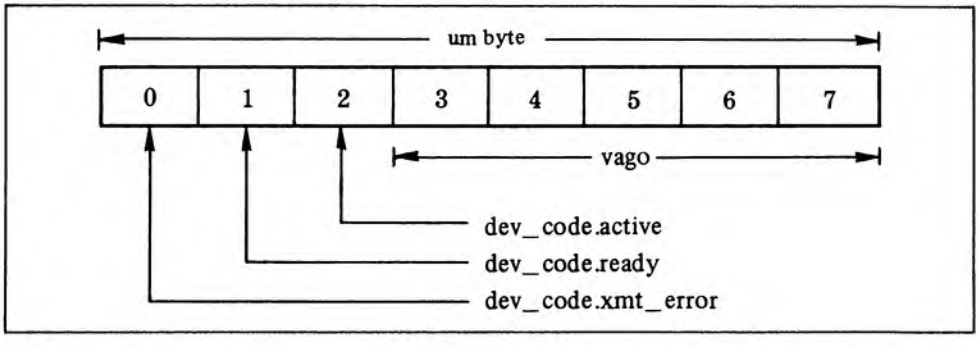

Figura 9.3 A variável campo-de-bit **dev_code em memória**

Variáveis de campo-de-bit apresentam certas restrições. Você não pode pegar o endereço de uma variável de campo-de-bit. As variáveis campo-de-bit não podem ser agrupadas em vetores. Você não pode ultrapassar o tamanho de um int, assim como não sabe, entre diversas máquinas, se os campos são da direita para a esquerda ou da esquerda para a direita; significa que qualquer código que utiliza campos-de-bit poderá ser dependente da máquina.

UTILIZANDO UMA UNIÃO

Em C uma **união** é um local da memória utilizado por diversas variáveis diferentes de tipos potencialmente diferentes. Aqui temos a definição de uma unidade, chamada **u**, de um caractere e de um inteiro:

```
union u
{
int  i;
char ch;
};
```

Assim como com as estruturas, esta definição não declara nenhuma variável. Você pode declarar uma variável colocando seu nome no fim da definição ou utilizando uma declaração em separado. Para declarar uma variável de **união cnvt** do tipo **u**, utilizando a definição dada há pouco, você escreveria

```
union u cnvt;
```

Em **union cnvt**, tanto o inteiro **i** como o caractere **ch** compartilham o mesmo espaço de memória. (Evidentemente, **i** ocupa dois bytes e **ch** ocupa apenas um.) A Figura 9.4 mostra como é que **i** e **ch** compartilham o mesmo endereço.

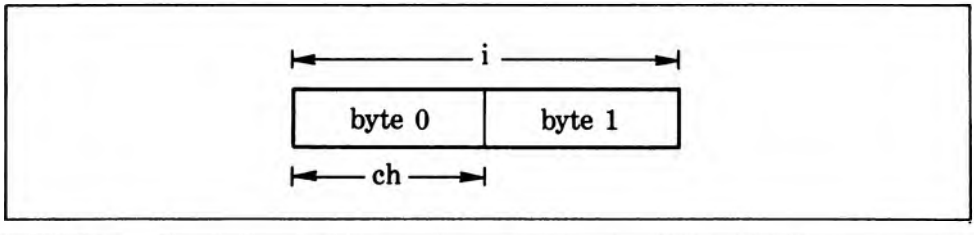

Figura 9.4 O local de memória ocupado pelo inteiro **i** e caractere **ch**

Quando uma **união** é declarada, o compilador automaticamente gera uma variável com o tamanho suficiente para manter o maior tipo de variável da **união**.

Para acessar o elemento de uma **união**, você utiliza a mesma sintaxe que utilizaria para as estruturas: os operadores de ponto e de flecha. Caso a variável **union** seja global, você utilizará o operador de ponto; caso a variável **union** seja passada para uma função, você utilizará o operador flecha. Por exemplo, caso **union cvnt** seja global, para atribuir o inteiro 10 ao seu elemento **i**, você escreverá

```
cnvt.i = 10;
```

Entretanto, caso **cnvt** seja passada para uma função, você deve utilizar o operador –>:

```
func1(un)
union u *un;
{
un->i = 10; /* atribui o valor 10 a cnvt usando funcao*/
}
```

A utilização de **union** poderá ajudar na produção de um código independente da máquina ou portátil. Pelo fato do compilador controlar todos os tamanhos, nenhuma dependência de máquina é produzida. Você não precisa se preocupar com o tamanho de um inteiro, caractere ou **float**. Por exemplo, você pode utilizar uma **union** em conversões de tipo. A função **putw( )** discutida nos capítulos anteriores vai escrever a representação binária de um inteiro para um arquivo em disco. Você já viu duas maneiras de escrever a função: a primeira com vetores e a segunda com ponteiros. Os dois métodos utilizam o fato de que qualquer endereço *pode* ser atribuído para uma variável ponteiro. Uma outra maneira, talvez mais elegante, é utilizando **union**. Primeiro você deve criar **union** com um inteiro e um vetor de caracteres de dois bytes:

```
union pw
{
int  i;
char ch[2];
};
```

Agora, **putw( )** pode ser escrito utilizando **union**:

```
putw(word,fp)
union pw *word;
FILE *fp;
{
putc(word->ch[0],fp);
putc(word->ch[1],fp);
}
```

Para escrever uma palavra em um arquivo em disco, você chamaria **putw( )** com o valor do inteiro que você quer escrever.

UTILIZANDO sizeof

Você viu que tanto as estruturas como as **uniões** podem ser utilizadas para criar variáveis de grandes dimensões, e que o tamanho dessas variáveis podem mudar de máquina para máquina. O operador unário **sizeof** poderá ser utilizado para informar-lhe o tamanho de qualquer tipo de variável, incluindo estruturas, **uniões** e variáveis definidas pelo usuário. Isso poderá ajudar na eliminação de códigos dependentes-da-máquina de seus programas.

Por exemplo, estes são os tamanhos dos tipos de dados para uma implementação C comum a muitos compiladores C de microcomputadores:

Tipo	Tamanho em bytes
char	1
int	2
long int	4
float	8
double	16

Portanto, esta listagem

```
char ch;
int  i;
float f;

printf("%d",sizeof(ch));
printf("%d",sizeof(i));
printf("%d",sizeof(f));
```

vai imprimir os números 1, 2 e 8 na tela.

sizeof é um *operador do momento-de-compilação:* toda a informação necessária para computar o tamanho de qualquer variável é conhecida no momento da compilação. Por exemplo, considere o seguinte código:

```
union x
{
char  ch;
int   i;
float f;
} tom;
```

sizeof(tom) será 8. No momento do processamento, não importa o que realmente está contido em **union tom**; tudo o que importa é o tamanho da maior variável que ele pode manter, porque **union** deverá ter o tamanho de seu maior elemento.

UTILIZANDO typedef

C permite que você defina novos nomes de tipos de dados, utilizando explicitamente a palavra-chave **typedef**. Você não está realmente *criando* uma nova classe de dado, mas sim definindo um novo nome para um tipo de dado. Esse processo pode ajudar a fazer com que os programas dependentes da máquina fiquem mais portáteis; unicamente as declarações **typedef** deveriam ser alteradas. Pode ainda ajudar na documentação de seu código, permitindo nomes descritivos para os tipos de dados padrão. A forma geral da declaração **typedef** é

```
typedef tipo nome;
```

onde **type** é qualquer tipo de dado permissível, e **name** é o novo nome para este tipo. O novo nome que você define é em adição ao – não uma substituição do – tipo de nome existente.

Por exemplo, você poderia criar um novo nome para **float** utilizando

```
typedef float balance;
```

Esta declaração diz ao compilador para que reconheça **balance** como outro nome de **float**. Em seguida você poderia criar uma variável **float**, utilizando **balance**:

```
balance over_due;
```

Onde **over_due** é uma variável de ponto flutuante do tipo **balance**, que é outra palavra para **float**.

Você pode utilizar **typedef** para criar nomes para tipos mais complexos também; por exemplo,

```
typedef struct client
{
float due;
int   over_due;
char  name[40];
};

client clist[NUM_CLIENTS]; /* define um vetor de
                      estruturas do tipo client */
```

A utilização de **typedef** poderá ajudá-lo a tornar seu código mais legível e mais fácil de ser transferido para outra máquina. Mas lembre-se, você *não* está criando outro tipo de dado.

EXERCÍCIOS

1. Defina uma estrutura chamada **player** que seja capaz de armazenar a seguinte informação:

   nome do jogador
   nome do time
   média de pontos

2. Utilizando **player** do exercício anterior, declare um vetor de estruturas de 100 elementos. Chame este vetor **p_info**.
3. Escreva uma função curta denominada **enter( )**, que introduza o nome do jogador, o nome do time e a média de pontos, a partir do teclado. Sua função terá o índice do vetor como argumento único. Assuma que **p_info** seja global.
4. Reescreva **enter( )** do exercício anterior, assumindo que **p_info** é passado para a função junto com o índice do vetor.
5. Escreva uma estrutura denominada **crypt** que apresente as seguintes variáveis de campo-de-bit:

   bit 0: **first**
   bit 1: **second**
   bit 4: **check_bit**

6. Defina uma **union** das seguintes variáveis:

   int i[4];
   char ch[8];
   float f;

7. O que há de errado com o seguinte código?

```
if(sizeof(u) == 2) printf("u possui um inteiro\n");
else printf("u possui um caractere\n");
```

8. Escreva a declaração **typedef** que fará o nome **width** o mesmo que **float**.

RESPOSTAS

1.
```
struct player
{
char  player_name[40];
char  player_team[40];
float batting_average;
};
```

2.
```
struct player p_info[100];
```

3.
```
enter(rec)
int rec;
{
printf("entre nome do jogador :");
gets(p_info[rec].player_name);
printf("entre tempo do jogador:");
gets(p_info[rec].player_team);
printf("entre com a media do jogador:");
scanf("%f",p_info.batting_average):
}
```

4.
```
enter(rec,pi)
int rec;
struct player *pi[];
{
printf("entre nome do jogador:");
gets(pi[rec]->player_name);
printf("entre tempo do jogador:");
gets(pi[rec]->player_team);
printf("entre com a media do jogador:");
scanf("%f",pi->batting_average);
}
```

5.
```
struct crypt
{
unsigned first      : 1;
unsigned second     : 1;
unsigned            : 2;
unsigned check_bit  : 1;
};
```

6.
```
union u
{
int   i[4];
char  ch[8];
float f;
};
```

7. sizeof é um operador durante o tempo de compilação e nada tem a ver com o tipo de variável que possa estar armazenado em algum local.

8.
```
typedef float width;
```

CAPÍTULO **10**

ESCREVENDO UM PROGRAMA C

Neste capítulo vamos desenvolver um programa Lista de Endereçamento Postal mais complexo, que enfatiza procedimentos de projeto e estilos e ilustra muito da linguagem C. Este exemplo vai lhe mostrar os diversos passos necessários para que se escreva um programa C bem-sucedido.

Existem três maneiras possíveis de se escrever um programa: *de-cima-para-baixo* (top-down), *de-baixo-para-cima* (bottom-up) e *ad-hoc*. No processo *top-down* você inicia com a rotina de nível mais elevado e se dirige às rotinas de nível inferior. O processo *bottom-up* opera em sentido inverso: você inicia com determinadas rotinas e as constrói progressivamente em estruturas mais complexas, terminando na rotina mais elevada. O processo *ad-hoc* não apresenta um método predeterminado. O procedimento *top-down* é normalmente considerado como sendo o melhor e será o único método utilizado aqui.

PROGRAMAÇÃO top-down

C como uma linguagem estruturada permite seguir o método top-down. Esse método poderá fornecer um código limpo, legível e de fácil manutenção; poderá ainda ajudá-lo a esclarecer a estrutura geral e a operação do programa, antes de codificar as funções de nível inferior. Isso poderá reduzir o tempo decorrente de inícios errados.

ESBOÇANDO O PROGRAMA

O método top-down assemelha-se a um esboço, em que você inicia com a idéia geral e progressivamente define melhor a cada nível subseqüente. Uma boa maneira de iniciar a codificação de qualquer programa é definir exatamente o que o programa vai fazer, no nível mais elevado. A definição da Lista de Endereçamento Postal é

- Coloque um novo nome
- Cancele um nome
- Imprima a lista
- Procure por um nome
- Salve a lista em um arquivo em disco
- Carregue a lista
- Saia do programa

Esses passos podem formar a base para as funções do programa.

Após as definições das funções gerais do programa, você poderá iniciar o esboço dos detalhes de cada área funcional, iniciando com a malha principal. A malha principal deste programa é

```
malha principal
{
        do {
                apresenta menu
                pega seleção do usuário
                processa a seleção
        } enquanto a seleção não for saia
}
```

A utilização de uma anotação de algoritmo desse tipo poderá ajudá-lo a esclarecer a estrutura geral de seu programa antes que você inicie a utilização do computador. Utilizamos a sintaxe C, porque ela agora nos é familiar; porém, qualquer tipo de sintaxe é aceitável.

Você deveria fornecer uma definição similar para cada área funcional. Por exemplo, a função de salvar-no-disco poderia ser definida como

```
salve no disco {
      abrir arquivo de disco
      enquanto houver dados para escrever {
            escreva os dados no disco
      }
      fecha arquivo de disco
}
```

Observe que a definição não menciona a estrutura de dados ou variáveis. É intencional. Neste momento você está interessado apenas em definir o que seu programa vai fazer, e não como vai fazê-lo. Esse processo de definição vai ajudá-lo a decidir como deveria ser a estrutura de dados.

ESCOLHENDO UMA ESTRUTURA DE DADOS

Após ter determinado o esboço geral de seu programa, o próximo passo é decidir como serão armazenados os dados utilizados pelo seu programa. As seleções da estrutura de dados e suas implementações são críticas, porque ajudam a determinar os limites de projeto de seu programa.

Em capítulos anteriores, as listas de informação eram representadas como vetores de tamanho fixo. Poderemos utilizar um vetor de estrutura de tamanho fixo para a Lista de Endereçamento Postal. Entretanto, nesse caso um vetor de tamanho fixo apresenta duas desvantagens. Primeiro, o tamanho do vetor limita o tamanho da listagem postal, pela colocação de um limite arbitrário no programa. Segundo, um vetor de tamanho fixo não aproveita a memória adicional que possa ser acrescentada ao computador, e caso haja eliminação de memória – por exemplo, falha de um circuito impresso – o programa nem poderá operar, porque o vetor de tamanho fixo não combina. Portanto, esse Programa de Endereçamento Postal vai utilizar as funções de alocação dinâmica de memória C, **malloc( )** e **free( )**, que permitem à listagem postal ter o tamanho da memória livre.

Apesar do processo de armazenamento ter sido determinado pela alocação dinâmica, a forma exata dos dados ainda não foi definida. O programa vai utilizar uma estrutura, que conterá todas as informações de endereçamento e dois ponteiros: um ponteiro para a próxima entrada na lista e outro para apontar a entrada anterior da lista. A utilização de dois ponteiros de entrada fará da listagem postal uma *lista ligada dupla*. Significa que cada entrada terá um ponteiro para a entrada que a preceder e para a entrada que a seguir.

A estrutura do endereço é

```
struct addr
{
        char  name[30];
        char  street[40];
        char  city[20];
        char  state[3];
        char  zip[10];    /* guarda codigo postal */
struct addr *next; /* ponteiro p/ proxima entrada */
struct addr *prior;/* ponteiro p/ entrada anterior */
} list_entry;
```

Cada entrada da lista terá a estrutura de **addr**. Os dois ponteiros de **addr** são deste tipo porque estarão apontando para estruturas deste tipo.

Você deve tomar o cuidado de escolher nomes descritivos para os elementos da estrutura, pois se trata da estrutura principal de dados. A utilização de nomes descritivos facilita a identificação dos elementos, à medida que você escreve o programa.

Uma variável de estrutura, **list_entry**, também foi declarada. Essa variável vai ajudá-lo mais tarde a determinar o tamanho da estrutura.

A Figura 10.1 mostra como uma lista ligada dupla com elementos do tipo **addr** aparece na memória. Uma lista ligada dupla apresenta três grandes vantagens. Primeiro, para inserir e retirar elementos em e da lista, tudo que será necessário fazer será comutar os ponteiros. Segundo, a lista pode ser lida de frente para trás e de trás para frente. Terceiro, uma lista ligada dupla é adequada para uma ordenação rápida, porque apenas os ponteiros precisam ser comutados, e não todo o registro.

Uma lista ligada dupla apresenta duas extremidades: o seu começo e o seu fim. O último elemento da lista deverá ter seu ponteiro **next** colocado em nulo, indicando a inexistência de outra entrada. O caso inverso também é verdadeiro; o ponteiro **prior** da primeira entrada também deve ser colocado em zero, para indicar que não existe entrada prévia.

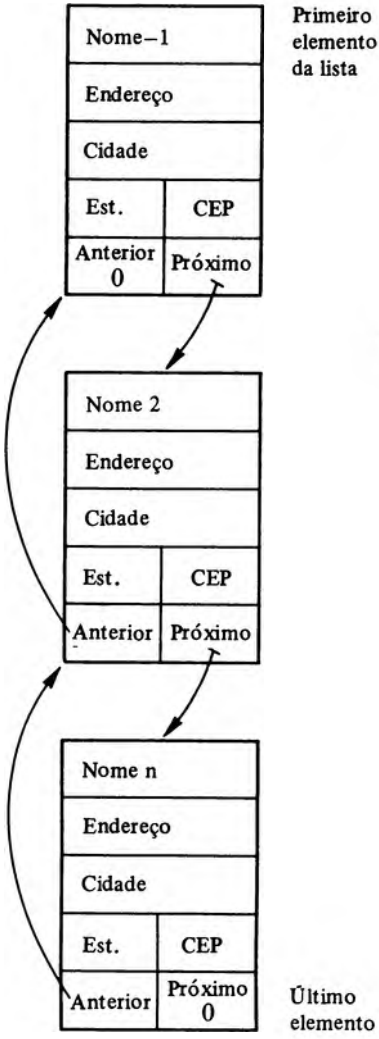

Figura 10.1 Uma lista ligada dupla na memória

Tanto a descrição geral do programa como a estrutura dos dados foi definida. Um bom programador sempre deve repassar esses dois passos com cuidado, antes de iniciar a codificação do programa. Esta é a única maneira em que programas bem escritos, confiáveis e fáceis de se ler podem ser consistentemente desenvolvidos.

DEFININDO AS VARIÁVEIS GLOBAIS E A FUNÇÃO main( )

Você agora está pronto para escrever as variáveis globais e a função **main( )**. Este programa de Endereçamento Postal exige apenas dois globais: o primeiro para conter o ponteiro para a primeira entrada da lista, e o segundo para conter o ponteiro da última entrada da lista. O ponteiro de início-da-lista informa ao programa onde está a primeira entrada. Em seguida, os ponteiros de cada entrada vão encontrar as entradas subseqüentes. O ponteiro de-fim-de-lista ajuda a acelerar toda a rotina de entrada. Caso o programa mantenha controle do fim da lista, a adição de um novo endereço requer apenas que se atualize um ponteiro da última entrada para que aponte para a nova última entrada. Caso contrário, o programa terá de varrer a cadeia completa das entradas para encontrar o fim.

Os dois ponteiros variáveis são

```
struct addr *start; /* ponteiro para a primeira
                       entrada da lista */
struct addr *last;  /* ponteiro para a ultima
                       entrada */
```

A função **main( )** é quase que uma tradução direta do esboço. É bastante fácil para qualquer um que esteja lendo o programa ver exatamente o que o programa faz. A única inicialização necessária é colocar **start** em 0, o que indica uma listagem vazia.

```
main()
{
int choice;

start = 0; /* lista de tamanho zero */
do
  {
  choice = menu_select();
  switch(choice)
    {
    case 1 : enter();  /* entre uma nova entrada */
             break;
    case 2 : delete(); /* retira uma entrada */
             break;
    case 3 : list();   /* mostra a lista */
             break;
```

```
    case 4 : search(); /* procura uma entrada */
             break;
    case 5 : save();   /* salva a lista em disco */
             break;
    case 6 : load();   /* restaura a lista do disco */
             break;
    case 7 : exit(0);
    }
  } while(1);
}
```

Apesar de terem sido escolhidos nomes descritivos para as funções principais **enter( ), delete( ), list( ), search( ), save( )** e **load( )**, o acréscimo de comentários ajuda a esclarecer o que cada caso faz. Alguém lendo o programa pode facilmente encontrar a função correta que deseja examinar em cada área de programa.

A função **menu_select( )** é basicamente a mesma que aquelas utilizadas em outros locais deste livro. Poderia escrevê-la da seguinte maneira:

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

printf("1.Entra um nome\n");
printf("2.Retira um nome\n");
printf("3.Lista o arquivo\n");
printf("4.Procura\n");
printf("5.Salva o arquivo\n");
printf("6.Restaura o arquivo\n");
printf("7.Fim\n");
do
  {
  printf("\nEntre sua opcao: ");
  gets(s);
  c = atoi(s);
  } while(c < 0 || c > 7);
return c;
}
```

Até que uma solicitação válida seja feita, o programa faz as solicitações e percorre a malha.

Ao escrever programas, você deve lembrar-se de que alguém terá de usá-los (inclusive você). É importante pegar as entradas não válidas. Neste caso, é válida apenas uma resposta entre 1 e 7. Isso ajuda a impedir surpresas desagradáveis enquanto o programa estiver em uso.

DEFININDO enter

Pelo fato do programa da Listagem de Endereçamento utilizar alocação dinâmica de memória para permitir o armazenamento das entradas de lista individuais, a rotina **enter( )** deverá primeiro alocar um espaço suficiente de memória para a nova entrada. Este programa vai utilizar a função padrão **malloc( )** para o alocamento de memória. (Alguns compiladores o chamam **alloc( )**; certifique-se.) Você poderia ter contado os bytes para determinar o tamanho da estrutura **addr**, mas C fornece **sizeof**, que faz isso automaticamente. Você necessita de um ponteiro variável para reter o ponteiro entregue por **malloc( )** e deve ser declarado como um ponteiro para **addr**. Aqui está **enter( )**:

```
enter()
{
struct addr *info;
char *malloc();
do
  {
  info = (struct addr *)malloc(sizeof(list_entry));
  if(info == 0)
    {
    printf("\nmemoria esgotada");
    return;
    }
  inputs("entre nome: ",info->name,30);
  if(!info->name[0]) break; /* termina entrada */
  inputs("entre endereco: ",info->street,40);
  inputs("entre cidade: ",info->city,20);
  inputs("entre estado: ",info->state,3);
  inputs("entre codigo postal: ",info->zip,10);
  if(start! = 0)
    { /* se nao for primeira entrada */
    last->next = info;
    info->prior = last;
    last = info;
    last->next = 0;
    }
  else
    { /* e' primeira entrada */
    start = info;
    start->next = 0;
    last = start;
    start->prior = 0;
    }
  } while(1); /* malha de entrada */
```

Sempre que **malloc( )** é utilizado, você deve verificar se existe memória disponível. **malloc( )** devolve um endereço de memória válido, que aponta para o começo do número de bytes solicitados, ou 0, que indica falha na alocação, porque não havia memória livre disponível. Uma falha no teste dessa condição poderá levar a um "crash" catastrófico. Na maioria dos sistemas, se você tentar utilizar a região de memória que inicia em 0, estará escrevendo sobre uma parte do sistema operacional, de seu programa ou de ambos. Tome muito cuidado com isto. Observe também que **malloc( )** é declarado de modo a devolver um ponteiro de caractere. Lembre-se de que C presume que um inteiro será devolvido caso não haja uma declaração expressa em sentido contrário.

Pelo fato de ser muito comum inserir diversos endereços ao mesmo tempo, a função **enter( )** repete a malha até que uma linha em branco seja colocada no campo nome.

Observe a utilização do operador –>. Já que todas as referências à informação de endereço postal são feitas por meio de ponteiros, você terá de utilizar o operador-flecha em lugar do operador-ponto, para o acesso aos campos individuais. Essa necessidade se deve também ao fato de que o programa utiliza armazenamento dinâmico para a lista; armazenamento dinâmico não pode devolver uma estrutura, apenas um endereço de memória. Entretanto, pela utilização de um ponteiro de estrutura, o compilador C pode organizar o bloco de memória recebido de **malloc( )** como se fosse uma variável de estrutura.

O C padrão não apresenta uma função de solicitação de entrada; entretanto, este programa precisa solicitar ao usuário que coloque entradas e impedir ultrapassagem de limites nas variáveis série que formam cada estrutura. Uma função especial denominada **inputs( )** fornece esse serviço. É uma função completamente generalizada, exceto pelo fato de que a linha de entrada não pode exceder 255. A função **inputs( )** é escrita como

```
inputs(prompt,s,count) /* esta funcao ira ler uma
                          serie de caracteres com
                          comprimento em count. Isto
                          previne o estouro da serie,
                          e mostra um sinal de
                          pronto. */
char *prompt;
char *s;
int   count;
{
char p[255];

do
  {
  printf(prompt);
  gets(p);
  if(strlen(p)>=count) printf("\nmuito grande\n");
  } while(strlen(p) >= count);
strcpy(s,p);
}
```

Observe que a série de entrada não é colocada diretamente no argumento, mas é copiada apenas quando o programa determina que serve. Essa função não é completamente à prova de erros, porque uma entrada maior que 255 cobrirá a variável local s. Nos exercícios, você será convidado a escrever uma versão melhor de **inputs( )**, reescrevendo **gets( )**, a função C padrão.

DEFININDO delete( )

A função **delete( )** é utilizada para remover uma entrada da lista postal. Para remover uma entrada, você terá de saber seu nome exato. **delete( )** vai procurar na lista até encontrar uma coincidência ou alcançar o fim da lista. Caso encontre uma coincidência, a entrada é cancelada e os ponteiros das entradas anterior e posterior são atualizados de maneira a "contornarem" a entrada cancelada. O ponteiro da região de memória da entrada cancelada passa para **free( )**, a função C padrão, que devolve essa parte não utilizada da memória ao sistema, para utilização posterior. A função **delete( )** é escrita como segue

```
delete()
{
struct addr *info, *find();
       char  s[255];
inputs("entre nome: ",s,30);
info = find(s);
if(info)
  {
  if(start == info)
    {
    start = info->next;
    start->prior = 0;
    }
  else
    {
    info->prior->next = info->next;
    if(info != last) info->next->prior = info->prior;
    else last = info->prior;
    }
  free(info); /* retorna memoria ao sistema */
  }
}
```

Observe que **inputs( )** foi utilizado para inserir o nome que será cancelado. Esta é a vantagem de uma função generalizada – pode ser utilizada repetidas vezes. Observe também que **find( )** é declarada para devolver um ponteiro para uma estrutura do tipo **addr**. Preste atenção especial ao código de rearranjo do ponteiro. A Figura 10.2 mostra como os ponteiros são alterados após uma eliminação.

Considere esta declaração:

```
info->prior->next = info->next;
```

Uma vez que o operador-seta avalia da esquerda para a direita, a expressão é a mesma que

```
(info->prior)->next = info_next;
```

Isso significa que você estará atribuindo o valor do ponteiro **next** da entrada cancelada ao ponteiro **next** da entrada anterior.

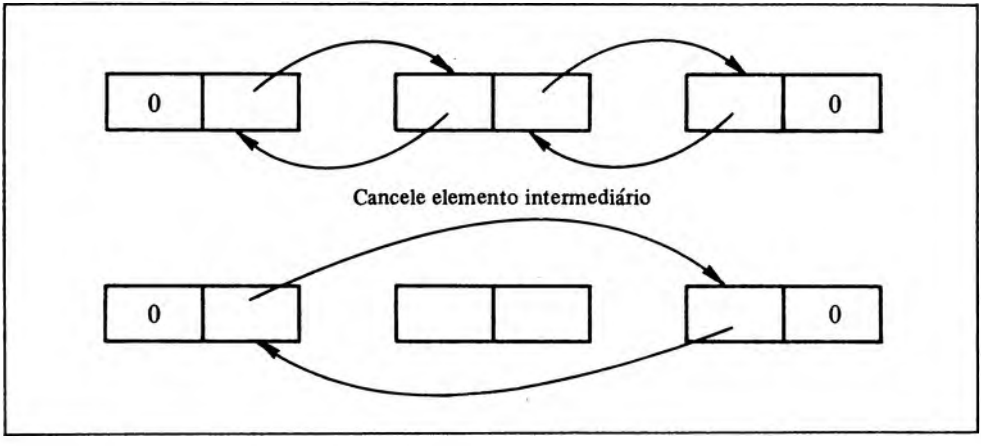

Figura 10.2 O ponteiro muda após a deleção

A função **find( )** é uma procura linear da lista. Começa no início da lista e pára no fim ou em uma entrada que coincida com o nome da entrada. O fim da lista é reconhecido quando o ponteiro **next** é 0. A função **find( )** é escrita da seguinte maneira

```
struct addr *find(name)
char *name;
{
struct addr *info;

info = start;
while(info)
  {
```

```
  if(!strcmp(name,info->name)) return info;
  info = info->next; /* pegue proxima entrada */
  }
printf("nome nao encontrado\n");
return 0; /* nao encontrado */
}
```

Caso o nome não seja encontrado, a função devolverá 0, porque uma região de memória validada nunca terá o endereço 0.

IMPRIMINDO A LISTAGEM POSTAL

A maioria das listagens postais gera etiquetas de endereçamento. Pelo fato do método de acesso exato à impressora poder variar de acordo com o sistema operacional e o compilador C que estiverem sendo utilizados, este programa simplesmente imprimirá a listagem postal na tela de seu computador. Como em um dos exercícios do capítulo, você é incentivado a modificar esta rotina para imprimir em sua impressora.

É necessária uma função **display( )** genérica, porque tanto a função **list( )** como a função **search( )** necessitam dela para apresentar informação na tela. A função **display( )** é utilizada para encurtar o código em **list( )** e torná-la mais fácil de ser lida:

```
list()
{
register int   t;
struct   addr *info;

info = start;
while(info)
  {
  display(info);
  info = info->next;  /* pegue proxima entrada */
  }
printf("\n\n");
}

display(info)
struct addr *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");
}
```

ENCONTRANDO UM NOME NA LISTAGEM POSTAL

Uma listagem postal normalmente é utilizada para localizar o endereço de alguém. A função **search( )** possibilita isso. Para utilizar **search( )**, você insere o nome da pessoa que deseja encontrar. Em **search( )**, **inputs( )** é utilizado para digitar o nome:

```
search()
{
        char  name[30];
struct addr *info,*find();
inputs("entre nome a procurar: ",name,30);
if(!(info = find(name))) printf("nao encontrado\n");
else display(info);
}
```

SALVANDO E CARREGANDO A LISTA

Para salvar a listagem postal em disco, necessitamos da função **save( )**. Nesta versão simples de **save( )**, o nome **mlist** é codificado permanentemente como o nome-do-arquivo.

```
save()
{
register int   t,size;
struct   addr *info;
         char *p;
         FILE *fp;
if((fp = fopen("mlist","w")) == 0)
  {
  printf("nao pode abrir arquivo\n");
  exit(0);
  }
printf("\nsalvando arquivo\n");
size = sizeof(list_entry);
info = start;
while(info)
  {
  p = (char *)info;  /* convertendo para ponteiro de
  caractere */
  for(t = 0;t < size;++t) putc(*p++,fp);  /* salva
  um byte */
  info = info->next; /* pegue proximo endereco */
  }
putc(EOF,fp);  /* envia um EOF explicito */
fclose(fp);
}
```

A função começa no topo da lista e vai até o fim, escrevendo todos bytes em cada estrutura, utilizando **putc( )**, a função padrão da biblioteca C. Ao ponteiro de caracteres **p** é atribuída a posição inicial de cada estrutura, e **sizeof(list_entry)** controla o número de bytes que devem ser escritos. Apesar de a maior parte da informação do arquivo ser possível de se imprimir, os ponteiros não o serão, de modo que você não deve tentar imprimir este arquivo em sua tela ou impressora.

A função **load( )** é um pouco mais complicada. Ela tem que montar a lista e posicionar todos ponteiros à medida que lê os dados.

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

if((fp = fopen("mlist","r")) == 0)
  {
  printf("nao pode abrir arquivo\n");
  exit(0);
  }
printf("\ncarregando arquivo\n");
size = sizeof(list_entry);
start = (struct addr *)malloc(size);
if(!start)
  {
  printf("memoria esgotada\n");
  return;
  }
info = start;
p = (char *)info;  /* converte para ponteiro de
                      caractere */
while((*p++ = getc(fp)) != EOF)
  {
  for(t = 0;t < size - 1;++t) *p++ = getc(fp);
      /* carrega byte a byte */
  info->next = (struct addr *)malloc(size);
  /* pegue mais memoria */
  if(!info->next)
    {
    printf("memoria esgotada\n");
    return;
    }
  info->prior = temp;
  temp = info;
```

```
    info = info->next;
    p = (char *)info;
    }
free(temp->next);   /* devolve memoria nao usada */
temp->next = 0;     /* ultima entrada */
last = temp;
start->prior = 0;
fclose(fp);
}
```

LIMITAÇÕES

Foram projetadas diversas limitações neste programa de Listagem Postal. Algumas tornam o exemplo simples, enquanto outras foram decisões específicas de projeto. Um bom programador deverá estar ciente das duas limitações bem como das possibilidades do programa.

Uma limitação evidente está na própria estrutura de dados. Existe espaço apenas para um nome e uma rua. Isso está ótimo para a maioria dos endereços residenciais, mas com freqüência uma empresa apresenta dois nomes e dois endereços. Uma segunda linha de nome permitiria também "em atenção à". Pelo fato de tudo no programa ser dinâmico e o programa ter utilizado **sizeof** quando o tamanho da estrutura foi necessário, tudo o que é preciso para acrescentar mais espaço é acrescentar campos à estrutura de dados e para as funções **enter( )** e **display( )**.

Uma pergunta feita com freqüência a respeito deste tipo de programa é "Quantas entradas podem ser armazenadas?" Uma vez que o programa utiliza armazenamento dinâmico, não existe uma resposta absoluta. O número de entradas está limitado pela quantidade de memória livre do sistema. Você poderia ter uma certa estimativa do número de entradas, dividindo o tamanho de **addr** pela memória livre. Mas lembre-se de que outras funções do programa – inclusive as funções de armazenamento em disco – também exigem memória.

Atualmente o programa não apresenta meios de classificar a listagem postal. Nesse aspecto, o projeto do programa poderá não atender às necessidades da maioria dos usuários. Porém, esta limitação poderá não ter importância para aqueles que não querem uma lista classificada. É importante saber para quem você está escrevendo um programa. Mesmo que seja você o usuário final, seu programa deverá satisfazer suas necessidades também.

PROBLEMAS COM O PROGRAMA DE LISTAGEM POSTAL

Caso você insira e compile todas as funções listadas neste capítulo, terá uma Listagem de Endereçamento Postal que funciona conforme descrito. Vai funcionar. Falta-lhe elasticidade contra certos tipos de falhas de hardware e apresenta pelo menos um defeito.

O problema da flexibilidade tem a ver com a maneira com que os ponteiros são rearranjados em **delete( )**. Todos os programadores devem reconhecer o fato de que o hardware falha de tempos em tempos. Caso ocorra um erro de memória no meio do processo de cancelamento, é possível perdermos todas as entradas que seguem a entrada cancelada. O ponto cancelado é desconectado

da cadeia. Porém, um erro de memória faz com que o ponteiro **next** perca um bit, fazendo com que aponte para um local desconhecido da memória. Este ponteiro inválido agora é colocado no ponteiro **next** da entrada precedente. Isso interromperá a cadeia. Apesar de ser impossível eliminar completamente este tipo de problema, colocando um controle redundante na codificação, poderia contornar a maioria desses erros e informar ao usuário a existência de um problema.

É de responsabilidade do programador fazer com que o hardware execute da melhor forma possível, mesmo que isso signifique antecipar problemas potenciais de hardware.

O problema existente no programa foi deixado intencionalmente. Examine com cuidado a rotina **load( )**. O que vai acontecer com chamadas repetidas a essa função? Tente encontrar o problema antes de continuar com a leitura.

A resposta consiste em que eventualmente a memória seja totalmente utilizada porque não existe nenhuma liberação antes do carregamento. Para cada carregamento existe nova alocação de memória; caso você carregasse a mesma lista duas vezes, estaria alocando uma nova região de memória para cada carregamento. A solução seria escrever uma função denominada **free_list( )** que liberasse a memória para qualquer lista atualmente em uso. Isso é matéria para um exercício.

O PROGRAMA DE LISTAGEM POSTAL COMPLETO

O programa completo da Listagem Postal está relacionado na Figura 10.3.

```
struct addr
{
        char  name[30];
        char  street[40];       char  city[20];
        char  state[3];
        char  zip[10];    /* guarda codigo postal */
struct addr *next;  /* ponteiro p/ proxima entrada */
struct addr *prior; /* ponteiro p/ entrada anterior */
} list_entry;
struct addr *start; /* ponteiro para a primeira
                       entrada da lista */
struct addr *last;  /* ponteiro para a ultima
                       entrada */

main()
{
int choice;
```

Figura 10.3 Programa completo da listagem postal

```
start = 0; /* lista de tamanho zero */
do
  {
  choice = menu_select();
  switch(choice)
    {
    case 1 : enter();  /* entre uma nova entrada */
             break;
    case 2 : delete(); /* retira uma entrada */
             break;
    case 3 : list();   /* mostra a lista */
             break;
    case 4 : search(); /* procura uma entrada */
             break;
    case 5 : save();   /* salva a lista em disco */
             break;
    case 6 : load();   /* restaura a lista do disco */
             break;
    case 7 : exit(0);
    }
  } while(1);
}

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

printf("1.Entra um nome\n");
printf("2.Retira um nome\n");
printf("3.Lista o arquivo\n");
printf("4.Procura\n");
printf("5.Salva o arquivo\n");
printf("6.Restaura o arquivo\n");
printf("7.Fim\n");
do
  {
  printf("\nEntre sua opcao: ");
  gets(s);
  c = atoi(s);
  } while(c < 0 || c > 7);
```

Figura 10.3 *(continuação)*

```
return c;
}

enter()
{
struct addr *info;

char *malloc();
do
   {
   info = (struct addr *)malloc(sizeof(list_entry));
   if(info == 0)
     {
     printf("\nmemoria esgotada");
     return;
     }
   inputs("entre nome: ",info->name,30);
   if(!info->name[0]) break; /* termina entrada */
   inputs("entre endereco: ",info->street,40);
   inputs("entre cidade: ",info->city,20);
   inputs("entre estado: ",info->state,3);
   inputs("entre codigo postal: ",info->zip,10);
   if(start! = 0)
     { /* se nao for primeira entrada */
     last->next = info;
     info->prior = last;
     last = info;
     last->next = 0;
     }
   else
     { /* e' primeira entrada */
     start = info;
     start->next = 0;
     last = start;
     start->prior = 0;
     }
   } while(1); /* malha de entrada */
}

inputs(prompt,s,count) /* esta funcao ira ler uma
                          serie de caracteres com
                          comprimento em count.
```

Figura 10.3 *(continuação)*

```
                                Isto previne o estouro da
                                serie, e mostra um sinal
                                de pronto. */
char *prompt;
char *s;
int   count;
{
char p[255];

do
  {
  printf(prompt);
  gets(p);
  if(strlen(p) >= count) printf("\nmuito grande\n");
  } while(strlen(p) >= count);
strcpy(s,p);
}

delete()
{
struct addr *info, *find();
       char  s[255];

inputs("entre nome: ",s,30);
info = find(s);
if(info)
  {
  if(start == info)
    {
    start = info->next;
    start->prior = 0;
    }
  else
    {
    info->prior->next = info->next;
    if(info != last) info->next->prior = info->prior;
    else last = info->prior;
    }
  free(info); /* retorna memoria ao sistema */
  }
}
```

Figura 10.3 *(continuação)*

```
struct addr *find(name)
char *name;
{
struct addr *info;

info = start;
while(info)
  {
  if(!strcmp(name,info->name)) return info;
  info = info->next; /* pegue proxima entrada */
  }
printf("nome nao encontrado\n");
return 0; /* nao encontrado */
}

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

info = start;
while(info)
  {
  display(info);
  info = info->next;  /* pegue proxima entrada */
  }
printf("\n\n");
}

display(info)
struct addr *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()
```

Figura 10.3 *(continuação)*

```
{
        char  name[30];
struct addr *info,*find();

inputs("entre nome a procurar: ",name,30);
if(!(info = find(name))) printf("nao encontrado\n");
else display(info);
}

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

if((fp = fopen("mlist","w")) == 0)
  {
  printf("nao pode abrir arquivo\n");
  exit(0);
  }
printf("\nsalvando arquivo\n");
size = sizeof(list_entry);
info = start;
while(info)
  {
  p = (char *)info;  /* convertendo para ponteiro
                        de caractere */
  for(t = 0;t < size;++t) putc(*p++,fp);
                                /* salva um byte */
  info = info->next; /* pegue proximo endereco */
  }
putc(EOF,fp);  /* envia um EOF explicito */
fclose(fp);
}

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

Figura 10.3 *(continuação)*

```
if((fp = fopen("mlist","r")) == 0)
  {
  printf("nao pode abrir arquivo\n");
  exit(0);
  }
printf("\ncarregando arquivo\n");
size = sizeof(list_entry);
start = (struct addr *)malloc(size);
if(!start)
  {
  printf("memoria esgotada\n");
  return;
  }
info = start;
p = (char *)info;  /* converte para ponteiro
                      de caractere */
while((*p++ = getc(fp)) != EOF)
  {  for(t = 0;t < size - 1;++t) *p++ = getc(fp);
                          /* carrega byte a byte */
  info->next = (struct addr *)malloc(size);
                       /* pegue mais memoria */
  if(!info->next)
    {
    printf("memoria esgotada\n");
    return;
    }
  info->prior = temp;
  temp = info;
  info = info->next;
  p = (char *)info;
  }
free(temp->next);  /* devolve memoria nao usada */
temp->next = 0;      /* ultima entrada */
last = temp;
start->prior = 0;
fclose(fp);
}
```

Figura 10.3 *(continuação)*

EXERCÍCIOS

1. Compile e processe o programa de Listagem Postal descrito neste capítulo.
2. Reescreva a estrutura **addr** de modo que possa acomodar dois nomes de campo.
3. Reescreva **list( )** para seu computador, de maneira que imprima os endereços em sua impressora.
4. Escreva a função **free_list( )**, que vai devolver, toda a memória atualmente alocada, ao sistema.

RESPOSTAS

2.
```
struct addr
{
        char  name[30];
        char  name2[30];
        char  street[40];
        char  city[20];
        char  state[3];
        char  zip[10];    /* guarda codigo postal */
struct addr *next;        /* ponteiro para proxima
                             entrada */
struct addr *prior;       /* ponteiro para entrada
                             anterior */
} list_entry;
```

4.
```
free_list()
{
struct addr *info,*temp;

into = start;
while(info)
  {
  temp = info->next;
  free(info);
  info = temp;
  }
}
```

CAPÍTULO 11

ERROS DE PROGRAMAÇÃO COMUNS

Apesar de ser difícil encontrar um capítulo devotado aos erros mais comuns de programação, em um livro que ensina uma linguagem de programação, este tipo de capítulo é necessário neste livro, porque C apresenta determinadas qualidades que podem gerar erros normalmente não encontrados em outras linguagens de programação.

C é bem robusto: apresenta poucos relatórios para verificação de erro durante o processamento, definidos diretamente na linguagem. Ao contrário do BASIC, que, quando a solicitação de um disco falha, emite uma mensagem dizendo **não posso abrir o arquivo**, C nada fará a não ser que você tenha feito uma solicitação específica de verificação em seu programa.

TIPOS DE ERRO

Geralmente, quando um programa não processa corretamente, existem algumas razões possíveis, das quais as três a seguir são as mais freqüentes:

- Falha de hardware
- Erro nos dados de entrada pelo usuário
- Erros de programação

Como programador, você deve determinar onde o erro está e em seguida corrigi-lo. Destas três categorias possíveis de erro, erros de entrada e de programação estão diretamente sob seu controle; caso haja falha de hardware, você será responsável apenas pela demonstração conclusiva da falha para que possa ser eliminada. Seus programas devem pegar os erros de entrada cometidos pelo usuário, antes que criem problemas. Finalmente, você é diretamente responsável pela criação de uma codificação sem erro de sintaxe, implementação ou projeto.

Você deverá ter a experiência necessária para corrigir todos os erros de sintaxe relatados pelo compilador. Mas ter um programa sem erros de sintaxe é apenas o primeiro passo. Seu programa deverá fazer aquilo que você pensa que ele vai fazer. No restante deste capítulo vamos discutir os erros que podem aparecer em seu código e alguns erros de sintaxe pouco usuais.

ERROS DE ORDEM DE PROCESSAMENTO

Os operadores de decremento e incremento são utilizados na maioria dos programas escritos em C. Você deve lembrar, no entanto, que a ordem em que a operação é feita depende se esses operadores precedem ou seguem a variável. Por exemplo, estas duas declarações:

```
y = 10;
x = y++;
```

e

```
y = 10;
x = ++y;
```

não são as mesmas. Na primeira o valor 10 é atribuído a **x** e em seguida incrementa **y**. A segunda incrementa **y** para 11 e em seguida atribui o valor 11 a **x**. Portanto, no primeiro caso **x** contém 10; no segundo, **x** contém 11.

As operações de incremento ou decremento ocorrem antes de outras operações, se precederem o operando; caso contrário, ocorrem depois.

Esse tipo de confusão pode ser difícil de detectar. Poderá haver indicações como malhas que não se processam corretamente ou rotinas deslocadas em 1. Assegure-se de que você está entendendo os operadores de incremento e decremento.

PROBLEMAS COM PONTEIROS

Um erro comum em C é o emprego indevido dos ponteiros. Os problemas com ponteiros caem em duas categorias: o primeiro é um engano de indireção e operadores ponteiros; o segundo é uma utilização acidental de um ponteiro inválido. A solução do primeiro problema é uma revisão da linguagem C, assegurando-se da compreensão de como construiu o programa; a solução ao segundo problema é verificar sempre a validade de um ponteiro antes de utilizá-lo.

A Figura 11.1 mostra um programa com erros comuns em ponteiros, feitos pelos iniciantes em programação C. Este programa sem dúvida vai "quebrar", provavelmente levando com ele o sistema operacional. A razão pela qual não opera é que o endereço devolvido por **malloc( )** foi atribuído "*não para* **p**", mas para a posição de memória apontada por **p**, que é completamente desconhecida neste caso. Isso certamente não é o que se deseja. Para corrigir este programa, você deve substituir

```
p = malloc(100); /* esta correto */
```

pela linha que está errada.

O programa tem um segundo erro, mais insidioso. Não há nenhuma verificação durante o processamento do endereço devolvido por **malloc( )**. Lembre-se de que se houve esgotamento de memória, **malloc( )** retorna 0, que nunca é um ponteiro válido em C. O funcionamento errado devido a esse tipo de problema é difícil de ser encontrado, porque ocorre muito raramente, quando uma solicitação de alocação falha. A melhor maneira para tratar isto é impedir seu acontecimento. A Figura 11.2 fornece uma versão correta do programa, que inclui uma verificação da validade do ponteiro.

```
main() /* este programa esta errado */
{
char *p;
char *malloc();

*p = malloc(100); /* esta errado */
gets(p);
printf(p);
}
```

Figura 11.1 Um programa mostrando erros típicos de ponteiros

O terrível a respeito de ponteiros "perdidos" é que são difíceis de serem localizados. Caso você esteja fazendo atribuições para uma variável apontadora que não contenha um endereço de ponteiro válido, seu programa poderá parecer estar funcionando corretamente uma parte do tempo e falhando em outra parte. Quanto menor o programa, maior a certeza de operar corretamente, mesmo com um ponteiro errante, porque pouca memória é utilizada. À medida que seu programa cresce, as falhas se tornam mais comuns, mas você pode pensar que o erro está na nova parte do programa ou nas correções feitas, e não nos ponteiros. Assim, a tendência é olhar no lugar errado, à procura do erro.

Um sinal de que está havendo problemas com ponteiros é que os erros tendem a ser erráticos. Seu programa opera corretamente um instante e erradamente no instante seguinte. Algumas vezes outras variáveis contêm "lixo" sem explicação razoável. Caso esse tipo de problema comece a ocorrer, verifique seus ponteiros. Como norma de procedimento, certifique-se de conferir todos os ponteiros quando os erros começam a acontecer.

Lembre-se, entretanto, de que apesar dos ponteiros poderem causar problemas, são também um dos aspectos mais poderosos e úteis da linguagem C, compensando qualquer problema que possam causar. Esforce-se para aprender a utilizá-los corretamente.

Uma observação final a respeito dos ponteiros: você deve inicializá-los antes que sejam utilizados. Isso pode parecer muito simples de ser lembrado, mas muitos programadores C de categoria ainda esquecem algumas vezes. Por exemplo, este trecho

```
int *x;
*x=100;
```

provoca um desastre, porque você não sabe para onde x está apontando. A atribuição de um valor para aquela posição desconhecida, provavelmente está destruindo algo de valor – como outro código ou dado de seu programa.

```
main() /* este programa esta correto */
{
char *p;
char *malloc();

p = malloc(100); /* esta errado */
if(p == 0)
  {
  printf("sem memoria\n");
  exit(0);
  }
gets(p);
printf(p);
}
```

Figura 11.2 Versão correta do programa da Figura 11.1

REDEFININDO FUNÇÕES

Você pode, mas não deve, chamar suas funções pelo mesmo nome daquelas da biblioteca padrão de C. A maioria dos compiladores vai utilizar sua função sobre aquela da biblioteca. Isso poderá causar problemas diretos e indiretos.

Por exemplo, os fragmentos de programa mostram um problema causado diretamente pela redefinição de uma função de biblioteca:

```
main( )
{
   FILE *fp
   char big[1000];
   init_ array(big);
   if((fp= open("name","r"))==-1) {
      printf("não posso abrir arquivo \n");
      exit(0);
   }
   .
   .
   .
}
```

```
init_array(s)
char *s;
{
   register in t;
   for(t= 0;t<1000; t++,p ++) {
      *p= t;
      if(t% 100) open(p);
   }
}
open(p)
char *p;
{
   *p='O';
}
```

A função padrão **open( )** foi redefinida no programa para atribuir o caractere **0** para certos elementos de vetor. Nada tem a ver com o **open( )**, utilizado para abrir um arquivo em disco chamado em **main( )**. A utilização de **open( )** dessa maneira fará com que o programa "quebre" ou faça coisas estranhas.

Um problema ainda pior ocorre quando uma função de biblioteca padrão é redefinida, mas a função padrão é utilizada indiretamente por outra função padrão. Por exemplo,

```
char text[1000];
main( )
{
   int x;
   scanf("%d",&x);
   .
   .
   .
}
getc(p) / *retorna caractere do vetor *
{
   return text[p];
}
.
.
.
```

Este programa não vai operar com a maioria dos compiladores porque **scanf( )**, uma função C padrão, provavelmente vai chamar **getc( )**, outra função padrão de C, que foi redefinida no programa. Este poderá ser um problema frustrante porque não haverá nenhum indício de que você criou um problema adicional. Apenas parecerá que **scanf( )** não está operando corretamente.

A única maneira de evitar esses problemas é nunca dar para uma função que você esteja escrevendo o mesmo nome que um existente na biblioteca padrão do C. Caso você não tenha certeza, coloque suas iniciais no início do nome, como **hs_getc( )** em lugar de **getc( )**.

ERROS DE SINTAXE BIZARROS

Vez ou outra, o compilador C poderá relatar erro de sintaxe que você não reconhece como sendo um erro, porque o compilador C poderá apresentar um problema que provoque a informação de erros falsos. A única solução é reprojetar sua codificação. Outros erros incomuns apenas exigem algum retrocesso de sua parte.

Um erro particularmente confuso ocorrerá quando você tentar compilar um código contendo este fragmento:

```
main( )
{
   char *p, *myfunc( ); /* myfunc( ) returns
                           char pointer */



}
myfunc( )
{



}
```

A maioria dos compiladores emite uma mensagem de erro como **function redefined** e aponta para **myfunc( )**. Como pode ser? A codificação não tem duas **myfunc( )**. A resposta é que você declarou **myfunc( )** para devolver um ponteiro de caractere dentro de **main( )**. Isso provocou uma entrada na tabela de símbolos com essa informação: quando o compilador, posteriormente, encontrou **myfunc( )** no programa, não havia indicação de que deveria devolver outra coisa que não fosse um inteiro, do tipo "default". Assim, você estava "redefinindo" a função. A correção seria:

```
main( )
{
   char *p, *myfunc( ); /* myfunc( ) returns
                           char pointer */



}
char *myfunc( )
{



}
```

A Figura 11.3 lista um programa com outro erro de sintaxe que é difícil de entender. O erro neste programa é o ponto-e-vírgula após a declaração de **func1( )**. O compilador verá isso como uma declaração externa a qualquer função, o que é um erro. Entretanto, a maneira pela qual os diversos compiladores relatam esse erro difere. Muitos compiladores emitirão uma mensagem de erro do tipo **bad declaration syntax**, enquanto apontarem para o primeiro parêntese aberto após **func1( )**. Pelo fato de você estar acostumado a ver ponto-e-vírgula após as declarações, poderá ser difícil localizar a origem do erro.

```
main() /* este programa possui um erro de sintaxe */
{
func1();
}

func1();
{
printf("Este e' func1 \n");
}
```

ERROS DE INDEXAÇÃO

Como você já deve estar sabendo, todos os índices de C iniciam com 0. Um erro comum envolve a utilização de uma malha **for** para acessar os elementos de um vetor. A Figura 11.4 fornece um programa que supomos inicializar um vetor de 100 inteiros. A malha **for** deste programa está errada por dois motivos: primeiro, ele não vai inicializar **num[0]**, o primeiro elemento do vetor **num**; em segundo lugar a malha vai até 100, um após o último elemento do vetor, **num[99]**. A maneira correta de se escrever esse programa é demonstrada na Figura 11.5. Lembre-se, um vetor de 100 elementos varia de 0 até 99.

```
main() /* este programa nao ira trabalhar */
{
int x,num[100];

for(x = 1;x <= 100;++x) num[x] = x;
}
```

Figura 11.4 Um programa que utiliza erradamente uma malha **for** para inicializar um vetor

```
main() /* este esta certo */
{
int x,num[100];

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

Figura 11.5 A versão correta do programa da Figura 11.4

ERROS DE LIMITE

C e muitas funções de biblioteca padronizadas têm pouca ou nenhuma verificação de limite durante o processamento. Por exemplo, é possível escrever sobre vetores, arquivos de disco e com atribuição de ponteiros, sobre qualquer variável. Esses erros normalmente não ocorrem, mas quando o fazem, ligar o sintoma com a causa poderá ser muito difícil.

Por exemplo, o programa da Figura 11.6 deverá ler uma série do teclado e apresentá-la na tela. Neste caso não existem propriamente erros de codificação. Indiretamente, porém, chamar **get_string( )** com s provoca um erro. A variável s é declarada como tendo 10 caracteres de comprimento, mas **get_string( )** lerá 20 caracteres. Isso fará com que s seja sobreposto. O problema reside no fato de que enquanto s apresenta corretamente os 20 caracteres, **var1** ou **var2** não conterá o valor certo.

A razão disso está na maneira com que o compilador utiliza a memória. Todos os compiladores C devem alocar uma região de memória para variáveis locais. Esta normalmente é a região da pilha. As variáveis **var1**, s e **var2** estarão localizadas na memória conforme indicado na Figura 11.7. Seu compilador C poderá alterar a ordem de **var1** e **var2**, mas continuarão envolvendo s. Quando s é coberto, a informação adicional é colocada na área que se supõe esteja reservada para **var2**, destruindo qualquer conteúdo anteriormente existente. Assim, em lugar de imprimir o número 10 para as duas variáveis inteiras, aquela destruída pela sobreposição de s vai apresentar outra coisa. Ao ver isso você poderá iniciar a procura do erro em outro local.

```
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("digite vinte caracteres\n");
for(t = 0;t < 20;++t) *string++ = getchar();
}
```

Figura 11.6 Um programa que cobre uma variável

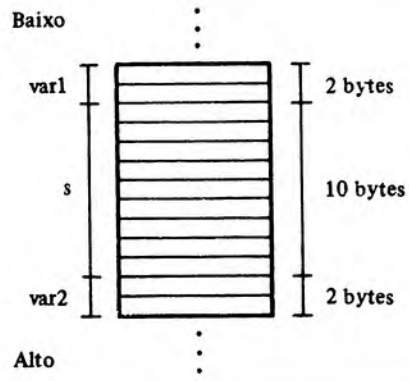

Figura 11.7 As variáveis locais **var1** e **var2** em uma pilha

OMISSÕES DA DECLARAÇÃO DE UMA FUNÇÃO

Toda vez que uma função devolver um tipo de valor diferente de um inteiro, ela deverá ser declarada dentro de cada função que a utiliza. Um exemplo do que ocorre quando falta a declaração de uma função é o programa da Figura 11.8, que multiplica dois números em ponto flutuante entre si.

Neste programa, **main( )** vai esperar que um valor inteiro seja devolvido por **mul( )**, mas **mul( )** devolve um número em ponto flutuante. Você vai receber resultados sem significado porque **main( )** copiará apenas dois bytes dos oito necessários por um flutuante.

```
main() /* esta errado */
{
float x,y;

scanf("%f%f",&x,&y);
printf("%f\n",mul(x,y));
}

float mul(a,b)
float a,b;
{
return a * b;
}
```

Figura 11.8 Um programa em que falta uma declaração de função

A maneira de corrigir este programa é declarar **mul( )** em **main( )**, conforme indica a Figura 11.9. A função **mul( )** foi acrescentada à lista de declaração **float**, que informa **main( )** para aguardar a devolução de um valor em ponto flutuante de **mul( )**.

```
main() /* esta correto */
{
float x,y,mul();

scanf("%f%f",&x,&y);
printf("%f\n",mul(x,y));
}

float mul(a,b)
float a,b;
{
return a * b;
}
```

Figura 11.9 A versão corrigida do programa da Figura 11.8

ERROS DO ARGUMENTO DE CHAMADA

Você deve combinar todo tipo de argumento que uma função espera com o tipo que você lhe dá. Um exemplo importante é **scanf( )**. Lembre-se de que **scanf( )** espera receber o *endereço* de seu argumento, e não o seu valor. Isso significa que você deve chamar **scanf( )** com argumentos, utilizando o operador & . Por exemplo,

```
int x;
char string[10];
scanf("%d%s",x,string);
```

está errado, enquanto

```
scanf("%d%s",&x,string);
```

está correto. Lembre-se, porém, de que as séries já passam os seus endereços para as funções, de modo que você não deve utilizar o operador & neles.

Outro erro comum é esquecer que, a não ser que seja explicitado de outra maneira, as funções C não podem modificar os seus argumentos. Caso seja necessário modificar o argumento de uma função, você deve passar o endereço do argumento para a função e utilizar referências de ponteiro para acesso.

Caso os parâmetros formais de uma função sejam do tipo **float**, você deve passar variáveis em ponto flutuante para a função. Por exemplo, no programa da Figura 11.10, você não pode utilizar uma função em ponto flutuante como **div( )** para devolver um valor inteiro. Ainda, **div( )** não vai operar corretamente, porque está esperando números em ponto flutuante e não inteiros.

```
main() /* este programa esta incorreto */
{
int x,y;

scanf("%d%d",x,y,);
printf("%d\n",div(x,y));
}

float div(a,b)
float a,b;
{
return a / b;
}
```

Figura 11.10 Um programa que mostra um erro de argumento de chamada

TESTE INCREMENTAL

Todos têm uma abordagem diferente para programação e depuração. Entretanto, certas técnicas, com o tempo, foram provadas como sendo melhores do que outras. Em caso de depuração, *teste incremental* é considerado como o método de menor custo, apesar de parecer à primeira vista atrasar o processo de desenvolvimento.

O teste incremental é simplesmente o processo de sempre ter código de trabalho. Assim que for possível processar uma parte de seu programa, você deverá fazê-lo, para testar esta seção completamente. À medida que você faz acréscimos ao programa, continue a testar as novas seções, analisando a maneira como eles se ligam às seções testadas anteriormente. Assim, você estará concentrando qualquer problema possível em uma pequena área da codificação.

A teoria do teste incremental é geralmente baseada em probabilidade e áreas. Como você sabe, a área é uma dimensão quadrática. Cada vez que você acrescenta comprimento, dobra a área. Portanto, à medida que seu programa cresce, existe uma área elevada a N que você deve analisar. Como um programador, você quer trabalhar com a menor área possível enquanto faz a depuração. Por intermédio de teste incremental, você tem condição de subtrair a área já testada da área total, reduzindo assim a região em que foi encontrado um problema.

Uma última observação: a perfeição é conseguida apenas como um acidente; portanto, os bons programadores são também bons em depuração.

EXERCÍCIOS

1. O que há de errado com o seguinte trecho de código?

```
/* atribui numeros 0 ate 9 a num */
int t,num[10];

t = 0;
do num[t] = ++t; while(t < 10);
```

2. O que há de errado com este programa?

```
main()
{
char *c;
int t;

c = &t;
*c = 10;
}
```

3. Porque este programa não opera? Mostre como corrigi-lo.

```
main() /* troque dois inteiros */
{
int a,b;

a = 10;
b = 20;
swap(a,b);
}

swap(x,y)
int x,y;
{
int temp;

temp = x;
x = y;
y = temp;
}
```

4. Este programa tem fim?

```
main()
{
int t;

for(t = 0;t != 10;++t)
   {
   printf("%d\n",t);
   if(t == 10) t = 0;
   }
}
```

5. O que há de errado neste trecho de código?

```
int x;
scanf("%f",x);
```

RESPOSTAS

1. Porque t é incrementado antes de ser designado, os números de 1 até 10 serão colocados no vetor **num**. A linha de atribuição deveria ser

```
num[t] = t++;
```

2. O programa está tentando fazer uma atribuição, utilizando indireção, um valor inteiro pela utilização de um ponteiro de caractere. Isso poderá ser desejado sob circunstâncias muito especiais, mas da maneira em que este programa se encontra, é considerado um erro.
3. C utiliza chamada por valor, o que significa que os argumentos das funções não podem ser alterados por aquelas funções. Para corrigir este programa, tanto a chamada para **swap( )** como o próprio **swap( )** deverão ser alterados para aceitar ponteiros. Quando corrigido o programa será

```
main() /* troque dois inteiros */
{
int a,b;

a = 10;
b = 20;
swap(&a,&b);
}
```

```
swap(x,y)
int *x,*y;
{
int temp;

temp = *x;
*x = *y;
*y = temp;
}
```

4. Não.

5. Duas coisas estão erradas: primeiro, **x** é um inteiro e **scanf( )** está sendo informado para ler um número em ponto flutuante; segundo, **scanf( )** deverá ser chamado com o *endereço* da variável, e não seu valor.

APÊNDICE A

UM RESUMO DE C

C tem 28 palavras-chaves. Estas palavras, combinadas com a sintaxe formal de C, formam a linguagem de programação C. Aqui temos uma relação de todas as palavras-chaves de C:

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

RESUMO DAS DECLARAÇÕES

Esta seção resume muitas declarações C comuns e fornece exemplos de como podem ser utilizadas.

Break

Um **break** é utilizado para sair de uma malha **do, for** ou **while**, contornando a condição normal da malha. É utilizado também para sair de uma declaração **switch.**

Este é o exemplo do **break** em uma malha:

```
while(x < 100)
  {
  x = get_new_x();
```

```
if(keystroke()) break; /* tecla pressionada */
process(x);
}
```

Caso uma tecla seja pressionada, a malha termina, não importa qual seja o valor de **x**.

Em uma declaração **switch**, **break** realmente evita que a execução do programa passe para o próximo **case**. (Veja a declaração **switch** para explicação detalhada.)

Continue

Continue é utilizado para contornar partes do código em uma malha e força o teste condicional a ser feito. Por exemplo, esta malha **while** lerá caracteres do teclado até que s seja pressionado:

```
while(ch = getchar())
  {
  if(ch != 's') continue; /* leia outro caractere */
  process(ch);
  }
```

A chamada para **process( )** não vai ocorrer até que **ch** contenha o caractere s

Do

A malha **do** é uma das três construções de malha disponíveis em C. A forma geral da malha **do** é

```
do {
    statemente block
} while(condition);
```

Caso apenas uma declaração seja repetida, a chave não será necessária, mas acrescentará clareza à declaração.

A malha **do** é a única de C que terá sempre ao menos uma iteração, porque a condição é testada no final da malha.

Uma utilização bastante comum da malha **do** é a leitura de arquivos em disco. Este código lerá um arquivo até que seja encontrada uma **EOF**.

```
do
  {
  ch = getc(fp);
  store(ch);
  } while(ch != EOF);
```

For

O laço **for** permite uma inicialização e incremento automáticos de uma variável contadora. A forma geral é

```
for(inicialização; condição; incremento){
   bloco de declaração
}
```

Caso o bloco de declaração seja uma declaração simples, as chaves não serão necessárias.

Apesar do **for** permitir um número de variações, a inicialização normalmente é utilizada para posicionar o valor inicial de uma variável contadora. A condição normalmente é uma declaração relacional que compara a variável contadora com um valor final, e o incremento incrementa (ou decrementa) o valor do contador.

Esta codificação imprime a mensagem **hello** dez vezes.

```
for(t = 0;t < 10;++t) printf("hello\n");
```

A próxima codificação aguarda o acionamento de uma tecla após imprimir **hello**.

```
for(t = 0;t < 10;t++)
  {
  printf("hello\n");
  getchar();
  }
```

Goto

Goto faz com que a execução do programa salte para o rótulo especificado na declaração **goto**. A forma geral de **goto** é

```
        goto rótulo;
        .
        .
        .
rótulo:
```

Todos os rótulos devem terminar em dois pontos e não devem entrar em conflito com palavras-chaves ou nomes de funções.

Por exemplo, este código vai imprimir a mensagem **right**, mas não a mensagem **wrong**.

```
goto lab1;
        printf("errado");
lab1:   printf("certo");
```

If e Else

A forma geral da declaração **if** é

```
if(condição) {
      bloco de declaração 1
}
else {
      bloco de declaração 2
}
```

Caso sejam utilizadas declarações simples, as chaves não serão necessárias. O **else** é opcional.

A condição pode ser qualquer condição. Caso esta expressão faça uma avaliação diferente de 0, o bloco de declaração 1 será executado; caso contrário, se existir, o bloco de declaração 2 será executado.

Este fragmento pode ser utilizado como entrada do teclado e procurar **q**, cujo significado é "sair":

```
ch = getchar();
if(ch == 'q')
  {
  printf("programa terminado");
  exit(0);
  }
else proceed();
```

Return

A declaração **return** força o retorno de uma função e pode ser utilizada para devolver um valor a uma rotina chamadora. Por exemplo, a função abaixo devolve o produto de seus dois argumentos inteiros:

```
mul(a,b)
int a,b;
{
return (a * b);
}
```

Lembre-se de que sempre que se encontra um **return**, a função retorna, pulando qualquer outro código na função.

Switch

A declaração **switch** é a declaração de ramificação multidirecional de C: é utilizada para direcionar a execução em uma ou mais direções. A forma geral dessa declaração é

```
switch(variável) {
     case (constante1): conjunto de declarações 1;
                        break;
     case (constante2): conjunto de declarações 1;
                        break;
     .
     .
     .
     case (constante n): conjunto de declarações n;
                         break;
     default: declaração default
}
```

Cada conjunto de declarações poderá ser formado por uma ou mais declarações. A parte **default** é opcional.

swich opera confrontando a variável com todas as constantes. Assim que uma concordância é encontrada, o conjunto de instruções é executado. Você pode pensar nos **cases** como um rótulo.

Se a declaração **break** for omitida, a execução continuará até que seja encontrada uma declaração **break** ou até que termine o **switch**.

O exemplo que segue poderá ser utilizado para o processamento de uma seleção de menu:

```
ch = getchar();
switch(ch)
  {
  case 'e' : enter();
             break;
  case 'l' : list();
             break;
  case 's' : sort();
             break;
  case 'q' : exit(0);
  default  : printf("comando desconhecido\n");
             printf("tente novamente\n");
}
```

While

A malha **while** apresenta a forma geral

```
while(condição) {
      bloco de declaração
}
```

Caso o objeto de **while** seja uma declaração simples, as chaves poderão ser omitidas.

while testa a condição no início da malha. Portanto, se a condição for falsa, a malha poderá não ser executada. A condição poderá ser uma expressão.

Neste exemplo do **while**, 100 caracteres serão lidos de um arquivo em disco e armazenados em um vetor de caracteres:

```
t=0;
while  (t<100) {
       s[t] =getc(fp);
       t++;
}
```

TIPOS DE DADOS

C apresenta os seguintes dados inerentes:

Tipo	Palavra-chave C
character	char
integer	int
long integer	long int
short integer	short int
unsigned integer	unsigned int
floating point	float
double floating point	double

Além dos tipos inerentes, você pode gerar combinações com eles, utilizando **struct** e **union**. Você pode ainda criar novos nomes para tipos de variáveis utilizando **typedef**.

A forma geral da declaração **struct** é

```
struct nome_struct {
     elemento 1;
     elemento 2;
     .
     .
     .
} variável_struct;
```

A forma geral de **union** é

```
union nome_união {
    elemento 1;
    elemento 2;
    .
    .
    .
} variável_união;
```

Você pode declarar as variáveis como sendo estáticas, fazendo com que permaneçam existindo durante toda execução do programa.

O modificador **register** poderá ser utilizado em inteiros e caracteres. Faz com que as variáveis especificadas sejam armazenadas em um registrador da UCP em vez de em uma posição da memória. Isso acelera o acesso a essas variáveis.

Caso **extern** seja colocado antes de um nome de variável, o compilador saberá que aquela variável deverá ser declarada em outro local. A utilização mais comum disso é quando você tem dois ou mais arquivos, compartilhando as mesmas variáveis globais.

APÊNDICE B

FUNÇÕES DE BIBLIOTECA PADRÕES DE C

As funções descritas neste apêndice serão encontradas nas bibliotecas-padrão da maioria dos compiladores C. As descrições fornecidas aqui servem de guia geral de sua utilização; mas, consulte seu manual de usuário para os detalhes exatos.

atoi(p)
char *p;

atoi( ) converte uma série de dígitos em seu valor inteiro. A função **atoi( )** tem um argumento ponteiro de caracteres simples que aponta para a série de dígitos. O valor inteiro daquele argumento é devolvido. Caso a série passada para **atoi( )** não contenha um número inteiro válido, será devolvido um 0. Espaços iniciais e tabulações são geralmente ignorados e poderá ser utilizado um sinal de menos.

O programa que segue vai colocar um número do teclado e convertê-lo em um inteiro:

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

printf("entre um numero: ");
gets(s);
n = atoi(s);
}
```

close(fd)
int fd;

close( ) é utilizado para fechar um arquiov em disco que fora aberto ao utilizar **open( )** ou **creat( )**. É parte do sistema de arquivo de E/S sem armazenamento intermediário. **close( )** devolve 0 quando a operação é bem-sucedida.

O descritor de arquivo **fd** é um inteiro devolvido pela chamada à **open( )** ou **creat( )**.

O programa seguinte abre um arquivo chamado **test** para uma E/S sem armazenamento intermediário e em seguida fecha-o:

```
main()
{
int fd;
char buf[128];

if((fd = open("test",0)) == -1)
  {
  printf(nao pode abrir arquivo.\n");
  exit(1);
  }
read(fd,buf,128); /* leia um buffer */
close(fd);
}
```

creat (name, mode)
char *name;
int mode;

Utilizamos um **creat( )** para gerar um arquivo novo e abri-lo para operações de escrita. A função **creat( )** é parte do sistema de arquivo de E/S sem armazenamento intermediário.

A série de caracteres **name** deverá ser um ponteiro para um nome de arquivo válido; **mode** especifica o modo de proteção e é opcional na maioria das implementações de microcomputadores com C.

Caso **creat( )** seja bem-sucedido na abertura de um arquivo novo, devolverá um descritor de arquivo; caso contrário, devolverá −1, o que indica uma falha.

Caso já exista um arquivo com o mesmo nome, ele será apagado, de modo que a utilização de **creat( )** deverá ser feita com cuidado.

A função que segue vai criar e abrir um arquivo definido pelo usuário para operações de leitura, e devolver o descritor de arquivo.

```
cr_file()
{
char name[80];
int fd;

printf("entre nome do arquivo: ");
gets(name);
if((fd = creat(name,0)) == -1)
  {
  printf("nao pode abrir arquivo\n");
  return -1;
  }
return fd;
}
```

fclose(fp)
FILE *fp;

fclose(fp) é parte do sistema de arquivo E/S com armazenamento intermediário. É utilizado para escrever qualquer dado que ainda exista na área intermediária para o arquivo e fechar o arquivo. O arquivo deverá ter sido previamente aberto com a utilização de **fopen( )**, a função de abertura da E/S com área intermediária. O ponteiro de arquivo é **fp**, anteriormente devolvido pela chamada à **fopen( )**.

Caso **fclose( )** seja bem-sucedido, devolverá 0; caso contrário, devolverá −1.

Por exemplo, este programa abre um arquivo para operações de escrita com armazenamento intermediário e fecha-o; apenas um caractere é escrito no arquivo.

```
main()
{
FILE *fp;

if((fp = fopen("test","w")) == 0)
  {
  printf("nao pode abrir arquivo.\n");
  exit(1);
  }
putc('A',fp); /* escreve o caractere A */
fclose(fp);
}
```

FILE *fopen(name,mode)
char *name;
char *mode;

A função **fopen( )** é utilizada para abrir um arquivo para operações E/S com armazenamento intermediário. **name** especifica o nome do arquivo e **mode** é uma série que especifica como o arquivo será acessado. Apresentamos a seguir uma lista das opções **mode**:

r	abrir arquivo apenas para leitura
w	abrir arquivo apenas para escrita
a	abrir arquivo para escrita e acréscimos no final
rw	abrir arquivo para leitura/escrita

Caso o arquivo seja aberto por meio de **w**, qualquer arquivo de mesmo nome que já exista será cancelado. Para fazer acréscimos em um arquivo, utilize **a** para anexar ao final.

Um **fopen( )** devolve um ponteiro de arquivo do tipo **FILE** em caso de sucesso, e 0 em caso contrário.

A função que segue vai abrir um arquivo para o modo de leitura/escrita. Neste exemplo, o nome do arquivo é passado para a função e, em caso de sucesso, o ponteiro de arquivo será devolvido.

```
FILE *op_file(name)
char *name;
{
FILE *fp;

if((fp = fopen(name,"rw")) == 0)
   {
   printf("nao pode abrir arquivo.\n");
   }
return(fp);
}
```

getc(fp)
FILE *fp;

A função **getc( )** devolve o próximo caractere do arquivo apontado por **fp**, o ponteiro de arquivo.

Por exemplo, o programa que segue abre um arquivo denominado **test** para entrada e lê um caractere por vez até que seja encontrado um caractere de EOF. Todo caractere lido é impresso na tela.

```
main()
{
FILE *fp;
char ch;

if((fp = fopen("test","r")) == 0)
  {
  printf("nao pode abrir arquivo.\n");
  exit(1);
  }
do
  {
  ch = getc(fp);
  putchar(ch);
  } while(ch != EOF);
fclose(fp);
}
```

getchar( )

A função **getchar( )** devolve o próximo caractere do console.

A função que segue vai ler uma série de dígitos colocados pelo console e devolver o seu valor inteiro.

```
get_num()
{
char s[80],*temp;

temp = s;
do
  {
  *temp = getchar(); /* leia um digito */
  if(isdigit(*temp)) temp++;
  } while(*(temp - 1) != '\r'); /* ate retorno */
*temp = '\0'; /* nulo terminador */
return(atoi(s));
}
```

char *gets(s)
char *s;

A função **gets( )** é utilizada para ler uma série de caracteres do teclado e colocar a série no vetor de caracteres s. A entrada é encerrada com a digitação de um retorno de carro ou quando um

EOF é recebido. Entretanto, nem o retorno de carro ou o caractere de EOF torna-se parte da série. Após a chamada, s será encerrado com um nulo. A função **gets( )** devolve um ponteiro para s, ou um nulo, caso seja encontrado um erro ou um EOF.

Normalmente, **gets( )** permite a utilização de retrocesso e tabulações. Algumas implementações permitem ainda outros caracteres especiais de edição.

O programa exemplo abaixo insere uma série do teclado e imprime-a de trás para frente na tela:

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

printf("digite uma serie: ");
gets(s);
for(t = strlen(s);t;--t) putchar(s[t]);
```

char * malloc(size)
unsigned size;

Denominado **alloc( )** em alguns sistemas, **malloc( )** é utilizado para alocar um determinado número, **size**, de caracteres de memória livre e devolver um ponteiro ao seu início. A função **malloc( )** faz parte da rotina de alocação dinâmica.

Caso haja falha em uma solicitação de alocação – isto é, caso haja insuficiência de memória para atender à solicitação – um ponteiro nulo será devolvido.

Você sempre deve tomar muito cuidado em assegurar que está recebendo um ponteiro válido de **malloc( )**.

Esta é a forma geral de um programa que vai alocar 80 bytes de memória, e em seguida liberá-los.

```
main( )
{
    char *p;
    p= malloc(80);
    if(!p) {
        printf("sem memória \n");
        exit(2);
    }
    .
    .
    .
    free(p);
}
```

open(name, mode)
char *name
int mode;

A função **open( )** é utilizada para abrir um arquivo de E/S sem armazenamento intermediário. Devolve um descritor de arquivo inteiro em caso de sucesso, e −1 em caso de falha.

O **name** é qualquer nome de arquivo válido e **mode** determina a forma de acesso. A seguir indicamos os valores possíveis para **mode**:

0 apenas leitura
1 apenas escrita
2 leitura/escrita

Lembre-se de que **open( )** falha caso o arquivo especificado não exista; **open( )** não pode ser utilizado para gerar um arquivo.

A função que segue vai abrir um arquivo para acesso apenas de leitura em caso que não tenha armazenamento intermediário, se existir; caso não exista, será criado:

```
op_file(name)
{
int fd;

if((fd = open(name,2)) == -1)
  if((fd = creat(name,0)) == -1)
    printf("nao pode abrir arquivo\n");
return(fd);
}
```

printf(control,arglist)
char *control;

A função **printf( )** é a função de saída geral do console. Poderá ser utilizada para apresentar qualquer um dos tipos de dados inerentes ao C, bem como série. Permite ainda a utilização dos especificadores de campo para controlar a maneira com que a informação fica na tela. (Veja o Capítulo 6, que fornece uma explicação completa para **printf( )**.)

Após a execução das seguintes declarações

```
count = 10;
printf("contador e' : %d\n",count);
```

a tela de seu computador vai apresentar

```
contador e' : 10
```

```
putc(ch,fp)
  char ch;
  FILE *fp;
```

A função **putc( )** é utilizada para escrever um caractere em um arquivo anteriormente aberto, utilizando a função **fopen( )** de E/S com armazenamento intermediário, devolvendo **fp**. Após cada escrita bem-sucedida ele devolve **ch**, e um caractere **EOF** ao atingir o fim-de-arquivo.

A função abaixo poderá ser utilizada para escrever uma série no arquivo especificado:

```
wr_string(s,fp)
char *s;
FILE *fp;
{
while(*s) if(putc(*s++,fp) == EOF)
   {
   printf("fim-do-arquivo\n");
   return;
   }
}
```

```
read(fd,buffer,bufsize)
  int fd;
  int bufsize;
  char *buffer;
```

A função de leitura do sistema de E/S sem armazenamento intermediário é **read( )**. Lerá um número, **bufsize**, de caracteres na região de memória apontada pelo **buffer**. O **fd** deverá ter sido devolvido por uma chamada bem-sucedida à **open( )**.

Caso **read( )** seja bem-sucedido, devolverá o número de bytes lidos. Caso um fim-de-arquivo tenha sido observado sem que nenhum byte tenha sido lido, a função devolverá 0. Em caso de erro, **read( )** devolverá – 1.

A função abaixo lerá uma área intermediária, enchendo de dados o arquivo especificado:

```
rd_buf(buf,fd,size)
char *buf;
int fd;
int size;
{
if(read(fd,buf,size) == -1)
   {
   printf("erro de leitura");
   return(-1);
   }
}
```

scanf(control,arglist)
char *control;

scanf( ) é uma função de entrada generalizada que lerá informação do console e colocá-la, após devidamente formatada, nas variáveis em **arglist**. Todo argumento em **arglist** deverá ser um ponteiro.

Uma descrição completa de **scanf( )** é encontrada no Capítulo 6.

O programa abaixo lerá uma série e um inteiro do teclado:

```
main()
{
int x;
char s[80];
printf("entre uma serie e um inteiro:");
scanf("%s%d",s,&x);
}
```

char *strcat(s1,s2)
char *s1,*s2;

A função **strcat( )** concatena a série s2 ao fim de **s1**. As duas séries deverão estar terminadas em nulo e o resultado é terminado em nulo. Um **strcat( )** devolve um ponteiro ao **s1**.

O código abaixo imprimirá **hello there** na tela:

```
char first[20],second[20];

strcpy(first,"hello");
strcpy(second,"there");
strcat(first,second);
printf(first);
```

char *strcmp(s1,s2)
char *s1,*s2

A função **strcmp( )** compara duas séries terminadas em nulo e devolve 0 quando são iguais. Caso **s1** seja lexicograficamente maior do que s2, um número positivo será devolvido; caso contrário, um número negativo será devolvido. (Em vez disso, algumas implementações poderão devolver o primeiro caractere que não coincide, de maneira que convém verificar o manual de seu compilador C.)

Por exemplo, a seguinte função poderá ser utilizada como uma rotina de verificação de senha:

```
password()
{
char s[80],*strcmp();

printf("entre senha: ");
gets(s);
if(strcmp(s,"pass"))
   {
   printf("senha invalida\n");
   return 0;
   }
return 1;
}
```

char *strcpy(s1,s2)
char *s1,*s2;

Uma função **strcpy( )** é utilizada para copiar os conteúdos de s2 e s1. s2 deverá ser um ponteiro para uma série terminada em nulo. A função **strcpy( )** devolve um ponteiro para s1.

O seguinte trecho de codificação vai copiar **hello** na série **str**:

```
char *str;

if(str = malloc(80))
   {
   printf("sem memoria");
   exit(1);
   }
strcpy(str,"hello");
```

strlen(s)
char *s;

A função **strlen( )** devolve o comprimento da série terminada em nulo apontado por s.

O exemplo abaixo imprimirá o número 5 na tela.

```
strcpy(s,"hello");
printf("%d",strlen(s));
```

tolower(ch);
char ch;

Esta função devolve o equivalente em minúscula de **ch**, se **ch** for uma letra; caso contrário **ch** é devolvido inalterado.

Por exemplo,

```
putchar(tolower('Q'));
```

apresenta **q**, enquanto

```
putchar(tolower('!'));
```

apresenta **!**.

toupper(ch)
char ch;

A função **toupper( )** devolve o equivalente em maiúscula de **ch**, se **ch** for uma letra; caso contrário, **ch** é devolvido inalterado.

Por exemplo,

```
putchar(toupper('a'));
```

apresenta **A**, enquanto

```
putchar(toupper('!'));
```

apresenta **!**.

write(fd,buffer,bufsize)
int fd,bufsize;
char *buffer;

A função **write( )** é utilizada para escrever um número **bufsize** de caracteres de **buffer** para o arquivo especificado em **fd**. É parte do sistema E/S sem área intermediária. O **fd** deverá ter sido devolvido por uma chamada bem-sucedida para **open( )** ou **creat( )**.

Sendo bem-sucedido, **write( )** devolve o número de caracteres escritos; caso contrário será devolvido −1.

A seguinte função escreverá um "buffer" de dados para o arquivo especificado:

```
wr_buf(buf,fd,size)
char *buf;
int   fd;
int   size;
{
if(write(fd,buf,size) == -1)
  {
  printf("erro de escrita");
  return(-1);
  }
return 0;
}
```

ÍNDICE ANALÍTICO

Composição e Arte:
JAG Composições e Artes Gráficas Ltda.

Impressão e Acabamento
GRÁFICA E EDITORA FCA
AV. HUMBERTO DE ALENCAR CASTELO BRANCO, 3972 - TEL.: 419-0200
SAO BERNARDO DO CAMPO - CEP 09700 - SP

OUTROS LIVROS NA ÁREA

LINGUAGENS

Botelho – Basic Prático – Intermediário e Avançado
Carvalho – Assembler para o TK 90X
Carvalho – BASIC para o TK 90X
Fox/Fox – Iniciação ao BASIC
Gottfried – Programação com BASIC (SC)
Hehl – FORTRAN IV
Hehl – Linguagem de Programação Estruturada: FORTRAN 77
Hogan – Forth – Guia do Usuário
Medidata – MUMPS – Guia do Usuário
Newcomer – Cobol Estruturado (SC)
Peckham – Manual de BASIC para o Apple II
Siragusa – BASIC Estruturado

0-07-450414-2