

## E/S pelo Console

C é praticamente única no seu enfoque das operações de entrada/saída. A razão disso é que a linguagem não define nenhuma palavra-chave que realize E/S. Ao contrário, entrada e saída são efetuadas pelas funções da biblioteca. O sistema de E/S de C é uma parcela elegante da engenharia que oferece um flexível, porém coeso mecanismo para transferir dados entre dispositivos. No entanto, o sistema de E/S de C é muito grande e envolve diversas funções diferentes.

Em C, existe E/S pelo console e por meio de arquivo. Tecnicamente, C faz pouca distinção entre a E/S pelo console e a E/S de arquivo. Contudo, conceitualmente elas são mundos diferentes. Esse capítulo examina em detalhes as funções de E/S pelo console. O próximo capítulo apresenta o sistema de E/S de arquivo e descreve como os dois sistemas se relacionam.

Com uma exceção, esse capítulo cobre apenas as funções de E/S pelo console definidas pelo padrão C ANSI. Nem o padrão C ANSI nem o C tradicional definem qualquer função que realize o controle de uma tela “imaginária” ou gráficos porque essas operações variam enormemente entre máquinas. Ao contrário, as funções de E/S pelo console do padrão C realizam apenas saídas do tipo TTY. No entanto, a maioria dos compiladores inclui nas suas bibliotecas funções de controle de tela e gráficos que se aplicam ao ambiente específico para o qual o compilador foi projetado. A Parte 2 cobre uma amostra representativa dessas funções.

Este capítulo refere-se às funções de E/S através do console, como aquelas que realizam a entrada pelo teclado e a saída pela tela. Entretanto, essas funções têm, na realidade, a entrada e a saída padrões como o destino e/ou a origem de suas operações de E/S. Além disso, a entrada e a saída padrões podem ser redirecionadas a outros dispositivos. Esses conceitos são abordados no Capítulo 9.

## Lendo e Escrevendo Caracteres

A mais simples das funções de E/S pelo console são **getchar()**, que lê um caractere do teclado, e **putchar()**, que escreve um caractere na tela. A função **getchar()** espera até que uma tecla seja pressionada e devolve o seu valor. A tecla pressionada é também automaticamente mostrada na tela. A função **putchar()** escreve seu argumento caractere na tela a partir da posição atual do cursor. Os protótipos para **getchar()** e **putchar()** são mostrados aqui:

```
int getchar(void);
```

```
int putchar(int c);
```

O arquivo de cabeçalho dessas funções é **STDIO.H**. Como mostra seu protótipo, a função **getchar()** é declarada como retornando um inteiro. Contudo, você pode atribuir esse valor a uma variável **char**, como usualmente se faz, porque o caractere está contido no byte de baixa ordem. (Em geral, o byte de ordem mais alta é zero.) **getchar()** retorna **EOF** se ocorre um erro.

No caso de **putchar()**, apesar de ser declarada como pegando um parâmetro inteiro, você geralmente o chamará usando um argumento caractere. Apenas o byte de baixa ordem desse parâmetro é realmente passado para a tela. A função **putchar()** retorna o caractere escrito, ou **EOF** se ocorre um erro. (A macro **EOF** é definida em **STDIO.H** e é geralmente igual a -1.)

O programa seguinte lê caracteres do teclado e inverte a caixa deles. Isto é, escreve maiúsculas como minúsculas e minúsculas como maiúsculas. Para parar o programa, digite um ponto.

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

void main(void)
{
    char ch;

    printf("Entre com algum texto (digite um ponto para sair).\n");
    do {
        ch = getchar();

        if(islower(ch)) ch = toupper(ch);
        else ch = tolower(ch);

        putchar(ch);
    } while (ch!='.');
```

## Um Problema com `getchar()`

Existem alguns problemas potenciais com `getchar()`. O C ANSI definiu `getchar()` como sendo compatível com a versão original de C para o UNIX. Infelizmente, na sua forma original, `getchar()` armazena em um buffer a entrada até que seja pressionado ENTER. Isso porque os sistemas UNIX originais tinham um buffer de linha para os terminais de entrada — isto é, você tinha de pressionar ENTER para que a entrada fosse enviada ao computador. Isso deixa um ou mais caracteres esperando na fila depois que `getchar()` retorna, o que é incômodo em ambientes interativos. Embora o padrão ANSI especifique que `getchar()` pode ser implementada como uma função interativa, isso raramente ocorre. Portanto, se o programa anterior não se comportou como o esperado, você agora sabe por quê.

## Alternativas para `getchar()`

`getchar()` pode não ser implementada pelo seu compilador de modo a fazê-la útil em um ambiente interativo. Se esse for o caso, talvez você queira utilizar uma função diferente para ler caracteres do teclado. O padrão C ANSI não define nenhuma função que garanta uma entrada interativa, mas muitos compiladores C incluem funções alternativas de entrada pelo teclado. Embora essas funções não sejam definidas pelo ANSI, elas são recomendadas, já que `getchar()` não satisfaz às necessidades da maioria dos programadores.

As duas funções mais comuns, `getch()` e `getche()`, possuem os seguintes protótipos:

```
int getch(void);  
int getche(void);
```

Para a maioria dos compiladores, os protótipos para essas funções são encontrados em `CONIO.H`. A função `getch()` espera até que uma tecla seja pressionada e, então, retorna imediatamente. Ela não mostra o caractere na tela. A função `getche()` é igual a `getch()`, mas a tecla é mostrada. Este livro geralmente utiliza `getch()` ou `getche()` no lugar de `getchar()` quando um caractere precisa ser lido do teclado em um programa interativo. Contudo, se seu compilador não suporta essas funções alternativas, ou se `getchar()` for implementada como uma função interativa pelo seu compilador, você deverá substituir `getchar()` quando necessário.

Por exemplo, o programa anterior é mostrado aqui utilizando `getch()` em lugar de `getchar()`.

```
#include <stdio.h>  
#include <conio.h>  
#include <ctype.h>  
  
void main(void)  
{
```

```
char ch;

printf("Entre com algum texto (digite um ponto para
      sair).\n");
do {
    ch = getch();

    if(islower(ch)) ch = toupper(ch);
    else ch = tolower(ch);

    putchar(ch);
} while (ch!='.');
```

## Lendo e Escrevendo Strings

O próximo passo em E/S por meio do console, em termos de complexidade e capacidade, são as funções **gets()** e **puts()**. Elas lhe permitem ler e escrever strings de caracteres no console.

A função **gets()** lê uma string de caracteres inserida pelo teclado e coloca-a no endereço apontado por seu argumento ponteiro de caracteres. Você pode digitar caracteres no teclado até que o retorno de carro (CR) seja pressionado. O retorno de carro não se torna parte da string; em seu lugar é colocado um terminador nulo e **gets()** retorna. Não se pode devolver um retorno de carro utilizando **gets()** (embora **getchar()** possa). Você pode corrigir erros de digitação usando a tecla BACKSPACE antes de pressionar ENTER. O protótipo para **gets()** é

```
char *gets(char *str);
```

onde *str* é uma matriz de caracteres que recebe os caracteres enviados pelo usuário. **gets()** também retorna um ponteiro para *str*. O protótipo da função é encontrado em **STDIO.H**. O programa seguinte lê uma string para a matriz *str* e escreve seu comprimento.

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

void main(void)
{
    char str[80];

    gets(str);
    printf("o comprimento é %d", strlen(str));
}
```

A função **puts()** escreve seu argumento string na tela seguido por uma nova linha. Seu protótipo é

```
int puts(const char *str);
```

**puts()** reconhece os mesmos códigos de barra invertida de **printf()**, como `'\t'` para uma tabulação. Uma chamada a **puts()** requer bem menos tempo do que a mesma chamada a **printf()** porque **puts()** pode escrever apenas strings de caractere — não pode escrever números ou fazer conversões de formato. Portanto, **puts()** ocupa menos espaço e é executada mais rapidamente que **printf()**. Por essa razão, a função **puts()** é freqüentemente utilizada quando é importante ter um código altamente otimizado. A função **puts()** devolve EOF se ocorre um erro. Caso contrário, ela devolve um valor diferente de zero. No entanto, quando a escrita é feita no console, pode-se normalmente assumir que não ocorrerá nenhum erro, então o valor de **puts()** raramente é monitorado. O comando seguinte mostra **alo**:

```
puts("alo");
```

A Tabela 8.1 resume as funções mais simples que realizam operações de E/S por meio do console.

**Tabela 8.1** Funções de E/S simples.

Função	Operação
getchar()	Lê um caractere do teclado; espera o retorno de carro.
getche()	Lê um caractere com eco; não espera o retorno de carro; não definida pelo ANSI, mas é uma extensão comum.
getch()	Lê um caractere sem eco; não espera o retorno de carro; não definida pelo ANSI, mas é uma extensão comum.
putchar()	Escreve um caractere na tela.
gets()	Lê uma string do teclado.
puts()	Escreve uma string na tela.

O programa seguinte, um dicionário computadorizado simples, demonstra várias funções básicas de E/S pelo console. Primeiro, é pedido ao usuário que introduza uma palavra e, então, o programa verifica se a palavra coincide com alguma pertencente ao seu banco de dados interno. Se existe alguma palavra igual, o programa escreve o significado da palavra. Preste especial atenção na utilização do operador de indireção utilizada neste programa. Caso você tenha problema em entendê-lo, lembre-se de que a matriz **dic** contém ponteiros para strings. Observe que a lista deve ser terminada por dois nulos.

```
/* Um dicionário simples. */
#include <stdio.h>
#include <conio.h>
#include <string.h>
#include <ctype.h>

/* lista de palavras e significados */
char *dic[][40] = {
    "atlas", "um livro de mapas",
    "carro", "um veículo motorizado",
    "telefone", "um dispositivo de comunicação",
    "avião", "uma máquina voadora",
    "", "" /* nulo termina a lista */
};

void main(void)
{
    char word[80], ch;
    char **p;
    do {
        puts("\nEntre a palavra: ");
        gets(word);

        p = (char **)dic;

        /* encontra a palavra e imprime seu significado */
        do {
            if(!strcmp(*p, word)) {
                puts("significado:");
                puts(*(p+1));
                break;
            }
            if(!strcmp(*p, word)) break;
            p = p + 2; /* avança na lista */
        } while(*p);
        if(!*p) puts("a palavra não está no dicionário");
        printf("outra? (y/n): ");
        ch = getche();
    } while(toupper(ch) != 'N');
}
```

## E/S Formatada pelo Console

As funções `printf()` e `scanf()` realizam entrada e saída formatada — isto é, elas podem ler e escrever dados em vários formatos que estão sob seu controle.

A função **printf()** escreve dados no vídeo. A função **scanf()**, seu complemento, lê dados do teclado. As duas funções podem operar em qualquer dos tipos de dados intrínsecos, incluindo caracteres, strings e números.

## **printf()**

O protótipo para **printf()** é

```
int printf(const char *string_de_controle, ...);
```

O protótipo para **printf()** está em **STDIO.H**. A função **printf()** devolverá o número de caracteres escritos ou um valor negativo, se ocorrer um erro.

A *string\_de\_controle* consiste em dois tipos de itens. O primeiro tipo é formado por caracteres que serão impressos na tela. O segundo contém comandos de formato que definem a maneira pela qual os argumentos subsequentes serão mostrados. Um comando de formato começa com um símbolo percentual (%) e é seguido pelo código do formato. Deve haver o mesmo número de argumentos e de comandos de formato e estes dois são combinados na ordem, da esquerda para a direita. Por exemplo, a seguinte chamada a **printf()**.

**Tabela 8.2** Comandos de formato de **printf()**.

<b>Código</b>	<b>Formato</b>
%c	Caractere
%d	Inteiros decimais com sinal
%i	Inteiros decimais com sinal
%e	Notação científica (e minúsculo)
%E	Notação científica (E maiúsculo)
%f	Ponto flutuante decimal
%g	Usa %e ou %f, o que for mais curto
%G	Usa %E ou %F, o que for mais curto
%o	Octal sem sinal
%s	String de caracteres
%u	Inteiros decimais sem sinal
%x	Hexadecimal sem sinal (letras minúsculas)
%X	Hexadecimal sem sinal (letras maiúsculas)
%p	Apresenta um ponteiro
%n	O argumento associado é um ponteiro para inteiro no qual o número de caracteres escritos até esse ponto é colocado
%%	Escreve o símbolo %

```
printf("Eu gosto de %s e de %c", "muito!");
```

mostra

```
Eu gosto muito de C!
```

A função **printf()** aceita uma ampla variedade de comandos de formato, como mostrado na Tabela 8.2.

## Escrevendo Caracteres

Para escrever um caractere individual, deve-se utilizar **%c**. Isso faz com que o seu argumento associado seja escrito sem modificações na tela. Para escrever uma string, deve-se utilizar **%s**.

## Escrevendo Números

Pode-se utilizar tanto **%d** quanto **%i** para indicar um número decimal com sinal. Esses comandos de formato são equivalentes; ambos são suportados por razões históricas.

Para escrever um valor sem sinal, deve-se utilizar **%u**.

O especificador de formato **%f** apresenta os números em ponto flutuante.

Os comandos **%e** e **%E** instruem **printf()** a mostrar um argumento **double** em notação científica. Os números representados em notação científica tomam esta forma geral:

x.dddddE+/-yy

A letra “E” maiúscula pode ser mostrada pelo uso do especificador de formato **%e**; para a letra “e” minúscula, utilize **%e**.

Pode-se fazer com que **printf()** decida utilizar **%f** ou **%e** usando os comandos de formato **%g** ou **%G**. Isso faz com que **printf()** selecione o especificador de formato que produz a saída mais curta. Onde aplicável, deve-se utilizar **%G** para “E” mostrado em maiúscula e **%g** para “e” em minúscula. O programa seguinte demonstra o efeito do especificador de formato **%g**.

```
#include <stdio.h>

void main(void)
{
    double f;
```



```
for(f=1.0; f<1.0e+10; f=f*10)
    printf("%g ", f);
}
```

A seguinte saída é produzida:

```
1 10 100 1000 10000 100000 1e+06 1e+07 1e+08 1e+09
```

Podem-se apresentar inteiros sem sinal no formato octal ou hexadecimal utilizando `%o` ou `%x`, respectivamente. Como o sistema de número hexadecimal utiliza as letras de “A” a “E” para representar os números de 10 a 15, essas letras podem ser mostradas em maiúsculas ou em minúsculas. Para maiúsculas, utilize o especificador de formato `%X` e, para minúsculas, utilize `%x`, como mostrado aqui:

```
#include <stdio.h>

void main(void)
{
    unsigned num;

    for(num=0; num<255; num++) {
        printf("%o ", num);
        printf("%x ", num);
        printf("%X\n", num);
    }
}
```

## Mostrando um Endereço

Para mostrar um endereço, deve-se utilizar `%p`. Esse especificador de formato faz com que `printf()` mostre um endereço de máquina em um formato compatível com o tipo de endereçamento utilizado pelo computador. O próximo programa mostra o endereço de **sample**:

```
#include <stdio.h>

int sample;

void main(void)
{
    printf("%p", &sample);
}
```

## O Especificador %n

O especificador de formato **%n** é diferente de todos os outros. Em lugar de dizer a **printf()** para mostrar alguma coisa, ele faz com que **printf()** carregue a variável apontada por seu argumento correspondente com um valor igual ao número de caracteres que já foram escritos. Em outras palavras, o valor que corresponde ao especificador de formato deve ser um ponteiro para uma variável. Depois da chamada a **printf()**, essa variável contém o número de caracteres escritos até o ponto em que o **%n** foi encontrado. Examine esse programa para entender esse código de formato um tanto incomum.

```
#include <stdio.h>

void main(void)
{
    int count;

    printf("isso%n é um teste\n", &count);
    printf("%d", count);
}
```

Esse programa mostra **isso é um teste** seguido pelo número 4. O especificador de formato **%n** é utilizado fundamentalmente em formatação dinâmica.

## Modificadores de Formato

Muitos comandos de formatos podem ter modificadores que alteram ligeiramente seus significados. Por exemplo, pode-se especificar uma largura mínima de campo, o número de casas decimais e justificação à esquerda. O modificador de formato fica entre o sinal de percentagem e o código propriamente dito.

## O Especificador de Largura Mínima de Campo

Um número colocado entre o símbolo **%** e o código de formato age como um *especificador de largura mínima de campo*. Isso preenche a saída com espaços, para assegurar que ela atinja um certo comprimento mínimo. Se a string, ou o número, for maior do que o mínimo, ela será escrita por inteiro. O preenchimento padrão é feito com espaços. Para preencher com 0s, deve-se colocar um 0 antes do especificador de largura mínima de campo. Por exemplo, **%05d** preencherá um número de menos de cinco dígitos com 0s de forma que seu comprimento total seja cinco. O programa seguinte demonstra o especificador de largura mínima de campo.

```
#include <stdio.h>

void main(void)
{
    double item;

    item = 10.12304;

    printf("%f\n", item);
    printf("%10f\n", item);
    printf("%012f\n", item);
}
```

Esse programa produz a seguinte saída:

```
10.123040
10.123040
00010.123040
```

O modificador de largura mínima de campo é normalmente utilizado para produzir tabelas em que as colunas são alinhadas. Por exemplo, o próximo programa produz uma tabela com os quadrados e os cubos dos números de 1 a 19.

```
#include <stdio.h>

void main(void)
{
    int i;

    /* mostra uma tabela de quadrados e cubos */
    for(i=1; i<20; i++)
        printf("%8d %8d %8d\n", i, i*i, i*i*i);
}
```

Uma amostra de sua saída é vista a seguir:

1	1	1
2	4	8
3	9	27
4	16	64
5	25	125
6	36	216
7	49	343
8	64	512
9	81	729
10	100	1000

11	121	1331
12	144	1728
13	169	2197
14	196	2744
15	225	3375
16	256	4096
17	289	4913
18	324	5832
19	361	6859

## O Especificador de Precisão

O *especificador de precisão* segue o especificador de largura mínima de campo (se houver algum), consistindo em um ponto seguido de um número inteiro. O seu significado exato depende do tipo de dado a que está sendo aplicado.

Quando se aplica o especificador de precisão a dados em ponto flutuante, ele determina o número de casas decimais mostrado. Por exemplo, `%10.4f` mostra um número com pelo menos dez caracteres com quatro casas decimais.

Quando o especificador de precisão é aplicado a `%g` ou `%G`, ele determina a quantidade de dígitos significativos.

Aplicado a strings, o especificador de precisão determina o comprimento máximo do campo. Por exemplo, `%5.7s` mostra uma string de pelo menos cinco e não excedendo sete caracteres. Se a string for maior que a largura máxima do campo, os caracteres finais são truncados.

Quando aplicado a tipos inteiros, o especificador de precisão determina o número mínimo de dígitos que aparecerão para cada número. Zeros iniciais serão adicionados para completar o número solicitado de dígitos.

O programa seguinte ilustra o especificador de precisão:

```
#include <stdio.h>

void main(void)
{
    printf("%.4f\n", 123.1234567);
    printf("%3.8d\n", 1000);
    printf("%10.15s\n", "Esse é um teste simples.");
}
```

Ele produz a seguinte saída:

```
123.1235
00001000
Esse é um teste
```

## Justificando a Saída

Por padrão, toda saída é justificada à direita. Isso é, se a largura do campo for maior que os dados escritos, os dados serão colocados na extremidade direita do campo. A saída pode ser justificada à esquerda colocando-se um sinal de subtração imediatamente após o %. Por exemplo, `%-10.2f` justifica à esquerda um número em ponto flutuante com duas casas decimais em um campo de 10 caracteres.

O programa seguinte ilustra a justificação à esquerda:

```
#include <stdio.h>

void main(void)
{
    printf("justificado à direita:%8d\n", 100);
    printf("justificado à esquerda:%-8d\n",100);
}
```

## Manipulando Outros Tipos de Dados

Existem dois modificadores dos comandos de formato que permitem que `printf()` mostre inteiros curtos e longos. Esses modificadores podem ser aplicados aos comandos de tipo `d`, `i`, `o`, `u` e `x`. O modificador `l` diz a `printf()` que segue um tipo de dado **long**. Por exemplo, `%ld` significa que um **long int** será mostrado. O modificador `h` instrui `printf()` a mostrar um **short int**. Por exemplo, `%hu` indica que o dado é do tipo **short unsigned int**.

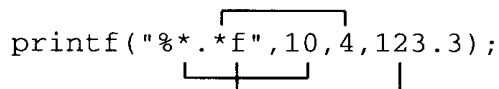
O modificador `L` também pode anteceder o especificador de ponto flutuante `e`, `f`, e `g`, e indica que segue um **long double**.

## Os Modificadores \* e #

A função `printf()` suporta dois modificadores adicionais para alguns dos seus comandos de formato: `*` e `#`.

Preceder os comandos `g`, `G`, `f`, `E` ou `e` com `#` garante que haverá um ponto decimal, mesmo que não haja dígitos decimais. Se o especificador de formato `x` ou `X` é precedido por um `#`, o número hexadecimal será escrito com um prefixo `0x`. Se o especificador `o` é precedido de `#`, o número será exibido com um zero à esquerda. O `#` não pode ser aplicado a nenhum outro especificador de formato.

Os comandos de largura mínima de campo e precisão podem ser fornecidos como argumentos de **printf()**, em lugar de constantes. Para que isso seja realizado, deve-se utilizar o **\*** como marcador. Quando a string de formato for examinada, **printf()** fará o **\*** combinar com um argumento na ordem em que ele ocorre. Por exemplo, na Figura 8.1, a largura mínima do campo é 10, a precisão é 4 e o valor a ser mostrado é 123,3.



```
printf("%*. *f", 10, 4, 123.3);
```

**Figura 8.1** Como o **\*** combina seus valores.

O programa seguinte ilustra **#** e **\***:

```
#include <stdio.h>

void main(void)
{
    printf("%x  %#x\n", 10, 10);
    printf("%*. *f", 10, 4, 1234.34);
}
```

## scanf()

**scanf()** é a rotina de entrada pelo console de uso geral. Ela pode ler todos os tipos de dados intrínsecos e converte automaticamente números ao formato interno apropriado. Ela é muito parecida com o inverso de **printf()**. O protótipo para **scanf()** é

```
int scanf(const char *string_de_controle, ...;
```

O protótipo para **scanf()** está em **STDIO.H**. A função **scanf()** devolve o número de itens de dados que foi atribuído, com êxito, a um valor. Se ocorre um erro, **scanf()** devolve **EOF**. A *string\_de\_controle* determina como os valores são lidos para as variáveis apontadas na lista de argumentos.

A string de controle consiste em três classificações de caracteres.

- Especificadores de formato
- Caracteres de espaço em branco
- Caracteres de espaço não-branco

Vamos olhar cada um deles agora.

## Especificadores de Formato

Os especificadores de formato de entrada são precedidos por um sinal % e informam a `scanf()` que tipo de dado deve ser lido imediatamente após. Esses códigos estão listados na Tabela 8.3. Os especificadores de formato coincidem, na ordem da esquerda para a direita, com os argumentos na lista de argumentos. Vejamos alguns exemplos.

**Tabela 8.3** Especificadores de formato de `scanf()`.

Código	Significado
%c	Lê um único caractere
%d	Lê um inteiro decimal
%i	Lê um inteiro decimal
%e	Lê um número em ponto flutuante
%f	Lê um número em ponto flutuante
%g	Lê um número em ponto flutuante
%o	Lê um número octal
%s	Lê uma string
%x	Lê um número hexadecimal
%p	Lê um ponteiro
%n	Recebe um valor inteiro igual ao número de caracteres lidos até então
%u	Lê um inteiro sem sinal
%[]	Busca por um conjunto de caracteres.

## Inserindo Números

Para ler um número decimal, deve-se utilizar os especificadores %d ou %i. (Esses especificadores, que fazem exatamente a mesma coisa, são incluídos para compatibilidade com versões mais antigas de C.)

Para ler um número em ponto flutuante, representado em notação científica ou padrão, deve-se utilizar %e, %f ou %g. (Novamente, esses especificadores, que fazem exatamente a mesma coisa, são incluídos para compatibilidade com versões mais antigas de C.)

`scanf()` pode ser utilizada para ler inteiros na forma octal ou hexadecimal usando os comandos de formato %o e %x, respectivamente. O %x pode estar em maiúscula ou minúscula. De qualquer forma, pode-se inserir letra de "A" a "F", em maiúsculas ou minúsculas, quando se estiver digitando números hexadecimais. O programa seguinte lê um número octal e um hexadecimal:

```
#include <stdio.h>

void main(void)
{
    int i, j;

    scanf("%o%x", &i, &j);
    printf("%o %x", i, j);
}
```

A função **scanf()** termina a leitura de um número quando o primeiro caractere não numérico é encontrado.

## Inserindo Inteiros sem Sinal

Para inserir um inteiro sem sinal, deve-se utilizar o especificador de formato **%u**. Por exemplo,

```
unsigned num;
scanf("%u", &num);
```

lê um número sem sinal e coloca-o em **num**.

## Lendo Caracteres Individuais com scanf()

Como você aprendeu anteriormente neste capítulo, é possível ler caracteres individuais utilizando **getchar()** ou uma função derivada. **scanf()** também pode ser utilizada para ler um caractere, usando-se o especificador de formato **%c**. No entanto, como muitas implementações de **getchar()**, **scanf()** guarda a entrada em um buffer quando **%c** é usado. Isto torna-a imprópria para um ambiente interativo.

Embora espaços, tabulações e novas linhas sejam utilizados como separadores de campos quando se lê outros tipos de dados, na leitura de um único caractere, caracteres de espaço em branco são lidos como qualquer outro caractere. Por exemplo, com "x y" como entrada, esse fragmento de código

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

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

## Lendo Strings

A função **scanf()** pode ser utilizada para ler uma string da stream de entrada, usando o especificador de formato **%s**. **%s** faz com que **scanf()** leia caracteres



até que seja encontrado um caractere de espaço em branco. Os caracteres lidos são colocados em uma matriz de caracteres apontada pelo argumento correspondente e o resultado tem terminação nula. Para `scanf()`, um caractere de espaço em branco é um espaço, um retorno de carro ou uma tabulação. Ao contrário de `gets()`, que lê uma string até que seja digitado um retorno de carro, `scanf()` lê a string até o primeiro espaço em branco. Isso significa que `scanf()` não pode ser utilizada para ler uma string como “isto é um teste” porque o primeiro espaço termina o processo de leitura. Para ver o efeito do especificador `%s`, tente esse programa, usando a string “alo aqui”.

```
#include <stdio.h>

void main(void)
{
    char str[80];

    printf("entre com uma string: ");
    scanf("%s", str);
    printf("eis sua string: %s", str);
}
```

O programa responde com apenas a porção “alo” da string.

## Inserindo um Endereço

Para inserir um endereço de memória (ponteiro), deve-se utilizar o especificador de formato `%p`. Não tente usar um inteiro sem sinal ou qualquer outro especificador de formato para inserir um endereço porque `%p` faz com que `scanf()` leia um endereço no formato usado pela CPU. Por exemplo, esse programa lê um ponteiro e, então, mostra o que há neste endereço de memória.

```
#include <stdio.h>

void main(void)
{
    char *p;

    printf("entre com um endereço: ");
    scanf("%p", &p);
    printf("na posição %p há %c\n", p, *p);
}
```

## O Especificador %n

O especificador **%n** instrui **scanf()** a atribuir o número de caracteres lidos da stream de entrada, no ponto em que o **%n** foi encontrado, à variável apontada pelo argumento correspondente.

## Utilizando um Scanset

O padrão C ANSI acrescentou a **scanf()** uma nova característica chamada scanset. Um *scanset* define um conjunto de caracteres que pode ser lido por **scanf()** e atribuído à matriz de caracteres correspondente. Um scanset é definido colocando-se uma string dos caracteres a serem procurados entre colchetes. O colchete deve ser precedido por um sinal percentual. Por exemplo, o seguinte scanset informa a **scanf()** para ler apenas os caracteres X, Y e Z.

```
■ %[XYZ]
```

Quando um scanset é utilizado, **scanf()** continua a ler caracteres e coloca-os na matriz de caracteres correspondente até que encontre um caractere que não pertença ao scanset. A variável correspondente deve ser um ponteiro para uma matriz de caracteres. Ao retornar, essa matriz conterá uma string terminada com um nulo que consiste nos caracteres que foram lidos. Para entender como isso funciona, tente este programa:

```
■ #include <stdio.h>

void main(void)
{
    int i;
    char str[80], str2[80];

    scanf("%d%[abcdefg]%s", &i, str, str2);
    printf("%d %s %s", i, str, str2);
}
```

Digite **123abcdtye** seguido de um ENTER. O programa mostrará, então, **123 abcdtye**. **scanf()** finaliza a leitura de caracteres para **str** quando encontra o “t” porque ele não faz parte do scanset. Os caracteres restantes são colocados em **str2**.

Pode ser especificado um conjunto invertido se o primeiro caractere do conjunto for um **^**. O **^** instrui **scanf()** a aceitar qualquer caractere que *não* está definido pelo scanset.

Também pode ser especificada uma faixa de caracteres, usando-se um hífen entre o caractere inicial e o final. Por exemplo, isso informa a `scanf()` para aceitar os caracteres de A a Z:

```
■ %[A-Z]
```

Um ponto importante que deve ser lembrado é que um `scanfset` diferencia maiúsculas de minúsculas. Para fazer uma busca tanto entre maiúsculas quanto minúsculas, você deve especificá-las individualmente.

## Descartando Espaços em Branco Indesejados

Um caractere de espaço em branco na string de controle faz com que `scanf()` salte um ou mais caracteres de espaço em branco da stream de entrada. Um caractere de espaço em branco é um espaço, uma tabulação ou uma nova linha. Em essência, um caractere de espaço em branco, na string de controle, faz com que `scanf()` leia, mas não armazene, qualquer número (incluindo zero) de caracteres de espaço em branco até que seja encontrado o primeiro caractere de espaço não-branco.

## Caracteres de Espaço Não-Branco na String de Controle

Um caractere de espaço não-branco na string de controle faz com que `scanf()` leia e ignore caracteres iguais na stream de entrada. Por exemplo, “`%d,%d`” faz com que `scanf()` leia um inteiro, leia e descarte uma vírgula e, então, leia outro inteiro. Se o caractere especificado não for encontrado, `scanf()` termina. Para descartar um sinal percentual, é usado `%%` na string de controle.

## Deve-se Passar Endereços para `scanf()`

Todas as variáveis utilizadas para receber valores por meio de `scanf()` devem ser passadas pelos seus endereços. Isso significa que todos os argumentos devem ser ponteiros para as variáveis usadas como argumentos. Recorde que essa é a maneira de C criar uma chamada por referência, o que permite a uma função alterar o conteúdo de um argumento. Por exemplo, para ler um inteiro para a variável `count`, seria usada a seguinte chamada a `scanf()`:

```
■ scanf ("%d", &count);
```

As strings são lidas para matrizes de caracteres e o nome da matriz, sem qualquer índice, é o endereço do primeiro elemento da matriz. Logo, para ler uma string para a matriz de caracteres **str**, seria usado

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

Nesse caso, **str** já é um ponteiro e não precisa ser precedido pelo operador **&**.

## Modificadores de Formato

Tal como ocorre com **printf()**, **scanf()** permite que alguns dos seus especificadores de formato sejam modificados.

Os especificadores de fomato podem determinar um modificador de largura máxima de campo. Isto é, um inteiro, colocado entre o % e o formato, limita o número de caracteres lido para aquele campo. Por exemplo, para ler até 20 caracteres para **str**, escreva

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

Se a stream de entrada for maior do que 20 caracteres, uma chamada subsequente a qualquer função de entrada de caracteres começará onde essa chamada termina. Por exemplo, se for entrado

ABCDEFGHIJKLMNOPQRSTUVWXYZ

como resposta à chamada a **scanf()** nesse exemplo, apenas os 20 primeiros caracteres, ou até o T, serão colocados em **str**, devido ao especificador de máximo tamanho. Isso significa que os caracteres restantes, UVWXYZ, ainda não foram usados. Se alguma outra chamada a **scanf()** for feita, como em

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

as letras UVWXYZ serão colocadas em **str**. A entrada para um campo pode terminar antes que o comprimento máximo seja atingido se for encontrado um espaço em branco. Nesse caso, **scanf()** avança para o próximo campo.

Para ler um inteiro longo, um **l** (letra ele) deve ser colocado antes do especificador de formato. Para ler um inteiro curto, um **h** deve ser colocado antes do especificador de formato. Esses modificadores podem ser usados com os formatos **d**, **i**, **o**, **u** e **x**.

Por padrão, os especificadores **f**, **e** e **g** instruem **scanf()** a atribuir dados a um **float**. Com um **l** antes de um desses especificadores, **scanf()** atribui o dado a um **double**. Para que **scanf()** receba um **long double**, utiliza-se um **L**.

## Suprimindo a Entrada

Pode-se informar a **scanf()** para ler um campo, mas não atribuí-lo a nenhuma variável, através da colocação de um **\*** precedendo o código do formato do campo. Por exemplo, dado

```
scanf ("%d%c%d", &x, &y);
```

you pode inserir **10,10**. A vírgula seria lida corretamente, mas não seria atribuída a nada. A supressão de atribuição é útil especialmente quando só é necessário processar uma parte do que está sendo digitado.