

pontos e um tamanho) são usados para preenchimento. O tamanho especial 0 pode ser usado para forçar o alinhamento no próximo limite de palavra.

Os campos são atribuídos da esquerda para a direita em algumas máquinas e da direita para a esquerda em outras. Isso significa que, embora os campos sejam úteis para manterem estruturas de dados definidas internamente, a questão de qual extremo vem primeiro deve ser cuidadosamente considerada quando se apanha dados definidos externamente; os programas que dependem dessas coisas não são portáteis. Os campos só podem ser declarados como int; por questão de portabilidade, especifique signed ou unsigned explicitamente. Eles não são vetores, e não possuem endereços, de forma que o operador & não pode ser aplicado aos mesmos.

## Capítulo 7

### ENTRADA E SAÍDA

As facilidades de entrada e saída são parte integrante da própria linguagem C, como enfatizado na nossa apresentação até agora. Apesar disso, os programas interagem com seu ambiente de formas muito mais complicadas do que aquelas que mostramos anteriormente. Neste capítulo descreveremos a biblioteca-padrão, um conjunto de funções que fornecem entrada e saída, manipulação de cadeias, gerenciamento de memória, rotinas matemáticas e uma série de outros serviços para programas em C. Iremos nos concentrar na entrada e saída.

O padrão ANSI define estas funções da biblioteca com precisão, de forma que possam existir em forma compatível com qualquer sistema onde exista o C. Os programas que confinam suas interações com o sistema a facilidades fornecidas pela biblioteca-padrão podem ser movidos de um sistema para outro sem mudanças.

As propriedades das funções de biblioteca são especificadas em mais de uma dúzia de arquivos-cabeçalho; já vimos diversos deles, incluindo `<stdio.h>`, `<string.h>` e `<ctype.h>`. Não apresentaremos toda a biblioteca aqui, pois estamos mais interessados em escrever programas em C que a utilizem. A biblioteca é descrita em detalhes no Apêndice B.

#### 7.1 Entrada e Saída-Padrão

Como dissemos no Capítulo 1, a biblioteca implementa um modelo simples de entrada e saída de texto. Um fluxo de texto consiste numa seqüência de linhas; cada linha termina com um caractere de nova-linha. Se o sistema não opera dessa forma, a biblioteca faz o que é necessário para que pareça assim. Por exemplo, a biblioteca poderia converter o retorno de carro e mudança de linha para nova-linha na entrada e novamente para a saída.

O mecanismo de entrada mais simples é ler um caractere de cada vez da *entrada-padrão*, normalmente o teclado, com getchar:

```
int getchar (void)
```

getchar retorna o próximo caractere da entrada toda vez que é chamado, ou EOF quando encontrar o fim do arquivo. A constante simbólica EOF é definida em `<stdio.h>`. O valor é normalmente -1, mas os testes devem ser escritos em termos de EOF para que se tornem independentes do valor específico.

Em muitos ambientes, um arquivo pode ser substituído pelo teclado usando-se a convenção < para redireção da entrada: se um programa prog usa getchar, então a linha de comando

```
prog >arquivo
```

faz com que prog leia caracteres do arquivo. A troca da entrada é feita de tal forma que o próprio prog se distraia com a mudança; em particular, a cadeia “<arquivo>” não é incluída nos argumentos da linha de comando em argv.. A troca da entrada é também invisível se a entrada vier de um outro programa por meio de um mecanismo de conduto: em alguns sistemas, a linha de comando

```
outroprog | prog
```

roda os dois programas, outroprog e prog, e canaliza a saída-padrão de outroprog para a entrada-padrão de prog.

A função

```
int putchar(int)
```

é usada para saída: putchar(c) coloca o caractere c na saída-padrão, que por default é a tela. putchar retorna o caractere escrito, ou EOF se houver um erro. Normalmente, em geral a saída pode ser direcionada para um arquivo com >arquivo: se prog usa putchar,

```
prog >arquivosai
```

escreverá a saída-padrão em arquivosai. Se os condutos forem suportados,

```
prog ' outroprog
```

coloca a saída-padrão de prog na entrada-padrão de outroprog.

A saída produzida por printf também é passada para a saída-padrão. As chamadas a putchar e printf podem ser intervaladas — a saída aparece na ordem em que foram feitas as chamadas.

Cada arquivo-fonte que se refere a uma função da biblioteca de entrada/saída deve conter a linha

```
#include <stdio.h>
```

antes de sua primeira referência. Quando o nome é delimitado por < e >, é feita uma procura ao cabeçalho em um conjunto padrão de locais (por exemplo, nos sistemas UNIX, normalmente no diretório /usr/include).\*

Muitos programas lêem somente um fluxo de entrada e escrevem um único fluxo de saída; para tais programas, a entrada e saída com getchar, putchar e printf pode ser inteiramente adequada, e certamente o bastante para dar início. Isso é particularmente verdade se a redireção for usada para conectar a saída de um programa à entrada do próximo. Por exemplo, considere o programa minúsculo, que converte sua entrada para letras minúsculas:

```
#include <stdio.h>
#include <ctype.h>

main() /* minúsculo: converte entrada para minúsculas */
{
    int c;

    while ((c = getchar()) != EOF)
        putchar(tolower(c));
    return 0;
}
```

A função tolower está definida em <ctype.h>; ela converte uma letra maiúscula em minúscula, e retorna outros caracteres sem modificação. Como dissemos anteriormente, “funções” como getchar e putchar em <stdio.h> e tolower em <ctype.h> são normalmente macros, evitando assim o adicional de uma chamada de função por caractere. Mostraremos como isso é feito na Seção 8.5. Independente de como as funções de <ctype.h> sejam implementadas em uma determinada máquina, os programas que as usam não necessitam do conhecimento do conjunto de caracteres.

**Exercício 7.1.** Escreva um programa que converta maiúsculas para minúsculas ou minúsculas para maiúsculas, dependendo do nome com que for chamado, encontrado em argv[0]. □

## 7.2 Saída Formatada — Printf

A função de saída printf traduz valores internos para caracteres. Usamos printf informalmente nos capítulos anteriores. A descrição aqui cobre os usos mais típicos, mas não é completa; para uma descrição completa, veja no Apêndice B.

```
int printf(char *formato, arg1, arg2, ...)
```

printf converte, formata e imprime seus argumentos na saída-padrão sob o controle do formato. Ele retorna o número de caracteres impressos.

A cadeia de formato contém dois tipos de objetos: caracteres comuns, que são copiados no fluxo de saída, e especificações de conversão, cada uma das causando uma conversão e impressão do próximo argumento sucessivo a printf. Cada especificação de conversão começa com um % e termina com um caractere de conversão. Entre o % e o caractere de conversão pode haver, em ordem:

\* No Turbo C 1.0, sob o MS-DOS, uma opção do menu principal (“ENVIRONMENT”) permite a especificação de vários diretórios onde o programa procurará arquivos como cabeçalhos (*cabeç.h*), arquivos fonte (*programa.c*), bibliotecas (*nome.lib*) e arquivos objeto (*programa.obj*). (N. do T.)

- \* Um sinal de menos, que especifica ajustamento à esquerda do argumento convertido.
- \* Um número que especifica a largura mínima do campo. O argumento convertido será impresso em um campo com pelo menos esta largura. Se necessário, ele será preenchido à esquerda (ou direita, se o ajustamento à esquerda for solicitado) para compor a largura do campo.
- \* Um ponto, que separa a largura do campo da sua precisão.
- \* Um número, a precisão, que especifica o número máximo de caracteres a ser impresso em uma cadeia, ou o número de dígitos após o ponto decimal de um valor de ponto-flutuante, ou o número mínimo de dígitos para um inteiro.
- \* Um h se o inteiro tiver que ser impresso como short, ou l (letra ele) como long.

Os caracteres de conversão são mostrados na Tabela 7-1. Se o caractere após o % não for uma especificação de conversão, o comportamento é indefinido.

Uma largura ou precisão pode ser especificada por \*, quando o valor é calculado convertendo-se o próximo argumento (que deve ser um int). Por exemplo, para imprimir até max caracteres da cadeia s,

```
printf("%.*s", max, s);
```

**TABELA 7-1. CONVERSÕES BÁSICAS DE PRINTF**

CARACTERE	TIPO DE ARGUMENTO; IMPRESSO COMO
d,i	int; número decimal.
o	int; número octal não sinalizado (sem um zero inicial).
x,X	int; número hexadecimal não sinalizado (sem um Ox ou OX inicial), usando abcdef ou ABCDEF para 10,...,15.
u	int; número decimal não sinalizado.
c	int; único caractere.
s	char *; imprime caracteres da cadeia até um '\0' ou número de caracteres indicado na precisão.
f	double; [-]m.ddddd, onde o número de d's é dado pela precisão (default 6).
e,E	double; [-]m.ddddd, e×8xx ou [-]m.ddddd E±xx, onde número de d's é dado pela precisão (default 6).
g,G	double; usa %e ou %E se o expoente for menor que - 4 ou maior ou igual à precisão; caso contrário, usa %f. Zeros adicionais e um ponto decimal final não são impressos.
p	void *; apontador (representação dependente da implementação).
%	nenhum argumento é convertido; imprime um %.

A maioria das conversões de formato já foi ilustrada em capítulos anteriores. Uma exceção é a precisão relacionada a cadeias de caracteres. A tabela a se-

guir mostra o efeito de uma série de especificações para imprimir "Brasil, hoje" (12 caracteres). Colocamos sinais de dois pontos ao redor de cada campo para que você possa ver sua extensão.

:%s:	:Brasil, hoje:
:%10s:	:Brasil, hoje:
:%.10s:	:Brasil, ho:
:%-10s:	:Brasil, hoje:
:%.15s:	:Brasil, hoje:
:%-15s:	:Brasil, hoje :
:%15.10s:	: Brasil, ho:
:%-15.10s:	:Brasil, ho :

Um aviso: printf usa seu primeiro argumento para decidir quantos argumentos virão a seguir e quais são seus tipos. Se não houver argumentos suficientes, ou se forem do tipo errado, a função irá se confundir, e você terá respostas erradas. Esteja também ciente da diferença entre estas duas chamadas:

```
printf(s);           /* FALHA se s tiver % */
printf("%s", s);    /* SEGURO */
```

A função sprintf faz as mesmas conversões que printf, mas armazena a saída em uma cadeia:

```
int (sprintf(char *cadeia, char *formato, arg1, arg2, ...))
```

sprintf formata os argumentos em arg<sub>1</sub>, arg<sub>2</sub>, etc., de acordo com o formato, mas coloca o resultado na cadeia ao invés de enviá-lo à saída-padrão; cadeia deve ter tamanho suficiente para receber o resultado.

**Exercício 7-2.** Escreva um programa que imprima uma entrada qualquer de forma comprehensível. No mínimo, ele deve imprimir caracteres não-visíveis em octal ou hexadecimal, segundo a preferência local, e dobrar linhas longas. □

### 7.3 Listas de Argumentos de Tamanho Variável

Esta seção contém uma implementação de uma versão mínima de printf, mostrando como escrever uma função que processa uma lista de argumentos de tamanho variável de forma portável. Como estamos interessados principalmente no processamento dos argumentos, minprintf processará a cadeia de formato e os argumentos mas chamará o printf real para fazer as conversões de formato.

A declaração apropriada para printf é

```
int printf(char *fmt, ...)
```

onde a declaração ... significa que o número de tipos destes argumentos pode

variável. A declaração ... só pode aparecer ao final de uma lista de argumento. Nossa minprintf é declarado da seguinte forma:

```
void minprintf(char *fmt, ...)
```

não retornaremos o contador de caracteres, como faz printf.

O truque aqui é a forma com que minprintf percorre a lista de argumento quando a lista nem sequer tem um nome. O cabeçalho-padrão <stdarg.h> contém um conjunto de definições de macro que define como percorrer uma lista de argumento. A implementação deste cabeçalho varia de uma máquina para outra, mas a interface apresentada pelo mesmo é uniforme.

O tipo va\_list é usado para declarar uma variável que referir-se-á a cada argumento por sua vez; em minprintf, esta variável é chamada aa, de “apontador de argumento”. A macro va\_start inicializa aa de forma que aponta para o primeiro argumento não nomeado. Ela deve ser chamada uma vez antes que seja usado. Deve haver pelo menos um argumento nomeado; o argumento nomeado final é usado por va\_star para dar início.

Cada chamada a va\_arg retorna um argumento e incrementa aa para o próximo; va\_arg usa um nome de tipo para determinar o tipo a retornar e o tamanho do passo a executar. Finalmente, va\_end realiza qualquer tarefa final necessária. Ela deve ser chamada antes que a função retorne.

Estas propriedades formam a base do nosso printf simplificado:

```
#include <stdarg.h>

/* minprintf: printf mínimo com lista argumento variável */
void minprintf(char *fmt, ...)
{
    va_list aa; /* aponta para cada argumento não nomeado */
    char *p, *valc;
    int vali;
    double vald;

    va_start(aa, fmt); /* faz aa apontar ao 1º arg. */
    for (p = fmt; *p; p++) {
        if (*p != '%') {
            putchar(*p);
            continue;
        }
        switch (*++p) {
        case 'd':
            vali = va_arg(aa, int);
            printf("%d", vali);
            break;
        case 'f':
            vald = va_arg(aa, double);
            printf("%f", vald);
            break;
        }
    }
}
```

```
case 's':
    for (valc = va_arg(aa, char *); *valc; valc++)
        putchar(*valc);
    break;
default:
    putchar(*p);
    break;
}
}
va_end(aa); /* finaliza */
}
```

**Exercício 7-3.** Modifique minprintf para lidar com outras facilidades de printf. □

## 7.4 Entrada Formatada — Scanf

A função scanf é a entrada correspondente a printf fornecendo muitas das facilidades de conversão na direção contrária.

```
int scanf(char *formato, ...)
```

scanf lê caracteres da entrada-padrão, interpreta-os segundo a especificação em formato, e armazena os resultados por meio dos argumentos restantes. O argumento de formato é descrito a seguir; os outros argumentos, *cada um devendo ser um apontador*, indicam onde a entrada convertida correspondente deve ser armazenada. Assim, como em printf, esta seção é um resumo das características mais úteis, e não uma lista completa.

scanf pára quando termina sua cadeia de formato, ou quando alguma entrada deixa de casar com a especificação de controle. Ela retorna, como seu valor, o número de itens de entrada casados e atribuídos com sucesso. Isso pode ser usado para decidir quantos itens foram encontrados. Ao final do arquivo, EOF é retornado; observe que este é diferente de 0, que significa que o próximo caractere de entrada não combina com a primeira especificação na cadeia de formato. A próxima chamada a scanf continua a procura imediatamente após o último caractere já convertido.

Há também uma função sscanf que lê de uma cadeia ao invés da entrada-padrão:

```
int sscanf(char *cadeia, char *formato, arg1, arg2, ...)
```

Ela analisa a cadeia de acordo com o formato, e armazena os valores resultantes em arg<sub>1</sub>, arg<sub>2</sub>, etc. Estes argumentos devem ser apontadores.

A cadeia de formato geralmente contém especificações de conversão, que são usadas para controlar a conversão da entrâda. A cadeia de formato pode conter:

- \* Espaços ou tabulações, que são ignorados.
- \* Caracteres comuns (não %), que devem combinar com o próximo caractere não espaço do fluxo de entrada.
- \* Especificações de conversão, consistindo no caractere %, um caractere \* opcional de supressão de atribuição, um número opcional especificando um tamanho máximo de campo, um h, l ou L opcional indicando o tamanho do destino, e um caractere de conversão.

Uma especificação de conversão direciona a conversão do próximo campo de entrada. Normalmente, o resultado é colocado na variável apontada pelo argumento correspondente. Se, entretanto, a supressão de atribuição for indicada pelo caractere, \* o campo de entrada é saltado; nenhuma atribuição é feita. Um campo de entrada é definido como uma cadeia de caracteres não espaço; ele estende-se ou até o próximo caractere de espaço ou até que a largura do campo, se especificada, for terminada. Isso significa que scanf lerá além dos limites de linha para encontrar sua entrada, pois as novas-linhas são espaço em branco. (Os caracteres de espaço são branco, tabulação, nova-linha, retorno de carro, tabulação vertical e alimentação de formulário.)

O caractere de conversão indica a interpretação do campo de entrada. O argumento correspondente deve ser um apontador, solicitado pela semântica da chamada por valor do C. Os caracteres de conversão são mostrados na Tabela 7-2.

**TABELA 7-2. CONVERSÕES BÁSICAS DE SCANF**

CARACTERE	DADO DE ENTRADA; TIPO DE ARGUMENTO
d	inteiro decimal; int*.
i	inteiro; int*. O inteiro pode estar em octal (0 no início) ou hexadecimal (0x ou 0X no início).
o	inteiro octal (com ou sem zero inicial); int*.
u	inteiro decimal não sinalizado; unsigned int *.
x	inteiro hexadecimal (com ou sem Ox ou OX inicial); int *.
c	caracteres; char *. Os próximos caracteres de entrada (default 1) são colocados no ponto indicado. O salto normal sobre espaço em branco é suprimido; para ler o próximo caractere não-espaco, use %1s.
s	cadeia de caracteres (sem aspas); char *, apontando para um vetor de caracteres com tamanho suficiente para a cadeia e o '\0' de término que será incluído.
e,f,g	número em ponto-flutuante com sinal opcional, ponto decimal opcional e expoente opcional; float *.
%	%literal; nenhuma atribuição é feita.

Os caracteres de conversão d, i, o, u e x podem ser precedidos por h para indicarem que um apontador para short ao invés de int aparece na lista de argumento, ou por l (letra ele) para indicar que um apontador para long aparece na lista de argumento. Semelhantemente, os caracteres de conversão e, f e g podem ser precedidos por l para indicarem que um apontador para double ao invés de float está na lista de argumento.

Como primeiro exemplo, a calculadora elementar do Capítulo 4 pode ser escrita com scanf para fazer a conversão de entrada:

```
#include <stdio.h>

main() /* calculadora elementar */
{
    double soma, v;

    soma = 0;
    while (scanf("%1f", &v) == 1)
        printf("\t%.2f\n", soma += v);
    return 0;
}
```

Suponha que queremos ler linhas de entrada que contenham datas com o formato

25 dez 1988

O comando scanf seria

```
int dia, ano;
char nomemes[20];

scanf("%d %s %d", &dia, nomemes, &ano);
```

Nenhum & é usado com nomes, pois um nome de vetor já é um apontador.

Os caracteres literais podem aparecer na cadeia de formato de scanf; eles devem combinar com os mesmos caracteres na entrada. Assim, poderíamos ler datas do formato dd/mm/aa com este comando scanf:

```
int dia, mes, ano;

scanf("%d/%d/%d", &dia, &mes, &ano);
```

scanf ignora espaços em branco e tabulações na sua cadeia de formato. Além do mais, ele salta espaços em branco (brancos, tabulações, novas-linhas etc.) à medida que procura valores de entrada. Para ler a entrada cujo formato não seja fixo, normalmente é melhor ler uma linha de cada vez, depois separá-la com scanf. Por exemplo, suponha que queremos ler linhas que possam conter uma data em qualquer um dos formatos acima. Então, poderíamos escrever

```
while (lelinha(linha, sizeof(linha)) > 0) {
    if (sscanf(linha, "%d %s %d", &dia, nomemes, &ano) == 3)
        printf("valido: %s\n", linha); /* forma 25 dez 1988 */
    else if (sscanf(linha, "%d/%d/%d", &dia, &mes, &ano) == 3)
        printf("valido: %s\n", linha); /* forma dd/mm/aa */
```

```

else
    printf("invalido: %s\n", linha); /* forma invalida */
}

```

Chamadas a `scanf` podem ser misturadas com chamadas a outras funções de entrada. A próxima chamada a qualquer função de entrada começará lendo o primeiro caractere não lido por `scanf`.

Um lembrete final: os argumentos para `scanf` e `sscanf` *devem* ser apontadores. O erro mais comum é o de escrever

```
scanf("%d", n);
```

ao invés de

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

Este erro geralmente não é detectado em tempo de compilação.

**Exercício 7-4.** Escreva uma versão particular de `scanf` semelhante a `minprintf` da seção anterior. □

**Exercício 7-5.** Reescreva a calculadora pós-fixada do Capítulo 4 para usar `scanf` e `sscanf` para realizar a entrada e conversão de números. □

## 7.5 Acesso a Arquivos

Todos os exemplos até agora leram a entrada-padrão e escreveram na saída-padrão, que são automaticamente definidos para um programa pelo sistema operacional local.

O próximo passo será escrever um programa que acessa um arquivo que já *não* está conectado ao programa. Um programa que ilustra a necessidade dessas operações é `cat`, que concatena um conjunto de arquivos nomeados na saída-padrão. `cat` é usado para imprimir arquivos na tela, e também como um coletor de entrada de uso geral para programas que não possuem a capacidade de acessar arquivos por nome. Por exemplo, o comando

```
cat x.c y.c
```

imprime o conteúdo dos arquivos `x.c` e `y.c` (e nada mais) na saída-padrão.

A questão é como conseguir que os arquivos nomeados sejam lidos — ou seja, como conectar os nomes externos que o usuário indica aos comandos que lêem os dados.

As regras são simples. Antes que possa ser lido ou gravado, um arquivo deve ser *aberto* pela função de biblioteca `fopen`. `fopen` usa um nome externo como `x.c` ou `y.c`, faz alguma tarefa inicial e negociação com o sistema operacional

(sendo que nos detalhes não estamos interessados), e retorna um apontador a ser usado em leituras ou escritas subsequentes no arquivo.

Este apontador, chamado *apontador de arquivo*, aponta para uma estrutura que contém informações sobre o arquivo, como o local de um buffer, a posição do caractere corrente no buffer, se o arquivo está sendo lido ou gravado, e se foram encontrados erros ou o fim de arquivo. Os usuários não precisam saber os detalhes, pois as definições obtidas de `<stdio.h>` incluem uma declaração de estrutura chamada `FILE`. A única declaração necessária para um apontador de arquivo é exemplificada por

```
FILE *fp;
FILE *fopen(char *nome, char *modo);
```

Isto diz que `fp` é um apontador para um `FILE` (arquivo), e `fopen` retorna um apontador para um `FILE`. Observe que `FILE` é um nome de tipo, como `int`, e não uma etiqueta de estrutura; ele é definido com um `typedef`. (Detalhes de como `fopen` pode ser implementado no sistema UNIX são dados na Seção 8.5).

A chamada a `fopen` em um programa é

```
fp = fopen(nome, modo);
```

O primeiro argumento de `fopen` é uma cadeia de caracteres contendo o nome do arquivo. O segundo argumento é o *modo*, também uma cadeia de caracteres, que indica como se pretende usar o arquivo. Os modos permitidos incluem leitura ("r"), gravação ("w") e anexação ("a"). Alguns sistemas distinguem arquivos de texto e binários; para o último, um "b" deve ser incluído na cadeia de modo.

Se um arquivo que não existe for aberto para escrita ou anexação, ele é criado, se for possível. Abrir um arquivo existente para gravação faz com que o conteúdo antigo seja desconsiderado, enquanto abre-lo para anexação o preserva. Tentar ler um arquivo que não existe é um erro, e também pode haver outras causas de erro, como tentar ler um arquivo quando não existe permissão. Se houver um erro, `fopen` retornará `NULL`. (O erro pode ser identificado mais exatamente; veja a discussão sobre funções de manipulação de erro ao final da Seção 1 do Apêndice B.)

O próximo passo necessário é uma forma de ler ou gravar no arquivo uma vez aberto. Há diversas possibilidades, das quais `getc` e `putc` são as mais simples. `getc` retorna o próximo caractere de um arquivo; ele precisa do apontador de arquivo para que saiba qual o arquivo a usar.

```
int getc(FILE *fp)
```

`getc` retorna o próximo caractere do fluxo referenciado por `fp`; ele retorna `EOF` para fim de arquivo ou erro.

`putc` é uma função de saída:

```
int putc(int c, FILE *fp)
```

`putc` escreve o caractere `c` no arquivo `fp` e retorna o caractere escrito, ou `EOF` se

houver erro. Assim como getchar e putchar, getc e putc podem ser macros no lugar de funções.

Quando um programa em C é iniciado, o ambiente do sistema operacional é responsável por abrir três arquivos e fornecer apontadores de arquivo para eles. Esses arquivos são a entrada-padrão, a saída-padrão, e o erro-padrão; os apontadores de arquivo correspondentes são chamados stdin, stdout e stderr, e são declarados em <stdio.h>. Normalmente stdin é conectado ao teclado e stdou e stderr são conectados à tela, mas stdin e stdout podem ser redirecionados para arquivos ou condutos, como descrevemos na Seção 7.1.

getchar e putchar podem ser definidos em termos de getc, putc, stdin e stdout da seguinte forma:

```
#define getchar()     getc(stdin)
#define putchar(c)     putc(c, stdout)
```

Para a entrada ou saída formatada de arquivos, as funções fscanf e fprintf podem ser usadas. Elas são idênticas a scanf e printf, exceto que o primeiro argumento é um apontador de arquivo que especifica o arquivo a ser lido ou gravado; a cadeia de formato é o segundo argumento.

```
int fscanf(FILE *fp, char *formato, ...)
int fprintf(FILE, *fp, char *formato, ...)
```

Com estas considerações preliminares, estamos em posição de escrever o programa cat para concatenar arquivos. O projeto básico é um que tem sido conveniente para muitos programas. Se houver argumentos na linha de comando, eles são interpretados como nomes-de-arquivo, e processados em ordem. Se não houver argumentos, a entrada-padrão é processada.

```
#include <stdio.h>

/* cat: concatena arquivos, versao 1 */
main(int argc, char *argv[])
{
    FILE *fp;
    void copia_arq(FILE *, FILE *);

    if (argc == 1) /* sem arg.: copia entrada-padrão */
        copia_arq(stdin, stdout);
    else
        while (--argc > 0)
            if ((fp = fopen(*++argv, "r")) == NULL) {
                printf("Cat: não posso abrir %s\n", *argv);
                return 1;
            } else {
                copia_arq(fp, stdout);
                fclose(fp);
            }
}
```

```
return 0;
}

/* copia_arq: copia arquivo ifp para arquivo ofp */
void copia_arq(FILE *ifp, FILE *ofp)
{
    int c;

    while ((c = getc(ifp)) != EOF)
        putc(c, ofp);
}
```

Os apontadores de arquivo stdin e stdou são objetos do tipo FILE\*. Eles são constantes, entretanto, e *não* variáveis; não tente atribuir *nada* a eles.

A função

```
int fclose(FILE *fp)
```

é o inverso de fopen; ela encerra a conexão entre o apontador de arquivo e o nome externo que foi estabelecido por fopen, liberando o apontador de arquivo para outro arquivo. Como a maioria dos sistemas operacionais tem um limite para o número de arquivos que um programa pode abrir simultaneamente, é uma boa ideia liberar os apontadores quando não são mais necessários, como fizemos em cat. Há ainda um outro motivo para fclose em um arquivo de saída — ele esvazia o buffer em que putc está coletando a saída. fclose é chamado automaticamente para cada arquivo aberto quando um programa termina normalmente. (Você pode fechar stdin e stdout se não forem necessários. Eles também podem ser reatribuídos pela função de biblioteca freopen).

## 7.6 Tratamento de Erro — Stderr e Exit

O tratamento de erros em cat não é ideal. O problema é que se um dos arquivos não puder ser acessado por alguma razão, o diagnóstico é impresso ao final da saída concatenada. Isso poderia ser aceitável se a saída vai para a tela, mas não se for para um arquivo ou outro programa por meio de um conduto.

Para lidar melhor com esta situação, um segundo fluxo de saída, chamado stderr, é atribuído a um programa da mesma forma que stdin e stdout. A saída escrita em stderr normalmente aparece na tela mesmo que a saída-padrão seja redirecionada.

Vamos revisar cat para escrever suas mensagens de erro na saída-padrão.

```
#include <stdio.h>

/* cat: concatena arquivos, versao 2 */
main(int argc, char *argv[])
{
}
```

```

FILE *fp;
void copia_arq(FILE *, FILE *);
char *prog = argv[0]; /* nome de programa p/erros */

if (argc == 1) /* sem arg.: copia entrada-padrão */
    copia_arq(stdin, stdout);
else
    while (--argc > 0)
        if ((fp = fopen(*++argv, "r")) == NULL) {
            fprintf(stderr, "%s: não posso abrir %s\n",
                    prog, *argv);
            exit(1);
        } else {
            copia_arq(fp, stdout);
            fclose(fp);
        }
    if (ferror(stdout))
        fprintf(stderr, "%s: erro escrevendo stdout\n", prog);
        exit(2);
}
exit(0);
}

```

O programa sinaliza erros de duas formas. Primeiro, a saída de diagnóstico produzida por `fprintf` vai para `stderr`, de forma que passará à tela ao invés de desaparecer por um conduto ou arquivo de saída. Incluímos o nome do programa, a partir de `argv[0]`, na mensagem, de forma que se o programa for usado com outros, a fonte de um erro possa ser identificada.

Segundo, o programa usa a função da biblioteca-padrão `exit`, que termina a execução de um programa quando for chamada. O argumento de `exit` é disponível a qualquer processo que tenha chamado este, de forma que o sucesso ou falha de um programa possa ser testado por um outro programa que use este como subprocesso. Por convenção, um valor de retorno 0 indica que tudo está bem; valores não-zero geralmente sinalizam situações anormais. `exit` chama `fclose` para cada arquivo de saída aberto, esvaziando qualquer saída ainda num buffer.

Dentro de `main`, `return expr` é equivalente a `exit(expr)`. `exit` tem a vantagem de poder ser chamada por outras funções, e que as chamadas a ela podem ser encontradas com um programa de pesquisa de padrão como o do Capítulo 5.

A função `ferror` retorna não-zero se houve um erro no fluxo de `fp`.

```
int ferror(FILE *fp)
```

Embora os erros de saída sejam raros, eles acontecem (por exemplo, se um disco estiver cheio), de forma que um programa em produção deve isto também.

A função `feof(FILE *)` é semelhante a `ferror`; ela retorna não-zero se houve um fim de arquivo no arquivo especificado.

```
int feof(FILE *fp)
```

Geralmente não nos preocupamos com estado da saída em nossos pequenos programas ilustrativos, mas qualquer programa sério deve tomar o cuidado de retornar valores de estado significativos e úteis.

## 7.7 Entrada e Saída de Linhas

A biblioteca-padrão fornece uma rotina `fgets` que é semelhante à função `lelinha` que nós usamos nos capítulos anteriores:

```
char *fgets(char *linha, int maxlin, FILE *fp)
```

`fgets` lê a próxima linha de entrada (incluindo o caractere de nova-linha) do arquivo `fp` no vetor de caracteres `linha`; no máximo `maxlin-1` caracteres serão lidos. A linha resultante é terminada com `'\0'`. Normalmente `fgets` retorna `linha`; ao fim de arquivo ou em caso de erro ela retorna `NULL`. (Nosso `lelinha` retorna o tamanho da linha, que é um valor mais útil; zero significa fim do arquivo.)

Para a saída, a função `fputs` grava uma cadeia (que não precisa conter um caractere de nova-linha) em um arquivo:

```
int fputs(char *line, FILE *fp)
```

Ela retorna `EOF` se houver um erro, e zero em caso contrário.

As funções de biblioteca `gets` e `puts` são semelhantes a `fgets` e `fputs`, mas operam com `stdin` e `stdout`. Para confundir, `gets` deleta o `'\n'` ao término, e `puts` o inclui.

Para mostrar que não há nada mágico sobre funções tais como `fgets` e `fputs`, aqui estão elas, copiadas da biblioteca-padrão no nosso sistema:

```
/* fgets: obtém nomáximo n chars de iop */
char *fgets(char *s, int n, FILE *iop)
{
    register int c;
    register char *cs;

    cs = s;
    while (--n > 0 && (c = getc(iop)) != EOF)
        if ((*cs++ = c) == '\n')
            break;
    *cs = '\0';
    return (c == EOF && cs == s) ? NULL : s;
}
```

```
/* fputs: grava a cadeia s no arquivo iop */
int fputs(char *s, FILE *iop)
{
    int c;
```

```

while (c = *s++)
    putc(c, iop);
return ferror(iop) ? EOF : 0;
}

```

Sem qualquer motivo óbvio, o padrão especifica diferentes valores de retorno para `ferror` e `fputs`.

Não é difícil a implementação de nosso leitor a partir de `fgets`:

```

/* leitor: lê uma linha, retorna tamanho */
int leitor(char *linha, int max)
{
    if (fgets(linha, max, stdin) == NULL)
        return 0
    else
        return strlen(linha);
}

```

**Exercício 7-6.** Escreva um programa para comparar dois arquivos, imprimindo a primeira linha onde eles diferem. □

**Exercício 7-7.** Modifique o programa de pesquisa de padrão do Capítulo 5 para aceitar sua entrada a partir de um conjunto de arquivos nomeados ou, se nenhum arquivo for nomeado como argumento, da entrada-padrão. Seria bom imprimir o nome do arquivo quando uma linha com o padrão especificado fosse encontrada? □

**Exercício 7-8.** Escreva um programa que imprima um conjunto de arquivos, iniciando cada um com uma nova página, com um título e um contador de página corrente para cada arquivo. □

## 7.8 Algumas Funções Diversas

A biblioteca-padrão provê uma variedade de funções. Esta seção é um breve resumo das mais úteis. Maiores detalhes e muitas outras funções podem ser encontrados no Apêndice B.

### 7.8.1 Operações de Cadeia

Já mencionamos as funções de cadeia `strlen`, `strcpy`, `strcat` e `strcmp`, achadas em `<string.h>`. Na tabela a seguir, `s` e `t` são `char *`'s, e `c` e `n` são ints.

<code>strcat(s,t)</code>	concatena <code>t</code> ao final de <code>s</code>
<code>strncat(s,t,n)</code>	concatena <code>n</code> caracteres de <code>t</code> ao final de <code>s</code>
<code>strcmp(s,t)</code>	retorna negativo, zero, ou positivo para <code>s &lt; t</code> , <code>s == t</code> ,

<code>strncmp(s,t,n)</code>	ou <code>s &gt; t</code> o mesmo que <code>strcmp</code> mas somente com os primeiros <code>n</code> caracteres
<code>strcpy(s,t)</code>	copia <code>t</code> para <code>s</code>
<code>strncpy(s,t,n)</code>	copia no máximo <code>n</code> caracteres de <code>t</code> para <code>s</code>
<code>strlen(s)</code>	retorna o tamanho de <code>s</code>
<code>strchr(s,c)</code>	retorna apontador para primeiro <code>c</code> em <code>s</code> , ou <code>NULL</code> se não for achado
<code> strrchr(s,c)</code>	retorna apontador para último <code>c</code> em <code>s</code> , ou <code>NULL</code> se não for achado

### 7.8.2 Teste e Conversão de Classe de Caracteres

Diversas funções de `<ctype.h>` executam testes e conversões de caracteres. A seguir, `c` é um int que pode ser representado como um `unsigned char`, ou `EOF`. As funções retornam int.

<code>isalpha(c)</code>	não-zero se <code>c</code> é alfabético, 0 se não
<code>isupper(c)</code>	não-zero se <code>c</code> é maiúscula, 0 se não
<code>islower(c)</code>	não-zero se <code>c</code> é minúscula, 0 se não
<code>isdigit(c)</code>	não-zero se <code>c</code> é dígito, 0 se não
<code>isalnum(c)</code>	não-zero se <code>isalpha(c)</code> ou <code>isdigit(c)</code> , 0 se não
<code>isspace()</code>	não-zero se <code>c</code> é branco, tabulação, nova-linha, retorno, alimentação de formulário, tabulação vertical
<code>toupper(c)</code>	retorna <code>c</code> convertido para maiúsculo
<code>tolower(c)</code>	retorna <code>c</code> convertido para minúsculo

### 7.8.3 Ungetc

A biblioteca-padrão fornece uma versão um tanto restrita da função `ungetch` que escrevemos no Capítulo 4; ela é chamada `ungetc`.

```
int ungetc(int c, FILE *fp)
```

devolve o caractere `c` para o arquivo `fp`, e retorna ou `c`, ou `EOF` indicando um erro. Apenas um caractere devolvido é permitido por arquivo. `ungetc` pode ser usado com qualquer função de entrada como `scanf`, `getc` ou `getchar`.

### 7.8.4 Execução de Comando

A função `system(char *s)` executa o comando contido na cadeia de caracteres `s`, e depois continua a execução do programa corrente. O conteúdo de `s` depende muito do sistema operacional local. Como exemplo trivial, em sistemas UNIX, o comando

```
system("date");
```

faz com que o programa date seja executado; ele imprime a data e hora correntes na saída-padrão. system retorna um valor de estado, dependente do sistema, a partir do comando executado. No sistema UNIX, o retorno é o valor indicado por exit.

### 7.8.5. Gerenciamento de Memória

As funções malloc e calloc obtêm blocos de memória dinamicamente.

```
void *malloc(size_t n)
```

retorna um apontador para n bytes de memória não inicializada, ou NULL se o pedido não puder ser satisfeito.

```
void *calloc(size_t n, size_t tamanho)
```

retorna um apontador para um espaço suficiente para conter um vetor de n objetos do tamanho especificado, ou NULL se o pedido não puder ser satisfeito. A memória é inicializada com zero.

O apontador retornado por malloc ou calloc tem o alinhamento adequado ao objeto em questão, mas deve ser moldado para o tipo apropriado, como em

```
int *ip;  
ip = (int *) calloc(n, sizeof(int));
```

free(p) libera o espaço apontador por p, onde p foi inicialmente obtido por meio de uma chamada a malloc ou calloc. Não há distinções na ordem em que o espaço é liberado, mas é terrivelmente errado liberar algo não obtido por uma chamada a calloc ou malloc.

Também é um erro usar algo depois de ter sido liberado. Um trecho de código comum, mas incorreto, é este laço que libera itens da lista:

```
for (p = inicio; p != NULL; p = p -> prox) /* ERRADO */  
    free(p);
```

A forma correta é salvar tudo o que for necessário antes de liberar a memória:

```
for (p = inicio; p != NULL; p = q) {  
    q = p -> prox;  
    free(p);  
}
```

A seção 8.7 mostra a implementação de um alocador de armazenagem como malloc, onde os blocos alocados podem ser liberados em qualquer ordem.

### 7.8.6 Funções Matemáticas

Há mais de vinte funções matemáticas declaradas em <math.h>; aqui estão algumas usadas com mais freqüência. Cada uma utiliza um ou mais argumentos double e retorna um double.

$\sin(x)$	seno de x, x em radianos
$\cos(x)$	co-seno de x, x em radianos
$\atan2(y,x)$	arco-tangente de $y/x$ , em radianos
$\exp(x)$	função exponencial e elevado a x
$\log(x)$	logaritmo natural (base e) de x ( $x > 0$ )
$\log10(x)$	logaritmo comum (base 10) de x ( $x > 0$ )
$\pow(x,y)$	x elevado a y
$\sqrt{x}$	raiz quadrada de x ( $x \geq 0$ )
$\fabs(x)$	valor absoluto de x

### 7.8.7 Geração de Número Randômico

A função rand() calcula uma seqüência de inteiros pseudo-aleatórios na faixa de 0 a RAND\_MAX, que é definido em <stdlib.h>. Uma forma de produzir números randômicos de ponto flutuante maiores ou iguais a zero, mas menores que um é

```
#define frand() ((double) rand() / (RAND_MAX+1))
```

(Se a sua biblioteca já possui uma função para números randômicos de ponto-flutuante, ela provavelmente terá melhores propriedades estatísticas do que esta.)

A função srand(unsigned) define a semente para rand. A implementação portável de rand e srand sugerida pelo padrão aparece na Seção 2.7.

**Exercício 7-9.** Funções tais como isupper podem ser implementadas a fim de economizar espaço ou tempo. Estude estas possibilidades. □