MAKRON Books

Parte 2

A Biblioteca C Padrão

A Parte 2 deste livro examina a biblioteca C padrão. O Capítulo 11 discute linkedição, bibliotecas e arquivos de cabeçalho. Dos Capítulos 12 até 18 são descritas as funções encontradas na biblioteca padrão — cada capítulo se concentra em um grupo específico de funções.

A Parte 2 discute três tipos de funções. Primeiro, ela inclui todas as funções da biblioteca como definidas pelo padrão C ANSI. Essas funções estão incluídas em todas as implementações de C padrão e o código que as utiliza é portável para todos os ambientes. A segunda categoria de funções são aquelas definidas pela versão original de C para UNIX. Aqui são descritas várias das funções baseadas em UNIX mais usadas. Por fim, embora não exista padrão para imagens, para controle de tela ou para as funções de interface com o sistema operacional, elas estão incluídas em virtualmente todas as implementações de C. Por esta razão, examinamos uma amostra representativa.

Ao explorar a biblioteca padrão lembre-se disto: a maioria dos desenvolvedores de compilador tem bastante orgulho da perfeição da sua biblioteca. Sua biblioteca de compilador provavelmente conterá muitas funções além daquelas descritas aqui. Portanto, é sempre uma boa idéia procurar o manual do usuário.

11

Linkedição, Bibliotecas e Arquivos de Cabeçalho

Quando se escreve um compilador em C, há na verdade dois trabalhos. Primeiro é necessário criar o próprio compilador. Depois, a biblioteca padrão deve ser codificada. O compilador C em si é relativamente fácil de desenvolver. É a biblioteca de funções que leva mais tempo e esforço. Uma razão para isso é que muitas funções (como o sistema de E/S) têm de realizar uma interface com o sistema operacional em que o compilador está sendo escrito. Além disso, a biblioteca C padrão define um conjunto grande e diversificado de funções. Certamente é a riqueza e a flexibilidade da biblioteca padrão que põe C à frente de muitas outras linguagens. Para entender completamente a natureza da biblioteca C padrão e como ela é usada para produzir um programa executável, você deve saber como o linkeditor trabalha, como as bibliotecas diferem de arquivos-objetos e o papel dos arquivos de cabeçalho.

O Linkeditor

O linkeditor tem duas funções. A primeira, como o próprio nome indica, é combinar (ligar) várias partes do código-objeto. A segunda é resolver os endereços das instruções de "jump" (desvio) e "call" (chamada) encontradas em formato relocável no arquivo objeto. Para entender a sua operação, vamos olhar mais de perto o processo de compilação separada.

Compilação Separada

Na *compilação separada*, partes de um programa são compiladas separadamente e depois linkeditadas para formar o programa executável completo. A saída do

compilador é um arquivo-objeto relocável e a saída do linkeditor é um arquivo executável. O papel que o linkeditor cumpre é duplo. Primeiro, ele combina fisicamente os arquivos especificados na lista de linkedição em um arquivo de programa. Segundo, resolve as referências externas. Uma referência externa é criada toda vez que o código em um arquivo refere-se ao código encontrado em outro arquivo. Por exemplo, quando os dois arquivos mostrados aqui forem linkeditados, a referência do arquivo 2 a **count** deve ser resolvida. O linkeditor informa ao código no arquivo 2 onde **count** será encontrada na memória.

Arquivo 1

```
int count;
void display(void);

void main(void)
{
   count = 10;
   display();
}
```

Arquivo 2

```
#include <stdio.h>

extern int count;

void display (void)
{
   printf("%d",count);
}
```

De maneira semelhante, o linkeditor informa ao arquivo 1 onde a função **display()** está localizada para que ela possa ser chamada.

Quando o compilador gera o código-objeto para **display()**, ele substitui por um espaço o endereço de **count**, pois não tem como saber onde **count** está. Algo semelhante ocorre quando **main()** é compilada. O endereço de **display()** é desconhecido, então um espaço é usado. Esse processo forma a base do código relocável.

Código Relocável

Código relocável é simplesmente o código-objeto que pode ser executado em qualquer região de memória disponível e que seja grande o bastante para recebê-lo. Em um arquivo-objeto relocável, o endereço de cada call, jump ou variável global não é fixo, mas relativo. Para entender como o código relocável é criado, você precisa entender o *código absoluto*. Para essa discussão, é necessário trabalhar com as instruções reais de máquina que seriam geradas pelo compilador C, usando um fragmento de código em pseudo-assembly que, de certa forma, lembra o código assembly do 8086. (Não se preocupe se você não sabe linguagem assembly, este exemplo é muito simples.)

O código em pseudo-assembly, mostrado aqui, corresponde ao código em C mostrado após ele:

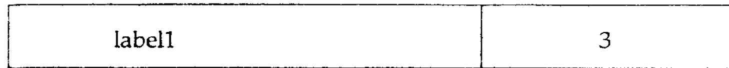

```
         mov a,0
label1:  out a, 123   ;envia-o à porta 123
         inc a        ;incrementa o registrador a
         cmp a,100    ;é igual a 100?
         out 123      ;saída para a porta 123
         jnz label1   ;salta se não zero

/* versão do código em C */
for (x=0; x<100; x++) out (x,123);
```

Nesse exemplo, o registrador **a** é inicializado com zero. Em seguida, o programa repete o laço 100 vezes, cada vez enviando o valor do registrador **a** à porta 123. A instrução **inc** incrementa o registrador **a** e **cmp** compara-o com o valor 100. Se o registrador **a** é igual a 100, o flag de zero é ativado. A instrução **jnz** é lida "pule se não zero" e significa pular para o endereço especificado se o flag de zero não está ativo. A função **out()**, na versão do código em C, é assumida como sendo traduzida para a instrução **out** correspondente no código em pseudo-assembly.

Se um montador absoluto é utilizado, o código é montado para ser executado em alguma posição absoluta especificada. Por exemplo, assuma que a instrução **mov** requer 3 bytes. Se o código fosse montado para ser executado no endereço 100, o rótulo associado à instrução **jnz** seria substituído pelo endereço 103. Isso também implica que o código-objeto desse programa só é executado corretamente se for colocado na memória começando em 100. Ele não pode ser carregado em nenhum outro lugar.

Endereçamento absoluto é razoavelmente comum em sistemas simples de microcomputador. Porém, é inadequado para o uso geral porque exige que a posição na memória em que o programa será executado seja conhecida no momento em que é montado. Isso não somente evita que o código seja linkeditado com outro código posteriormente como também priva o sistema operacional de fazer um uso mais eficiente dos seus recursos. Por essas razões, o código-objeto relocável é utilizado.

O código relocável é criado usando-se um montador *relocável*. Ele não usa endereços fixos, apenas deslocamentos a partir do início do arquivo. (Um *deslocamento* representa o número de bytes em que um rótulo está a partir do início do arquivo.) Os deslocamentos são armazenados pelo montador em uma tabela juntamente com os nomes dos rótulos. Essa *tabela* é denominada *tabela de deslocamento*. A tabela de deslocamento é, então, utilizada durante o processo de linkedição para determinar os endereços reais. Usando o exemplo do código assembly anterior, e assumindo que cada instrução requer 3 bytes, a entrada na tabela para **label1** é algo parecido com a seguinte ilustração.



Quando o linkeditor liga o fragmento de código contendo **label1** ao resto do código que compõe o programa, o offset 3 é adicionado ao valor atual do contador de posição para calcular o valor correto a ser usado na instrução **jnz**. Por exemplo, se o contador de posição for 230 quando o rótulo for resolvido, o valor 233 se tornará o endereço utilizado na instrução **jnz**. Essa discussão sobre o processo de relocação é, obviamente, uma abstração. Muito embora os princípios gerais estejam corretos, a implementação real variará amplamente entre ambientes.

A relocação de endereço é necessária para todos os tipos de instrução de desvio, como também para as instruções de chamada, porque usam endereços de memória. Além disso, todo dado global requer relocação.

Muito embora você venha a trabalhar com C e não assembler, o processo de relocação é manipulado basicamente da mesma forma.

Linkeditando com Overlays

Muitos fabricantes de compiladores C fornecem um linkeditor de overlay. Um *linkeditor de overlay* funciona como um linkeditor normal, mas também pode criar overlays. Um *overlay* é uma parte do código-objeto que é armazenada em um arquivo em disco e carregado e executado apenas quando necessário. O espaço na memória em que um overlay é carregado é chamado de *região de overlay*. Overlays permitem que se criem e executem programas que são maiores que a memória disponível, pois apenas as partes do programa atualmente em uso estão na memória.

Para entender como os overlays funcionam, imagine que você tenha um programa composto de sete arquivos-objetos chamados de F1 até F7. Assuma, também, que não haveria memória livre suficiente para executar o programa se os arquivos-objetos fossem linkeditados todos juntos na forma normal — você só poderia linkeditar os cinco primeiros arquivos antes de ficar sem memória. Para remediar essa situação, faça o linkeditor criar overlays consistindo nos arquivos F5, F6 e F7. Toda vez que uma função em um desses arquivos for chamada, o *gerenciador de overlay* (fornecido pelo linkeditor) encontrará o arquivo apropriado e o colocará na região de overlay, continuando a execução. Os códigos dos arquivos de F1 a F4 permanecem residentes todo o tempo. A Figura 11.1 ilustra essa situação.

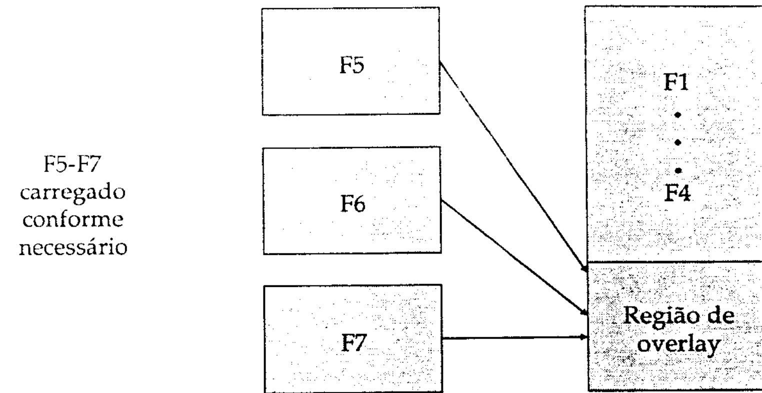


Figura 11.1 Um programa com overlays na memória.

Como você pode imaginar, a principal vantagem dos overlays é permitir que você escreva programas muito grandes. A principal desvantagem — e a razão pela qual overlays são o último recurso — é que o processo de carga toma tempo e tem impacto significativo na velocidade geral da execução do programa. Por essa razão, você deve agrupar funções relacionadas, se tiver de utilizar overlays, para que o número de cargas dos módulos seja minimizado. Por exemplo, se a aplicação é uma lista postal, faz sentido colocar todas as rotinas de ordenação em um overlay, rotinas de impressão em outro etc.

Algumas vezes, uma alternativa melhor para os overlays é o *encadeamento*. Pelo método de encadeamento, um programa instrui o sistema operacional a carregar e executar um outro programa. Quando o segundo programa termina, o primeiro volta a ser executado. Pode-se conseguir o encadeamento usando a função **system()** padrão, que é descrita no Capítulo 18.

Linkeditando com DLLs

Com o advento de sistemas operacionais como Windows, uma outra forma de linkedição *chamada de linkedição* dinâmica tornou-se uma parte comum do panorama de C. A linkedição dinâmica é o processo pelo qual o código real para uma função permanece em disco em um arquivo separado até que seja executado um programa que a utiliza. Quando o programa é executado, as funções linkeditadas dinamicamente exigidas pelo programa também são carregadas. As funções linkeditadas dinamicamente residem em um tipo de biblioteca especial denominada *biblioteca de linkedição dinâmica*, ou DLL (*Dynamic Link Library*).

A principal vantagem de se usar bibliotecas linkeditadas dinamicamente é que o tamanho dos programas é reduzido drasticamente porque cada programa não precisa mais incluir cópias redundantes das funções da biblioteca que utiliza. Além disso, quando as funções da DLL são atualizadas, os programas que as usam obterão seus benefícios automaticamente.

Enquanto a biblioteca padrão C geralmente não está contida em uma biblioteca de linkedição dinâmica, muitos outros tipos de funções o estão. Por exemplo, quando você programa para Windows, todo o conjunto de funções de API (*Application Program Interface*) é armazenado em DLLs. Afortunadamente, em relação a seu programa C, não faz diferença se uma função de biblioteca está armazenada em uma DLL ou em um arquivo de biblioteca normal.

A Biblioteca C Padrão

O padrão ANSI definiu o conteúdo e a forma da biblioteca C padrão. Ou seja, o padrão C ANSI nomeia e descreve um conjunto de funções que todos os compiladores ANSI padrão devem suportar. Contudo, muitos compiladores fornecem funções adicionais não especificadas pelo padrão. Por exemplo, é comum um compilador ter funções gráficas, rotinas manipuladoras de mouse etc. Caso você não vá transportar seu programa para um novo ambiente, pode usar essas funções fora do padrão. Porém, se seu código precisa ser portável, o uso dessas funções deve ser limitado. De um ponto de vista prático, virtualmente todos os programas C raros farão você utilizar funções fora do padrão, então você não deverá necessariamente evitar a utilização dele somente porque não fazem parte da função padrão.

Para muitos compiladores C, a biblioteca padrão está contida em um arquivo. Porém, algumas implementações têm as funções relacionadas agrupadas em suas próprias bibliotecas por eficiência ou devido a restrições de tamanho.

Arquivos de Biblioteca *Versus* Arquivos-Objetos

Embora bibliotecas e arquivos-objetos sejam semelhantes, eles têm uma diferença crucial: apenas parte do código na biblioteca é acrescentado ao seu programa. Quando você linkedita um programa que consiste em diversos arquivos-objetos, todo o código, em cada arquivo-objeto, torna-se parte do programa executável pronto. Isso acontece se o código é realmente utilizado ou não. Em outras palavras, todos os arquivos-objetos especificados no momento da linkedição são combinados para formar o programa. No entanto, isso não acontece com arquivos de biblioteca.

Uma biblioteca é uma coleção de funções. Ao contrário de um arquivo-objeto, um arquivo de biblioteca armazena o nome de cada função, o código-objeto da função, mais informações necessárias para o processo de linkedição. Quando seu programa referencia uma função contida em uma biblioteca, o linkeditor procura essa função e adiciona esse código ao seu programa. Dessa forma, apenas as funções que você realmente usa em seu programa são acrescentadas ao arquivo executável. Como as funções são adicionadas seletivamente ao seu programa, quando uma biblioteca é usada, as funções de biblioteca C padrão estão contidas em uma biblioteca em lugar de em arquivos-objetos. (Se estivessem em arquivos-objetos, todo programa que você escrevesse teria centenas de milhares de bytes!)

Arquivos de Cabeçalho

Cada função definida na biblioteca C padrão tem um arquivo de cabeçalho associada a ela. Os arquivos de cabeçalhos que relacionam as funções que você utiliza em seus programas devem estar incluídos (usando **#include**) em seu programa. Há duas razões para isto. Primeiro, muitas funções da biblioteca padrão trabalham com seus próprios tipos de dados específicos, aos quais seu programa deve ter acesso. Esses tipos de dados são definidos em *arquivos de cabeçalho* fornecidos para cada função. Um dos exemplos mais comuns é o arquivo STDIO.H, que fornece o tipo FILE necessário para operações com arquivos em disco.

A segunda razão para incluir arquivos de cabeçalho é obter os protótipos da biblioteca padrão. Se os arquivos de cabeçalho seguem o padrão C ANSI, eles também contêm os protótipos completos para as funções relacionadas com o arquivo de cabeçalho. Isso fornece um método de verificação mais forte que aquele anteriormente disponível ao programador C. Incluindo os arquivos de cabeçalho que correspondem às funções padrões utilizadas pelo seu programa, você pode encontrar erros potenciais de inconsistências de tipos. Por exemplo, o código a seguir, uma vez que inclui STRING.H (o cabeçalho para as funções de string), produz uma mensagem de advertência quando compilado:

```
#include <string.h>

char s1[]="alo ";
char s2[]="aqui";

void main(void)
{
   int p;
```

```
p = strcat(s1, s2);
```

Já que **strcat()** é declarada como retornando um ponteiro para caracteres, o compilador pode, agora, emitir um aviso de que um erro pode ter sido cometido na linha que atribui um inteiro a **p**.

A Tabela 11.1 mostra os arquivos de cabeçalho padrão definidos pelo padrão C ANSI.

Tabela 11.1 Os arquivos de cabeçalho padrões.

Arquivo de cabeçalho	Finalidades
ASSERT.H	Define a macro assert()
CTYPE.H	Manipulação de caracteres
ERRNO.H	Apresentação de erros
FLOAT.H	Define valores em ponto flutuante dependentes da implementação
LIMITS.H	Define limites em ponto flutuante dependentes da implementação
LOCALE.H	Suporta localização
MATH.H	Diversas definições usadas pela biblioteca de matemática
SETJMP.H	Suporta desvios não-locais
SIGNAL.H	Suporta manipulação de sinal
STDARG.H	Suporta listas de argumentos de comprimento variável
STDDEF.H	Define algumas constantes normalmente usadas
STDIO.H	Suporta E/S com arquivos
STDLIB.H	Declarações miscelâneas
STRING.H	Suporta funções de strings
TIME.H	Suporta as funções de horário do sistema

O padrão C ANSI reserva nomes de identificadores, começando com um caractere de sublinhado e seguido por um sublinhado ou uma letra maiúscula para uso em arquivos de cabeçalho.

Os demais capítulos deste livro, que descrevem cada função da biblioteca padrão, indicarão quais destes arquivos de cabeçalho serão necessários para cada função.

Macros em Arquivos de Cabeçalho

Muitas das funções padrões de C são, de fato, definições de macro contidas em um arquivo de cabeçalho. Por exemplo, **abs()**, que devolve o valor absoluto de seu argumento inteiro, poderia ser definida como uma macro, como mostrado aqui:


```
#define abs(i)  (i)<0 ? -(i) : (i)
```

Geralmente, não é importante se uma função padrão é definida como uma macro ou como uma função normal de C. Porém, em raras situações onde isso é inaceitável — por exemplo, onde o tamanho do código deve ser minimizado —, você tem de criar uma função real e substituir a macro. Algumas vezes a própria biblioteca de C tem uma função real que pode ser usada para substituir a macro.

Para forçar o compilador a usar a função real (da biblioteca ou escrita por você), é necessário impedi-lo de substituir a macro quando o nome da função for encontrado. Embora existam várias maneiras de fazer isso, a melhor é simplesmente retirar a definição do nome da macro usando **#undef**. Por exemplo, para forçar o compilador a substituir a função real em lugar da macro definida anteriormente, você deveria inserir essa linha de código próximo ao início de seu programa.

```
#undef abs
```

Assim, já que **abs** não é mais definida como uma macro, a versão da função é usada.

Redefinição das Funções da Biblioteca

Embora os linkeditores possam variar ligeiramente entre implementações, todos eles operam essencialmente da mesma forma. Por exemplo, se seu programa consiste em três arquivos, chamados F1, F2 e F3, a linha de comando para o linkeditor seria algo semelhante a isto:

LINK F1 F2 F3 LIBC

onde LIBC é o nome da biblioteca padrão.

NOTA: Alguns linkeditores usam automaticamente a biblioteca padrão e não requerem que ela seja especificada explicitamente. Além disso, muitos ambientes integrados de programação, como Turbo C e Quick C, automaticamente incluem os arquivos apropriados de biblioteca.

Quando o processo começa, o linkeditor geralmente tenta resolver todas as referências externas usando apenas os arquivos F1, F2 e F3. Uma vez isso feito, é pesquisada na biblioteca a existência de referências externas não resolvidas.

Como a maioria dos linkeditores procede na ordem há pouco descrita, você pode redefinir uma função da biblioteca padrão; a sua função é encontrada primeiro e utilizada para resolver todas as referências a ela. Portanto, no momento em que a biblioteca é examinada, não há referências não resolvidas à função redefinida, e ela não é carregada da biblioteca.

Você deve ser muito cuidadoso quando redefine funções da biblioteca, pois pode estar criando efeitos colaterais inesperados. Isso porque as funções da biblioteca usam outras funções da biblioteca. Por exemplo, a função **fwrite()** poderia usar **putc()**. Assim, uma redefinição de **putc()** poderia fazer com que **fwrite()** se comportasse de forma imprevisível. Uma idéia melhor é simplesmente usar um nome diferente para as funções que você quer redefinir. Isso afasta os efeitos colaterais inesperados.

12

Funções de E/S

A maioria dos compiladores C suporta pelo menos dois sistemas diferentes de E/S: o sistema bufferizado de arquivo definido pelo padrão C ANSI e o sistema de arquivo sem buffer tipo UNIX. Compiladores baseados em DOS geralmente suportam um terceiro sistema de E/S: E/S direto pelo console. Este capítulo descreve todas as funções que são parte do sistema de arquivo do padrão C ANSI, as mais importantes que fazem parte do sistema de arquivo tipo UNIX e diversas funções de E/S, baseadas em DOS, direto pelo console.

NOTA: Compiladores projetados para sistemas operacionais que suportam interfaces gráficas de usuário (GUI), tais como Windows e OS/2, podem fornecer funções de E/S que dizem respeito a esse ambiente específico. No entanto, essas funções não estão no escopo desta obra de referência. Se você estiver interessado em programar para um ambiente GUI, deverá consultar manuais que tratem diretamente desses sistemas operacionais.

As funções que compõem o sistema de entrada/saída do C ANSI podem ser agrupadas em duas categorias principais: E/S pelo console e E/S com arquivo. A E/S pelo console é composta de funções que são versões para o caso específico das funções mais gerais encontradas no sistema de arquivo. No entanto, a E/S pelo console e a E/S com arquivo são diferentes o bastante para que possam ser consideradas em separado. A primeira parte deste livro trata da E/S pelo console e da E/S com arquivo como sendo sistemas de uma certa forma distintos para enfatizar suas diferenças. Todavia, esta seção não faz nenhuma distinção, pois ambos os tipos de E/S usam uma interface lógica comum: a *stream*.

O arquivo de cabeçalho associado às funções de E/S, com buffer definido pelo ANSI, é chamado STDIO.H. Ele define diversas macros e tipos usados

pelo sistema de arquivo. O tipo mais importante é FILE, que é usado para declarar um ponteiro de arquivo. Dois outros tipos são **size_t** e **fpos_t**. O tipo **size_t** equivale essencialmente a *unsigned*. O tipo **fpos_t** define um objeto que pode conter todas as informações necessárias para especificar de forma única qualquer posição dentro de um arquivo. Outros itens definidos por STDIO.H são discutidos quando as funções que o usam são descritas.

Muitas das funções definidas pelo ANSI alteram a variável global inteira **errno** quando ocorre um erro. Seu programa pode verificar essa variável, em caso de erro, para obter maiores informações sobre o erro. Os valores que **errno** pode receber dependem da implementação.

O sistema de E/S tipo UNIX não é definido pelo padrão C ANSI e espera-se que sua popularidade diminua. As funções do sistema de E/S tipo UNIX mais usadas estão incluídas neste capítulo, desde que ainda são amplamente utilizadas em programas já existentes. Para muitos compiladores C, o arquivo de cabeçalho relacionado com o sistema de arquivo tipo UNIX é chamado IO.H. Verifique, porém, seu manual do usuário para detalhes.

Para compiladores baseados em DOS, as funções de E/S direto pelo console geralmente usam o arquivo de cabeçalho CONIO.H. (Verifique no seu manual do usuário por que o arquivo de cabeçalho pode ter um nome diferente.)

Para uma visão geral do sistema de E/S de C e uma discussão detalhada das funções mais importantes de E/S, consulte os Capítulos 8 e 9 na Parte 1.

#include <conio.h>
char *cgets(char *str);

A função **cgets()** não é definida pelo padrão C ANSI. Ela é normalmente incluída na biblioteca de compiladores baseada em DOS.

A função **cgets()** lê uma string digitada no teclado para a matriz apontada por *str*. Antes da chamada a **cgets()**, o primeiro byte de *str* deve ser preenchido com o comprimento máximo da string que você quer ler. No retorno, o segundo byte de *str* conterá o número de caracteres realmente lidos. Portanto, a matriz apontada por *str* deve ser, no mínimo, 2 bytes maior que a string que você quer ler. Quando tiver terminado de inserir a string, introduza um retorno de carro, que será convertido em um nulo para terminar a string. A string começará no terceiro byte da matriz.

A função **cgets()** devolve um ponteiro para **str[2]**.

Em algumas implementações, **cgets()** não permite redirecionamento para outros dispositivos além do teclado. Além disso, **cgets()** pode operar relativamente a uma janela em lugar da tela. Verifique seu manual do usuário para detalhes.

Exemplo

Este programa lê uma string do teclado de até 20 caracteres:

```
#include <conio.h>

void main(void)
{
  char s[23];

  s[0] = 20;
  cputs("digite uma string");
  cgets(s);
  cputs(&s[2]);
}
```

Funções Relacionadas

cputs(), gets(), fgets(), puts()

#include <stdio.h>
void clearerr(FILE *stream);

A função **clearerr()** coloca em zero (desligado) o indicador de erro para o arquivo apontado por *stream*. O indicador de fim de arquivo também é restituído.

Os indicadores de erro para cada *stream* são inicialmente colocados em zero por uma chamada bem-sucedida a **fopen()**. Uma vez que tenha ocorrido um erro, os indicadores permanecem ativos até que seja feita uma chamada explícita a **clearerr()** ou **rewind()**.

Erros com arquivos podem ocorrer por uma ampla variedade de razões, muitas das quais são dependentes do sistema. Você pode determinar a natureza exata do erro chamando **perror()**, que mostra que erro ocorreu (veja **perror()**).

Exemplo

Este programa copia um arquivo para outro. Se for encontrado um erro, será apresentada uma mensagem e o erro será limpo.

```
/* Copia um arquivo em outro */
#include <stdio.h>
#include <stdlib.h>

void main(int argc, char *argv[]);
{
  FILE *in, *out;
  char ch;

  if(argc!=3) {
    printf("Voce esqueceu de digitar o nome do arquivo.\n");
    exit(1);
  }

  if((in=fopen(argv[1], "rb")) == NULL) {
    printf("O arquivo não pode ser aberto.\n");
    exit(1);
  }
  if((out=fopen(argv[2], "wb")) == NULL) {
    printf("O arquivo não pode ser aberto.\n");
    exit(1);
  }

  while(!feof(in)) {
    ch = getc(in);
    if(ferror(in)) {
      printf("Erro de leitura");
      clearerr(in);
      break;
    } else {
      if (!feof (in)) putc(ch, out);
      if(ferror(out)) {
        printf("Erro de escrita");
        clearerr(out);
        break;
      }
    }
  }
  fclose(in);
  fclose(out);
}
```

Funções Relacionadas

feof(), ferror() e perror()

#include <io.h>
int close(int fd);

A função **close()** pertence ao sistema de arquivo tipo UNIX e não é definida pelo padrão C ANSI.

Quando **close()** é chamada com um descritor válido de arquivo, ela fecha o arquivo associado e esvazia os buffers de gravação, se necessário. (Descritores de arquivo são criados mediante uma chamada bem-sucedida a **open()** ou **creat()** e não têm relação com streams ou ponteiros de arquivo.)

Quando bem-sucedida, **close()** devolve zero, caso contrário, devolve -1. Há diversas razões pelas quais não se pode fechar um arquivo. A mais comum é a remoção prematura da mídia. Por exemplo, se você remove um disco do acionador antes que o arquivo seja fechado, ocorre um erro.

Exemplo

Este programa abre e fecha um arquivo usando o sistema de arquivo tipo UNIX.

```
#include <fnctl.h>
#include <io.h>
#include <stdio.h>
#include <stdlib.h>

void main(int argc, char *argv[])
{
  int fd;
  if((fd=open(argv[1], O_RDONLY))==-1) {
    printf("O arquivo não pode ser aberto.");
    exit(1);
  }

  printf("O arquivo já existe.\n");
  if(close(fd)) printf("Erro no fechamento do arquivo.\n");
}
```

Funções Relacionadas

open(), creat(), read(), write(), unlink()

#include <conio.h>
int cprintf(const char *format,...);

A função **cprintf()** não é definida pelo padrão C ANSI. Ela é normalmente incluída na biblioteca de compiladores baseada em DOS.

A função **cprintf()** opera exatamente como **printf()**, mas, em muitas implementações, a sua saída não pode ser redirecionada para outros dispositivos. Além disso, em alguns ambientes, **cprintf()** opera relativamente a uma janela em lugar da tela. Verifique seu manual do usuário para detalhes.

Exemplo

Este programa mostra a string **Eu gosto de C** na tela:

```
#include <conio.h>

void main(void)
{
  cprintf("Eu gosto de C");
}
```

Funções Relacionadas

cscanf(), cputs()

#include <conio.h>
int cputs(const char *str);

A função **cputs()** não é definida pelo padrão C ANSI. Ela é normalmente incluída na biblioteca de compiladores baseada em DOS.

A função **cputs()** escreve na tela a string apontada por *str*. Em algumas implementações, a sua saída não pode ser redirecionada. Além disso, para alguns ambientes, **cputs()** pode escrever sua string relativa a uma janela em lugar da tela. Consulte seu manual do usuário para detalhes.

cputs() devolve o último caractere escrito, caso seja bem-sucedida, e devolve EOF se não obteve sucesso.

Exemplo

Este programa escreve **isto é um teste** na tela:


```
#include <conio.h>

void main(void)
{
  cputs("isto é um teste");
}
```

Função Relacionada

cprintf()

#include <io.h>
int creat(const char *filename, int pmode);

A função **creat()** é parte do sistema de arquivo tipo UNIX e não é definida pelo padrão C ANSI. Sua finalidade é criar um arquivo novo com o nome apontado por *filename* e abri-lo para escrita. Em caso de sucesso, **creat()** devolve um descritor que é maior ou igual a zero; se há falha, ela devolve -1. (Descritores de arquivo são inteiros e não têm relação com streams e ponteiros de arquivo.)

O valor de *pmode* estabelece o tipo de acesso ao arquivo, algumas vezes chamado seu *modo de permissão.* O valor de *pmode* é altamente dependente do sistema operacional; verifique seu manual do usuário para detalhes. Em geral, os modos de acesso que um arquivo pode ter incluem apenas leitura, leitura/escrita e um tipo de acesso de segurança. Para muitos compiladores, os valores de *pmode* são definidos como macros no arquivo de cabeçalho chamado STAT.H (algumas vezes encontrado no diretório SYS). Os nomes mais comuns e seus significados são mostrados aqui:

S_IWRITE	Permite saída
S_IREAD	Permite entrada
S_IREAD ¦ S_IWRITE	Permite entrada/saída

Se o arquivo especificado já existe no momento da chamada a **creat()**, ele é apagado e todo conteúdo anterior é perdido.

Exemplo

O fragmento de código seguinte cria um arquivo chamado TEST.

```
#include <io.h>
#include <sys\stat.h>
#include <stdio.h>
#include <stdlib.h>

void main(void)
```

```
{
  int fd;

  if((fd=creat("test", S_IWRITE))==-1) {
    printf("O arquivo não pode ser aberto.\n");
    exit(1);
  }
}
```

Funções Relacionadas

open(), close(), read(), write(), unlink(), eof()

#include <conio.h>
int cscanf(const char *format,...);

A função **cscanf()** não é definida pelo padrão C ANSI. Ela é normalmente incluída na biblioteca de compiladores baseada no DOS.

A função **cscanf()** opera exatamente como **scanf()**, mas, em muitas implementações, sua entrada não pode ser redirecionada para nenhum outro dispositivo além do teclado. Além disso, em alguns ambientes **cscanf()** opera relativamente a uma janela em vez da tela. Verifique seu manual do usuário para detalhes.

Exemplo

Este programa lê uma string digitada no teclado:

```
#include <conio.h>

void main(void)
{
  char str[80];
  cprintf("digite uma string: ");
  cscanf("%s", str);
  cprintf(str);
}
```

Funções Relacionadas

cprintf(), cgets()

#include <io.h>
int eof(int fd);

A função **eof()** é parte do sistema de arquivo tipo UNIX e não é definida pelo padrão C ANSI. Quando chamada com um descritor válido de arquivo, **eof()** devolve 1 se o final do arquivo já foi alcançado; caso contrário, devolve zero. Se ocorre um erro, **eof()** devolve -1.

Exemplo

O programa seguinte mostra um arquivo-texto no console, usando **eof()** para determinar quando o final do arquivo foi alcançado:

```
#include <fcntl.h>
#include <io.h>
#include <stdio.h>
#include <stdlib.h>

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

  if((fd=open(argv[1], O_RDONLY))==-1) {
    printf("O arquivo não pode ser aberto.\n");
    exit(1);
  }

  while(!eof(fd)) {
    read(fd, &ch, 1); /* lê um caractere por vez */
    printf("%c", ch);
  }
  close(fd);
}
```

Funções Relacionadas

open(), close(), read(), write(), unlink()

#include <stdio.h>
int fclose(FILE *stream);

A função **fclose()** fecha o arquivo associado à *stream* e esvazia seu buffer. Após um **fclose()**, *stream* não está mais associada ao arquivo e quaisquer buffers automaticamente alocados são liberados.

Se **fclose()** for bem-sucedida, ela devolverá zero; caso contrário, devolverá um número diferente de zero. Tentar fechar um arquivo que já está fechado é um erro. Remover a mídia de armazenamento antes de fechar o arquivo é outro gerador de erro, como também a falta de espaço livre suficiente no disco.

Exemplo

O código seguinte abre e fecha um arquivo:

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

void main(void)
{
  FILE *fp;

  if((fp=fopen("test", "rb"))==NULL) {
    printf("O arquivo não pode ser aberto.\n");
    exit(1);
  }

  if(fclose(fp)) printf("Erro no fechamento do arquivo.\n");
}
```

Funções Relacionadas

fopen(), freopen(), fflush()

#include <stdio.h>
int feof(FILE *stream);

A função **feof()** verifica o indicador de posição de arquivo para determinar se foi atingido o final do arquivo associado à *stream*. Um valor diferente de zero é devolvido se o indicador de posição de arquivo está no final do arquivo; caso contrário, a função devolve zero.

Uma vez alcançado o final do arquivo, operações de leitura subseqüentes retornam **EOF** até que **rewind()** seja chamada ou que o indicador de posição de arquivo seja movido com **fseek()**. **EOF** está definido em STDIO.H.

A função **feof()** é particularmente útil quando se está trabalhando com arquivos binários, porque o marcador de final de arquivo também é um inteiro binário válido. Devem ser feitas chamadas explícitas a **feof()** para determinar quando o final de um arquivo binário foi atingido.

Exemplo

Este fragmento de código mostra a maneira apropriada de se ler até o final de um arquivo binário:

```
/*
    Assume que fp foi aberto como um arquivo binário para
    operações de leitura.
*/
while(!feof(fp)) getc(fp);
```

Funções Relacionadas

clearerr(), ferror(), perror(), putc(), getc()

#include <stdio.h>
int ferror(FILE *stream);

A função **ferror()** verifica a ocorrência de erros em uma dada *stream*. Um valor de retorno zero indica que nenhum erro ocorreu, enquanto um valor diferente de zero significa um erro.

O indicador de erro associado à *stream* permanece ativado até que o arquivo seja fechado ou **rewind()** ou **clearerr()** sejam chamadas.

Para determinar a natureza exata do erro, use a função **perror()**.

Exemplo

O fragmento de código seguinte interrompe a execução do programa se ocorre um erro de arquivo.

```
/*
  Assume que fp aponta para uma stream aberta para operações de
  escrita.
*/

while(!done) {
  putc(info, fp);
  if(ferror(fp)) {
    printf("Erro no arquivo\n");
    exit(1);
  }
}
```

Funções Relacionadas

clearerr(), feof(), perror()

#include <stdio.h>
int fflush(FILE *stream);

Se *stream* é associada a um arquivo aberto para escrita, uma chamada a **fflush()** escreve fisicamente no arquivo o conteúdo do buffer de saída. Se *stream* aponta para um arquivo de entrada, o conteúdo do buffer de entrada é esvaziado. Nos dois casos, o arquivo continua aberto.

Um valor de retorno zero indica sucesso; **fflush()** devolve **EOF** se ocorrer um erro de escrita.

Todos os buffers são automaticamente esvaziados com o término normal do programa ou quando estão cheios. Fechar um arquivo também esvazia seu buffer.

Exemplo

O fragmento de código seguinte esvazia o buffer após cada operação de escrita:

```
/*
  Assume que fp está associado a um arquivo de saída.
*/
.
.
.
fwrite(buf, sizeof(data_type), 1, fp);
fflush(fp);
.
.
.
```

Funções Relacionadas

fclose(), fopen(), fread(), fwrite(), getc(), putc()

#include <stdio.h>
int fgetc(FILE *stream);

A função **fgetc()** devolve o próximo caractere da *stream* de entrada na posição atual e incrementa o indicador de posição de arquivo. O caractere é lido como um **unsigned char**, que é convertido para um inteiro.

Se o final do arquivo for alcançado, **fgetc()** devolverá EOF. Porém, como EOF é um valor inteiro válido, você deve usar **feof()** para verificar o final do arquivo quando trabalhar com arquivos binários. Se **fgetc()** encontra um erro, também devolve EOF. Novamente, quando se trabalha com arquivos binários, deve-se usar **ferror()** para verificar erros com arquivos.

Exemplo

O programa seguinte lê e mostra o conteúdo de um arquivo-texto.

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

void main(int argc, char *argv[])
{
  FILE *fp;
  char ch;

  if((fp=fopen(argv[1],"r"))==NULL) {
    printf("O arquivo não pode ser aberto.\n");
    exit(1);
  }

  while((ch=fgetc(fp))!=EOF) {
    printf("%c", ch);
  }
  fclose(fp);
}
```

Funções Relacionadas

fputc(), getc(), putc(), fopen()

#include <stdio.h>
int fgetpos(FILE *stream, fpos_t *position);

A função **fgetpos()** armazena o valor atual do indicador de posição de arquivo no objeto apontado por *position*. O objeto apontado por *position* deve ser do tipo **fpos_t**, um tipo definido em STDIO.H. O valor armazenado é útil apenas em uma chamada subseqüente a **fsetpos()**.

Se ocorre um erro, **fgetpos()** retorna um valor diferente de zero; caso contrário, devolve zero.

Exemplo

O fragmento seguinte armazena a posição atual do arquivo em file_loc:

```
FILE *fp;
fpos_t file_loc;
.
.
.
fgetpos(fp, &file_loc);
```

Funções Relacionadas

fsetpos(), fseek(), ftell()

#include <stdio.h>
char *fgets(char *str, int num, FILE *stream);

A função **fgets()** lê *num-1* caracteres de *stream* e coloca-os na matriz de caracteres apontada por *str*. Os caracteres são lidos até que uma nova linha, ou um **EOF**, seja recebida ou até que o limite especificado seja atingido. Após os caracteres serem lidos, um nulo é colocado na matriz imediatamente após o último caractere. O caractere de nova linha é mantido e é parte de *str*.

Caso seja bem-sucedida, **fgets()** devolve *str*; um ponteiro nulo é devolvido quando ocorre alguma falha. Se ocorre um erro de leitura, o conteúdo da matriz apontada por *str* é indeterminado. Como um ponteiro nulo é devolvido, quando ocorre um erro ou quando o final do arquivo é atingido, você deve usar **feof()** ou **ferror()** para determinar o que realmente ocorreu.

Exemplo

Este programa usa **fgets()** para mostrar o conteúdo do arquivo-texto especificado no primeiro argumento da linha de comando:

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

void main(int argc, char *argv[])
{
  FILE *fp;
  char str[128];

  if((fp=fopen(argv[1], "r"))==NULL) {
    printf("O arquivo não pode ser aberto.\n");
```



```
    exit(1);
  }

  while(!feof(fp)) {
    if(fgets(str, 126, fp)) printf("%s", str);
  }

  fclose(fp);
}
```

Funções Relacionadas

fputs(), fgetc(), gets(), puts()

#include <stdio.h>
FILE *fopen(const char *fname, const char *mode);

A função **fopen()** abre um arquivo cujo nome é apontado por *fname* e devolve a stream associada a ele. Os tipos de operações permitidas nos arquivos são definidos pelo valor de *mode*. Os valores legais para *mode*, como especificado pelo padrão C ANSI, são mostrados na Tabela 12.1. O nome do arquivo deve ser uma string de caracteres que constitua um nome válido de arquivo, como definido pelo sistema operacional, e pode incluir uma especificação de percurso (*path*), se o ambiente a suporta.

Se **fopen()** abre o arquivo especificado, um ponteiro **FILE** é devolvido. Se o arquivo não pode ser aberto, é devolvido um ponteiro nulo.

Como mostra a Tabela 12.1, um arquivo pode ser aberto nos modos texto ou binário. No modo texto, podem ocorrer algumas traduções de caracteres. Por exemplo, novas linhas podem ser convertidas em seqüências de retorno de carro/alimentação de linha. Nenhuma tradução desse tipo ocorre em arquivos binários.

Este fragmento de código ilustra a maneira correta de abrir um arquivo:

```
FILE *fp;

if((fp = fopen("test", "w"))==NULL) {
  puts("O arquivo não pode ser aberto.\n");
  exit(1);
}
```

Tabela 12.1 Os valores legais para *mode*.

Modo	Significado
"r"	Abre arquivo-texto para leitura
"w"	Cria arquivo-texto para escrita
"a"	Anexa a um arquivo-texto
"rb"	Abre arquivo binário para leitura
"wb"	Cria arquivo binário para escrita
"ab"	Anexa a um arquivo binário
"r+"	Abre arquivo-texto para leitura/escrita
"w+"	Cria arquivo-texto para leitura/escrita
"a+"	Anexa arquivo-texto para leitura/escrita
"rb+"	Abre arquivo binário para leitura/escrita
"wb+"	Cria arquivo binário para leitura/escrita
"ab+"	Abre arquivo binário para leitura/escrita

Este método detecta qualquer erro na abertura do arquivo (como uma tentativa de abrir um arquivo em um disco protegido para escrita ou cheio) antes de tentar escrever nele. Um **NULL** é devolvido quando ocorre um erro, porque nenhum ponteiro de arquivo possui esse valor. **NULL** é definido em STDIO.H.

Se você usa **fopen()** para abrir um arquivo para escrita, qualquer arquivo preexistente com esse nome é apagado e um novo arquivo é iniciado. Se nenhum arquivo com esse nome existe, um é criado. Se você desejar adicionar ao final do arquivo, deve usar o modo "a". Abrir um arquivo para operações de leitura requer a existência do arquivo. Se o arquivo não existe, um erro é devolvido. Finalmente, se um arquivo é aberto para operações de leitura/escrita, ele não é apagado se já existe; porém, se ele não existe, é criado.

Quando acessar um arquivo aberto para operações de leitura/escrita, você não pode seguir uma operação de saída com uma operação de entrada sem uma chamada de intervenção a **fflush()**, **fseek()**, **fsetpos()** ou **rewind()**. Além disso, você não pode seguir uma operação de entrada com uma operação de saída sem uma chamada de intervenção a uma das funções mencionadas anteriormente.

Exemplo

Este fragmento abre um arquivo binário chamado TEST para operações de leitura/escrita:

```
FILE *fp;

if((fp=fopen("test", "rb+"))==NULL) {
  printf("O arquivo não pode ser aberto.\n");
```



```
    exit(1);
  }
```

Funções Relacionadas

fclose(), fread(), fwrite(), putc(), getc()

#include <stdio.h>
int fprintf(FILE *stream, const char *format,...);

A função **fprintf()** escreve na stream apontada por *stream* os valores dos argumentos da lista de argumentos, como especificado na string de formato. O valor de retorno é o número de caracteres realmente escritos. Se ocorre um erro, um número negativo é devolvido.

Pode haver de zero a vários argumentos — o número máximo depende do sistema.

As operações da string de controle de formato e os comandos são idênticos àqueles em **printf()**; veja a função **printf()** para a descrição completa.

Exemplo

Este programa cria um arquivo chamado TEST e escreve **isto é um teste 10 20.01** no arquivo, usando **fprintf()** para formatar os dados.

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

void main(void)
{
  FILE *fp;
  if((fp = fopen("test", "wb"))==NULL) {
    printf("O arquivo não pode ser aberto.\n");
    exit(1);
  }

  fprintf(fp, "isto é um teste %d %f", 10, 20.01);
  fclose(fp);
}
```

Funções Relacionadas

printf(), fscanf()

#include <stdio.h>
int fputc(int ch, FILE *stream);

A função **fputc()** escreve o caractere *ch* na stream especificada na posição atual do arquivo e avança o indicador de posição de arquivo. Embora *ch* seja declarado como um **int** por razões históricas, ele é convertido por **fputc()** em um **unsigned char**. Como todos os argumentos de caractere são elevados a inteiros no momento da chamada, variáveis tipo caractere geralmente são usadas como argumentos. Se um inteiro fosse usado, o byte mais significativo seria simplesmente ignorado.

O valor devolvido por **fputc()** é o do caractere escrito. Se ocorre um erro, **EOF** é devolvido. Para arquivos abertos para operações binárias, um **EOF** pode ser um caractere válido e você deve usar a função **ferror()** para determinar se realmente ocorreu um erro.

Exemplo

Esta função escreve o conteúdo da string na stream especificada:

```
void write_string(char *str, FILE *fp)
{
  while(*str) if(!ferror(fp)) fputc(*str++, fp);
}
```

Funções Relacionadas

fgetc(), fopen(), fprintf(), fread(), fwrite()

#include <stdio.h>
int fputchar(int ch);

A função **fputchar()** escreve o caractere *ch* em **stdout**. Essa função não é definida pelo padrão C ANSI, mas é um acréscimo muito comum. Muito embora *ch* seja declarado como um **int**, por razões históricas, ele é convertido por **fputchar()** em um **unsigned char**. Como todos os argumentos de caractere são elevados a inteiros no momento da chamada, variáveis tipo caractere geralmente são usadas como argumentos. Se um inteiro fosse usado, o byte mais significativo seria simplesmente descartado. Uma chamada a **fputchar()** equivale funcionalmente a uma chamada a **fputc(ch, stdout)**.

O valor devolvido por **fputchar()** é o do caractere escrito. Se ocorre um erro, EOF é devolvido. Para arquivos abertos a operações binárias, um EOF pode ser um caractere válido e você deve usar a função **ferror()** para determinar se realmente ocorreu um erro.

Exemplo

Essa função escreve o conteúdo da string em **stdout**:

```
void write_string(char *str)
{
  while(*str) if(!ferror(fp)) fputchar(*str++);
}
```

Funções Relacionadas

fgetc(), fopen(), fprintf(), fread(), fwrite()

#include <stdio.h>
int fputs(const char *str, FILE *stream);

A função **fputs()** escreve na stream especificada o conteúdo da string apontada por *str*. O terminador nulo não é escrito.

A função **fputs()** devolve um valor não negativo em caso de sucesso e EOF em caso de falha.

Se a stream é aberta no modo texto, certas traduções de caracteres podem ocorrer. Isso significa que pode não haver uma correspondência de um para um da string com o arquivo. No entanto, se a stream for aberta no modo binário, nenhuma tradução de caracteres ocorrerá e haverá uma correspondência de um para um entre a string e o arquivo.

Exemplo

Esse fragmento de código escreve a string **isso é um teste** na stream apontada por **fp**.

```
fputs("isso é um teste", fp);
```

Funções Relacionadas

fgets(), gets(), puts(), fprintf(), fscanf()

#include <stdio.h>
size_t fread(void *buf, size_t size, size_t count, FILE *stream);

A função **fread()** lê *count* números de objetos, cada um com *size* bytes de comprimento da stream apontada por *stream* e coloca-os na matriz apontada por *buf*. O indicador de posição de arquivo é avançado pelo número de caracteres lido.

A função **fread()** devolve o número de itens realmente lidos. Caso sejam lidos menos itens do que o solicitado na chamada, isso significa que ocorreu um erro ou que o final do arquivo foi atingido. Você deve utilizar **feof()** ou **ferror()** para determinar o que aconteceu.

Se a *stream* foi aberta para operações de texto, certas traduções podem ocorrer (como as seqüências de retorno de carro e alimentação de linha sendo transformadas em novas linhas).

Exemplo

O programa seguinte lê dez números em ponto flutuante de um arquivo chamado TEST para a matriz **bal**.

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

void main(void)
{
  FILE *fp;
  float bal[10];

  if((fp=fopen("test", "rb"))==NULL) {
    printf("O arquivo nao pode ser aberto.\n");
    exit(1);
  }

  if(fread(bal, sizeof(float), 10, fp)!=10) {
    if(feof(fp) printf("Fim de arquivo prematuro.");
    else printf("Erro na leitura do arquivo.");
  }

  fclose(fp);
}
```


Funções Relacionadas

fwrite(), fopen(), fscanf(), fgetc(), getc()

#include <stdio.h>
FILE *freopen(const char *fname, const char *mode, FILE *stream);

A função **freopen()** associa uma stream existente com um arquivo diferente. O nome do novo arquivo é apontado por *fname*, o modo de acesso, por *mode* e a stream a ser redefinida é apontada por *stream*. A string *mode* usa o mesmo formato de **fopen()**. Para uma discussão completa, veja a descrição de **fopen()**.

Quando chamada, **freopen()** primeiro tenta fechar um arquivo que pode estar atualmente associado a stream. Porém, se a tentativa de fechar o arquivo falha, a função ainda continua para abrir o outro arquivo.

A função **freopen()** devolve um ponteiro para *stream* em caso de sucesso; caso contrário, a função devolve um ponteiro nulo.

O principal uso de **freopen()** é redirecionar os arquivos definidos por **stdin**, **stdout** e **stderr** para algum outro arquivo.

Exemplo

O programa mostrado aqui usa **freopen()** para redirecionar a *stream* **stdout** para o arquivo chamado OUT. Como **printf()** escreve para **stdout**, a primeira mensagem é mostrada na tela e a segunda é escrita no arquivo em disco.

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

void main(void)
{
  FILE *fp;

  printf("Isso será mostrado na tela.\n");

  if((fp=freopen("OUT", "w" ,stdout))==NULL) {
    printf("O arquivo não pode ser aberto.\n");
    exit(1);
  }

  printf("Isso será escrito no arquivo OUT.");

  fclose(fp);
}
```

Funções Relacionadas

fopen(), fclose()

#include <stdio.h>
int fscanf(FILE *stream, const char *format,...);

A função **fscanf()** opera exatamente como **scanf()**, mas lê a informação da stream especificada por *stream* em lugar de **stdin**. Veja a função **scanf()** para mais detalhes.

A função **fscanf()** devolve o número de argumentos que realmente receberam valores. Esse número não inclui os campos ignorados. Um valor de retorno de **EOF** significa que ocorreu uma falha antes que a primeira atribuição tenha sido feita.

Exemplo

Esse fragmento de código lê uma string e um **float** da stream **fp**.

```
char str[80];
float f;

fscanf(fp, "%s%f", str, &f);
```

Funções Relacionadas

scanf(), fprintf()

#include <stdio.h>
int fseek(FILE *stream, long offset, int origin);

A função **fseek()** coloca o indicador de posição de arquivo associado a *stream* de acordo com os valores de *offset* e *origin*. Ela suporta operações de E/S aleatórias. *Offset* é o número de bytes a partir de *origin* até chegar à nova posição. O valor para *origin* deve ser uma destas macros (definidas em STDIO.H):

Nome	Significado
SEEK_SET	Move a partir do início do arquivo
SEEK_CUR	Move a partir da posição atual
SEEK_END	Move a partir do final do arquivo

Um valor zero devolvido significa que **fseek()** obteve sucesso. Um valor diferente de zero indica falha.

Na maioria das implementações, e como especificado pelo padrão C ANSI, *offset* deve ser do tipo **long** para suportar arquivos maiores que 64K.

Você pode usar **fseek()** para mover o indicador de posição a qualquer lugar no arquivo, mesmo além do final. Porém, é um erro colocar o indicador antes do início do arquivo.

A função **fseek()** limpa o indicador de fim de arquivo associado à stream especificada. Além disso, torna nulo qualquer **ungetc()** anterior na mesma stream. (Veja **ungetc()**.)

Exemplo

A função seguinte dirige-se à estrutura do tipo **addr** especificada. Observe o uso de **sizeof** para obter o tamanho da estrutura.

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

void find(long client_num)
{
  FILE *fp;

  if((fp=fopen("mail", "rb"))==NULL) {
    printf("O arquivo não pode ser aberto.\n");
    exit(1);
  }

  /* encontra a estrutura apropriada */
  fseek(fp, client_num*sizeof(struct addr), SEEK_SET);

  /* lê os dados para a memória */
  fread(&info, sizeof(struct addr), 1, fp);

  fclose(fp);
}
```

Funções Relacionadas

ftell(), rewind(), fopen(), fgetpos(), fsetpos()

#include <stdio.h>
int fsetpos(FILE *stream, const fpos_t *position);

A função **fsetpos()** move o indicador de posição de arquivo para o ponto especificado pelo objeto apontado por *position*. Esse valor deve ser previamente obtido por meio de uma chamada a **fgetpos()**. O tipo **fpos_t** é definido em STDIO.H. Depois que **fsetpos()** é executada, o indicador de fim de arquivo é desligado. Além disso, qualquer chamada anterior a **ungetc()** se torna nula.

Se **fsetpos()** falha, ela devolve um valor diferente de zero. Se obtém sucesso, devolve zero.

Exemplo

Este fragmento de código recoloca o indicador de posição de arquivo no valor armazenado em **file_loc**.

```
fsetpos(fp, &file_loc);
```

Funções Relacionadas

fgetpos(), fseek(), ftell()

#include <stdio.h>
long ftell(FILE *stream);

A função **ftell()** devolve o valor atual do indicador de posição de arquivo para a stream especificada. Para streams binárias, o valor é o número de bytes cujo indicador está a partir do início do arquivo. Para streams de texto, o valor de retorno pode não ser significativo, exceto como um argumento de **fseek()**, devido às possíveis traduções de caracteres. Por exemplo, retornos de carro/alimentações de linha podem ser substituídos por novas linhas, o que altera o tamanho aparente do arquivo.

A função **ftell()** devolve -1L quando ocorre um erro. Se a stream não permite acesso aleatório — se for um terminal, por exemplo —, o valor devolvido é indefinido.

Exemplo

Este fragmento de código devolve o valor atual do indicador de posição de arquivo para a stream apontada por **fp**:

```
long i;
```



```
if((i=ftell(fp))==-1L)
  printf("Erro no arquivo.\n");
```

Funções Relacionadas

fseek(), fgetpos()

#include <stdio.h>
size_t fwrite(const void *buf, size_t size_, size_t count, FILE *stream);

A função **fwrite()** escreve *count* objetos (cada um com *size* bytes de comprimento) na stream apontada por *stream* da matriz de caracteres apontada por *buf*. O indicador de posição de arquivo é avançado pelo número de caracteres escritos.

A função **fwrite()** devolve o número de itens realmente escritos, que será igual ao número solicitado, caso a função seja bem-sucedida. Se forem escritos menos itens que os solicitados, ocorreu algum erro. Para streams de texto, diversas traduções de caracteres podem ocorrer, mas não afetarão o valor devolvido.

Exemplo

Este programa escreve um **float** no arquivo TEST. Observe que é usado **sizeof** para determinar o número de bytes em uma variável **float** e, assim, assegurar portabilidade.

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

void main(void)
{
  FILE *fp;
  float f=12.23;

  if((fp=fopen("test", "wb"))==NULL) {
    printf("O arquivo não pode ser aberto.\n");
    exit(1);
  }

  fwrite(&f, sizeof(float), 1, fp);

  fclose(fp);
}
```

Funções Relacionadas

fread(), fscanf(), getc(), fgetc()

#include <stdio.h>
int getc(FILE *stream);

A função **getc()** devolve o próximo caractere da *stream* de entrada a partir da posição atual e incrementa o indicador de posição de arquivo. O caractere é lido como um **unsigned char** e é convertido em um inteiro.

Se o final do arquivo for alcançado, **getc()** devolverá EOF. Porém, como EOF é um valor inteiro válido, você deve usar **feof()** para verificar o final do arquivo quando trabalhar com arquivos binários. Se **getc()** encontra um erro, também devolve EOF. Quando se trabalha com arquivos binários, deve-se usar **ferror()** para verificar erros em arquivos.

As funções **getc()** e **fgetc()** são idênticas; em muitas implementações, **getc()** é definida simplesmente como a macro mostrada aqui:

```
#define getc(fp) fgetc(fp)
```

Isso faz com que a função **fgetc()** substitua a macro **getc()**.

Exemplo

O programa seguinte lê e mostra o conteúdo de um arquivo de texto.

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

void main(int argc, char *argv[])
{
  FILE *fp;
  char ch;

  if((fp=fopen(argv[1],"r"))==NULL) {
    printf("O arquivo não pode ser aberto.\n");
    exit(1);
  }

  while((ch=getc(fp))!=EOF) {
    printf("%c", ch);
  }

  fclose(fp);
}
```


Funções Relacionadas

fputc(), fgetc(), putc(), fopen()

#include <conio.h>
int getch(void); int getche(void);

As funções **getch()** e **getche()** não são definidas pelo padrão C ANSI. Porém, elas geralmente são incluídas em compiladores baseados em DOS.

A função **getch()** devolve o próximo caractere lido do console, mas não o mostra na tela.

A função **getche()** devolve o próximo caractere lido do console e mostra-o na tela.

Essas duas funções contornam as funções de E/S padrão de C e operam diretamente com o sistema operacional. Essencialmente, **getch()** e **getche()** efetuam a entrada diretamente do teclado.

Exemplo

Este fragmento de código usa **getch()** para ler a escolha do usuário no menu em um programa verificador de ortografia:

```
do {
  printf("1: Verificar ortografia\n");
  printf("2: Corrigir ortografia\n");
  printf("3: Procurar uma palavra no dicionário\n");
  printf("4: Sair\n");

  printf("\nEntre com sua escolha: ");
  choice = getch();
} while(!strchr("1234",choice));
```

Funções Relacionadas

getc(), getchar(), fgetc()

#include <stdio.h>
int getchar(void);

A função **getchar()** devolve o próximo caractere de **stdin**. O caractere é lido como um **unsigned char** e é convertido para um inteiro.

Se o final do arquivo for alcançado, **getchar()** devolverá **EOF**. Porém, como **EOF** é um valor inteiro válido, deve usar **feof()** para verificar o final do arquivo quando trabalhar com arquivos binários. Se **getchar()** encontra um erro, também devolve **EOF**. Se você está trabalhando com arquivos binários, você deve usar **ferror()** para verificar erros em arquivos.

A função (ou macro) **getchar()** equivale funcionalmente a **getc(stdin)**.

Exemplo

Este programa lê caracteres de **stdin** para a matriz **s** até que o usuário pressione ENTER. Finalmente, a string é apresentada.

```
#include <stdio.h>

void main(void)
{
  char s[256], *p;

  p = s;

  while((*p++=getchar())!='\n') ;
  *p = '\0'; /* acrescenta o terminador nulo */
  printf(s);
}
```

Funções Relacionadas

fputc(), fgetc(), putc(), fopen()

#include <stdio.h>
char *gets(char *str);

A função **gets()** lê caracteres de **stdin** e coloca-os na matriz apontada por *str*. Os caracteres são lidos até que seja recebido um caractere de nova linha ou um EOF. O caractere de nova linha não faz parte da string. Em vez disso, ele é traduzido em um nulo, terminando a string.

Caso seja bem-sucedida, **gets()** devolve *str*; um ponteiro nulo é devolvido em caso de falha. Se ocorre um erro de leitura, o conteúdo da matriz apontada por *str* é indeterminado. Como um ponteiro nulo é devolvido tanto quando ocorre um erro como quando o final do arquivo é atingido, use **feof()** ou **ferror()** para determinar o que realmente aconteceu.

Não há limite para o número de caracteres que **gets()** irá ler. Por essa razão, é seu dever assegurar que a matriz apontada por *str* não ultrapasse seus limites.

Exemplo

Este programa usa **gets()** para ler um nome de arquivo:

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

void main(void)
{
  FILE *fp;
  char fname[128];

  printf("Digite o nome do arquivo: ");
  gets(fname);

  if((fp=fopen(fname, "r"))==NULL) {
    printf("O arquivo não pode ser aberto. \n");
    exit(1);
  }

  fclose(fp);
}
```

Funções Relacionadas

fputs(), fgetc(), fgets(), puts()

#include <stdio.h>
int getw(FILE *stream);

A função **getw()** não é definida pelo padrão C ANSI, mas é incluída em muitos compiladores C.

A função **getw()** devolve o próximo inteiro de *stream* e avança o indicador de posição do arquivo adequadamente.

Como o inteiro lido pode ter um valor igual a **EOF**, você deve usar **feof()** ou **ferror()** para determinar quando o final do arquivo foi alcançado ou se ocorreu um erro.

Exemplo

Este programa lê inteiros do arquivo INTTESTE e apresenta a soma deles:

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

void main(void)
{
  FILE *fp;
  int sum=0;

  if((fp=fopen("intteste", "rb")==NULL) {
    printf("O arquivo não pode ser aberto.\n");
    exit(1);
  }

  while(!feof(fp))
    sum = getw(fp)+sum;

  printf("a soma é %d", sum);

  fclose(fp);
}
```

Funções Relacionadas

putw(), fread()

#include <conio.h>
int kbhit(void);

A função **kbhit()** não é definida pelo padrão C ANSI. Porém, ela é encontrada, sob diversos nomes, em virtualmente toda implementação de C. Ela devolverá um valor diferente de zero se uma tecla foi pressionada no console. Caso contrário, devolverá zero. (Sob nenhuma circunstância ela espera que uma tecla seja pressionada.)

Exemplo

Esse laço **for** terminará se uma tecla for pressionada:


```
for(;;) {
  if(kbhit()) break;
  .
  .
  .
}
```

Funções Relacionadas

fgetc(), getc()

#include <io.h>
long lseek(int fd, long offset, int origin);

A função **lseek()** é parte do sistema E/S tipo UNIX e não é definida pelo padrão C ANSI.

A função **lseek()** coloca o indicador de posição de arquivo no ponto especificado por *offset* e *origin*. O *offset* é o número de bytes a partir de *origin* até chegar a nova posição. O valor para *origin* deve ser uma destas macros (definidas em IO.H):

Nome	Significado
SEEK_SET	Move a partir do início do arquivo
SEEK_CUR	Move a partir da posição atual
SEEK_END	Move a partir do final do arquivo

A função **lseek()** devolve *offset* em caso de sucesso. Em caso de falha, ela devolve -1.

Exemplo

O exemplo mostrado aqui permite-lhe examinar um arquivo, um setor por vez, usando o sistema de E/S tipo UNIX. Talvez seja necessário mudar o tamanho do buffer para coincidir com o tamanho do setor do seu sistema.

```
#include <io.h>
#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>

#define BUF_SIZE  128

void main(int argc, char *argv[])
{
  char buf[BUF_SIZE], s[10];
```

```
  int fd, sector;

  buf[BUF_SIZE] = '\0'; /* termina o buffer com nulo
                           para printf */

  if((fd=open(argv[1], O_RDONLY))==-1) { /*abre para leitura*/
    printf("O arquivo não pode ser aberto.\n");
    exit(1);
  }

  do {
    printf("buffer: ");
    gets(s);

    sector = atoi(s); /* obtém o setor a ler */

    if(lseek(fd, (long)sector*BUF_SIZE, SEEK_SET)==-1L)
      printf("Erro na busca.\n");

    if(read(fd, buf, BUF_SIZE)==0) {
      printf("Setor fora da faixa.\n");
    }
    else
      printf("%s\n", buf);
  } while(sector>0);
  close(fd);
}
```

Funções Relacionadas

read(), write(), open(), close()

#include <fnctl.h>
#include <io.h>
int open(const char *fname, int mode);

A função **open()** é parte do sistema de E/S tipo UNIX e não é definida pelo padrão C ANSI.

Ao contrário do sistema de E/S bufferizado, o sistema tipo UNIX não usa ponteiros de arquivo do tipo **FILE**; em vez disto ele usa descritores de arquivo do tipo **int**. A função **open()** abre um arquivo cujo nome é apontado por *fname* e estabelece seu modo de acesso como especificado por *mode*. Os valores que *mode* pode assumir são mostrados aqui (estas macros estão definidas em FCNTL.H):

Modo	Efeito
O_RDONLY	Abre para leitura
O_WRONLY	Abre para escrita
O_RDWR	Abre para leitura e escrita

NOTA: Muitos compiladores têm modos adicionais — como texto, binário etc. —, portanto verifique seu manual do usuário.

Uma chamada a **open()** bem-sucedida devolve um inteiro positivo, que é o descritor de arquivo associado ao arquivo. Um valor devolvido igual a -1 significa que o arquivo não pode ser aberto.

Em muitas implementações, a operação falha se o arquivo especificado no comando **open()** não está no disco. No entanto, dependendo da implementação, pode ser possível usar **open()** para criar um arquivo que atualmente não existe. Verifique seu manual do usuário.

Exemplo

Você geralmente verá a chamada a **open()** desta forma:

```
int fd;

if((fd=open(filename, mode)) == -1) {
  printf("O arquivo não pode ser aberto.\n");
  exit(1);
}
```

Funções Relacionadas

close(), read(), write()

#include <stdio.h>
void perror(const char *str);

A função **perror()** gera uma mensagem de erro, por meio do valor da variável global **errno**, e escreve a string **str** em **stderr**. Se o valor de *str* não for nulo, a string é escrita primeiro, seguida por dois-pontos, e então vem a mensagem de erro definida pela implementação.

Exemplo

Neste fragmento, caso ocorra um erro, ele será apresentado.

```
#include <stdio.h>
.
.
.
if(ferror(fp)) perror("Erro no arquivo: ");
```

#include <stdio.h>
int printf(const char *format,...);

A função **printf()** escreve em **stdout** os argumentos que formam a lista de argumentos sob o controle da string apontada por *format*.

A string apontada por *format* contém dois tipos de itens. O primeiro consiste em caracteres que serão escritos na tela. O segundo tipo contém comandos de formato que definem a forma como os argumentos serão mostrados. Um comando de formato consiste em um símbolo de percentagem seguido pelo código do formato. Os comandos de formato são mostrados na Tabela 12.2. Deve haver exatamente o mesmo número de argumentos e de comandos de formato e eles serão associados na ordem. Por exemplo, essa chamada a **printf()**:

```
printf("Ola %c %d %s", 'c', 10, "aqui!");
```

mostra **Ola c 10 aqui!**.

Tabela 12.2 Comandos de formato de printf().

Código	Formato
%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 em decimal
%g	Usa %e ou %f, o que tiver menor comprimento
%G	Usa %E ou %F, o que tiver menor comprimento
%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	Mostra 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 %

Se não há argumentos suficientes para combinar com os comandos de formato, a saída é indefinida. Se há mais argumentos do que comandos de formato, os argumentos restantes são descartados.

A função **printf()** devolve o número de caracteres realmente escritos. Um valor de retorno negativo indica um erro.

Os comandos de formato podem ter modificadores que especificam a largura do campo, o número de casas decimais e uma indicação de ajuste à esquerda. Um inteiro colocado entre o sinal de percentagem e o comando de formato atua como um especificador de largura mínima de campo, preenchendo a saída com brancos ou zeros para garantir o comprimento mínimo. Se a string ou número é maior que o mínimo, ela será escrita por completo. O preenchimento padrão é feito com espaços. Se você quiser preencher com zeros, coloque um zero antes do especificador de largura de campo. Por exemplo, **%05d** preenche com zeros um número com menos de 5 dígitos de forma que seu comprimento total seja cinco dígitos.

Para especificar o número de casas decimais a ser escrito para um número em ponto flutuante, coloque um ponto decimal seguido do número de casas decimais desejado após o especificador de largura de campo. Para os formatos **e**, **E** e **f**, o modificador de precisão determina o número de casas decimais a imprimir. Por exemplo, **%10.4f** mostra um número de pelo menos dez caracteres com quatro casas decimais. No entanto, se usado com o especificador **g** ou **G**, a precisão determina o número máximo de dígitos significativos exibidos.

Quando um modificador de precisão é aplicado a inteiros, ele especifica o número mínimo de digitos a exibir (0 à esquerda serão adicionados se necessário).

Quando o modificador de precisão é aplicado a string, o número após o ponto especifica a largura máxima. Por exemplo, **%5.7s** exibe uma string que tem pelo menos 5 caracteres de comprimento e não passa de 7. Se a string for mais comprida que a largura máxima, então os caracteres finais serão truncados.

Por padrão, toda saída é ajustada à direita. Isto é, se a largura do campo é maior do que os dados escritos, os dados serão colocados na extremidade direita do campo. Você pode forçar para que a informação seja justificada à esquerda colocando um sinal de subtração imediatamente após o sinal de porcentagem. Por exemplo, **%-10.2f** justifica à esquerda um número em ponto flutuante com duas casas decimais em um campo de até dez caracteres.

Há dois especificadores de formato que permitem que **printf()** mostre inteiros **short** e **long**. Esses especificadores podem ser aplicados aos de tipo **d**, **i**, **o**, **u**, **x** e **X**. O especificador **l** diz a **printf()** que segue um tipo de dado **long**. Por exemplo, **%ld** significa que um **long int** será mostrado. O especificador **h** instrui **printf()** a mostrar um **short int**. Portanto, **%hu** indica que o dado é do tipo **short unsigned int**.

Embora tecnicamente não seja exigido, em muitas implementações o modificador **l** também pode vir como prefixo dos especificadores de ponto flutuante **e**, **E**, **f**, **g** e **G** indica que segue um **double**. **L** é usado para indicar um **long double**.

O formato **n** coloca o número de caracteres escritos até então em uma variável inteira cujo ponteiro é especificado na lista de argumentos. Por exemplo, este fragmento de código mostra o número **15** após a linha **isso e um teste**.

```
int i;

printf("isso e um teste %n",&i);
printf("%d", i);
```

O # tem um significado especial quando utilizado com alguns especificadores de formato de **printf()**. Preceder um especificador g, G, f, e ou E com um # assegura que o ponto decimal estará presente mesmo que não haja dígitos decimais. Se você precede o especificador de formato **x** com um #, os números hexadecimais serão escritos com o prefixo **0x**. Quando usa com o especificador **o**, ele provoca uma orientação O para ser impressa. O # não pode ser aplicado a nenhum outro especificador de formato.

Os especificadores de largura mínima de campo e de precisão podem ser fornecidos como argumentos para **printf()** em lugar de constantes. Para tanto, deve-se utilizar um * como um elemento de substituição. Quando a string de formato é varrida, **printf()** irá combinar os * com os argumentos na ordem em que aparecem.

Veja a discussão de **printf()** no Capítulo 8 para maiores detalhes.

Exemplo

Este programa apresenta a saída mostrada em seus comentários.

```
#include <stdio.h>

void main(void)
{
  /* Isso escreve "isso é um teste" justificado à esquerda
     em um campo de 20 caracteres.
  */
  printf(%-20s","isso é um teste");

  /* Isso escreve um float com 3 casas decimais em um campo
     de 10 caracteres. O resultado será "    12.235".
  */
  printf("%10.3f", 12.234657);
}
```


Funções Relacionadas

scanf(), fprintf()

#include <stdio.h>
int putc(int ch, FILE *stream);

A função **putc()** escreve o caractere contido no byte menos significativo de *ch* na stream de saída apontada por *stream*. Como os argumentos de caractere são elevados a inteiros no momento da chamada, você pode utilizar variáveis tipo caractere como argumentos para **putc()**.

A função **putc()** devolve o caractere escrito; ela devolve **EOF** se ocorre algum erro. Se a *stream* de saída foi aberta no modo binário, **EOF** é um valor válido para *ch*. Isso significa que você deve usar **ferror()** para determinar se ocorreu algum erro.

A função **putc()** é geralmente implementada como uma macro sendo substituída por **fputc()**, pois esta equivale funcionalmente a **putc()**.

Exemplo

O laço seguinte escreve os caracteres da string **str** na *stream* especificada por **fp**. O terminador nulo não é escrito.

```
for(; *str; str++) putc(*str, fp);
```

Funções Relacionadas

fgetc(), fputc(), getchar(), putchar()

#include <conio.h>
int putch(int ch);

A função **putch()** não é definida pelo padrão C ANSI. Porém, ela geralmente é incluída na biblioteca padrão de compiladores baseada em DOS.

A função **putch()** escreve na tela o caractere especificado no byte menos significativo de *ch*. Em muitas implementações, a sua saída não pode ser redirecionada. Além disso, em alguns ambientes, ela pode operar relativamente a uma janela em lugar da tela.

Exemplo

O seguinte escreve o caractere **A** na tela:

```
putch('A');
```

Funções Relacionadas

putc(), putchar()

#include <stdio.h>
int putchar(int ch);

A função **putchar()** escreve em **stdout** o caractere contido no byte menos significativo de *ch*. A função **putchar()** é funcionalmente equivalente a **putc(ch, stdout)**. Como os argumentos tipo caractere são promovidos a inteiros no momento da chamada, você pode usar variáveis tipo caractere como argumentos para **putchar()**.

Em caso de sucesso, a função **putchar()** devolve o caractere escrito. Ela devolve **EOF** se ocorre um erro. Se a stream de saída foi aberta no modo binário, **EOF** é um valor válido para *ch*. Isso significa que você deve usar **ferror()** para determinar se ocorreu algum erro.

Exemplo

O laço seguinte escreve em **stdout** os caracteres da string **str**. O terminador nulo não é escrito.

```
for(; *str; str++) putchar(*str);
```

Função Relacionada

putc()

#include <stdio.h>
int puts(const char *str);

A função **puts()** escreve a string apontada por *str* no dispositivo de saída padrão. O terminador nulo é traduzido para uma nova linha.

A função **puts()** devolve um valor não-negativo, se bem-sucedida, e **EOF** em caso de falha.

Exemplo

O código seguinte escreve a string **isso é um exemplo** em **stdout**.

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



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

  strcpy(str, "isso é um exemplo");

  puts(str);
}
```

Funções Relacionadas

putc(), gets(), printf()

#include <stdio.h>
int putw(int i, FILE *stream);

A função **putw()** não é definida pelo padrão C ANSI, mas geralmente é incluída na biblioteca padrão.

A função **putw()** escreve o inteiro *i* em stream na posição atual do arquivo e incrementa o indicador de posição de arquivo apropriadamente.

A função **putw()** devolve o valor escrito. Um valor de retorno **EOF** significa que ocorreu um erro na stream, caso se esteja no modo texto. Como **EOF** também é um valor inteiro válido, você deve usar **ferror()** para detectar erro em uma stream binária.

Exemplo

Este fragmento de código escreve o valor 100 na stream apontada por **fp**.

```
putw(100, fp);
```

Funções Relacionadas

getw, printf(), fwrite()

#include <io.h>
int read(int fd, void *buf, unsigned count);

A função **read()** é parte do sistema de E/S tipo UNIX e não é definida pelo padrão C ANSI.

A função **read()** lê *count* bytes do arquivo descrito por *fd* para o buffer apontado por *buf*. O indicador de posição de arquivo é incrementado no número de bytes lidos. Se o arquivo for aberto no modo texto, poderão ocorrer traduções de caractere.

O valor devolvido é igual ao número de bytes realmente lido. Esse número poderá ser menor que *count* se for encontrado o final do arquivo ou um erro. Um valor -1 significa um erro; um valor zero é devolvido se for efetuada uma tentativa de ler no final do arquivo.

A função **read()** tende a ser muito dependente da implementação e pode comportar-se de maneira diferente do exposto aqui. Por favor, verifique seu manual do usuário para mais detalhes.

Exemplo

Este programa lê os primeiros 100 bytes do arquivo TEST e coloca-os na matriz buffer.

```
#include <fcntl.h>
#include <io.h>
#include <stdlib.h>
#include <stdio.h>

void main(void)
{
  int fd;
  char buffer[100];
  if((fd=open("test", O_RDONLY))==-1) {
    printf("O arquivo não pode ser aberto.\ n");
    exit(1);
  }

  if(read(fd, buffer, 100)!=100) printf("erro de leitura");
}
```

Funções Relacionadas

open(), close(), write(), lseek()

#include <stdio.h>
int remove(const char *fname);

A função **remove()** apaga o arquivo especificado por *fname*. Ela devolve zero se a operação foi um sucesso e um valor diferente de zero se ocorreu um erro.

Exemplo

Este programa remove o arquivo cujo nome é especificado na linha de comando:

```
#include <stdio.h>

void main(int argc, char *argv[])
{
  if(remove(argv[1])) printf ("Erro de exclusão");
}
```

Função Relacionada

rename()

#include <stdio.h>
int rename(const char *oldfname, const char *newfname);

A função **rename()** altera o nome do arquivo especificado por *oldfname* para *newfname*. O *newfname* não pode coincidir com nenhum nome existente no diretório.

A função **rename()** devolve zero se bem-sucedida e diferente de zero caso ocorra um erro.

Exemplo

Este programa troca o nome do arquivo especificado no primeiro argumento da linha de comando pelo especificado no segundo. Assumindo que o programa se chame MUDE, uma linha de comando, consistindo em MUDE ISSO AQUILO, mudará o nome de arquivo chamado ISSO para AQUILO.

```
#include <stdio.h>

void main(int argc, char *argv[])
{
  if(rename(argv[1], argv[2])!=0)
    printf("erro na troca de nomes");
}
```

Função Relacionada

remove()

#include <stdio.h>
void rewind(FILE *stream);

A função **rewind()** move o indicador de posição de arquivo para o início da stream especificada. Ela também limpa os indicadores de erro e de final de arquivo associados a stream. Não há valor de retorno.

Exemplo

Esta função lê duas vezes a stream apontada por **fp**, mostrando o arquivo duas vezes.

```
void re_read(FILE *fp)
{
  /* lê uma vez */
  while(!feof(fp)) putchar(getc(fp));
  rewind(fp);

  /* lê duas vezes */
  while(!feof(fp)) putchar(getc(fp));
}
```

Função Relacionada

fseek()

#include <stdio.h>
int scanf(const char *format,...);

A função **scanf()** é uma rotina de entrada de uso geral que lê a stream **stdin**. Ela pode ler todos os tipos de dados intrínsecos e convertê-los automaticamente no formato interno apropriado. É o complemento de **printf()**.

A string de controle apontada por *format* consiste em três tipos de caracteres:

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

Os especificadores de formato de entrada são precedidos por um sinal de percentagem e informam a **scanf()** que tipo de dado deve ser lido em seguida. Esses códigos estão listados na Tabela 12.3. Por exemplo, **%s** lê uma string enquanto **%d** lê um inteiro.

Tabela 12.3 Códigos de formato para **scanf()**.

Código	Significado
%c	Lê um único caractere
%d	Lê um inteiro decimal
%i	Lê um inteiro decimal
%e	Lê um número tipo ponto flutuante
%f	Lê um número tipo ponto flutuante
%g	Lê um número tipo ponto flutuante
%o	Lê um número octal
%s	Lê uma string
%x	Lê um número hexadecimal
%p	Lê um apontador
%n	Retorna um número inteiro igual ao número de caracteres lidos
%u	Lê um inteiro não sinalizado
%[]	Procura um conjunto de caracteres.
%%	Lê o símbolo %

A string de formato é lida da esquerda para a direita e os códigos de forma- to coincidem, em ordem, com os argumentos que formam a lista de argumentos.

Um caractere de espaço em branco na string de controle faz com que **scanf()** leia e descarte um ou mais caracteres 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 o primeiro caractere diferente seja alcançado.

Um caractere de espaço não-branco faz com que **scanf()** leia e descarte um caractere igual. 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.

Todas as variáveis usadas para receber valores por meio de **scanf()** devem ser passadas através de seus endereços. Isso significa que todos os argumentos devem ser ponteiros para as variáveis usadas como argumentos. Essa é a maneira de C criar uma chamada por referência, permitindo que uma função altere o conteúdo de um argumento. Por exemplo, para ler um inteiro e armazená-lo na variável **count**, você usaria a seguinte chamada a **scanf()**:

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

Strings são lidas e armazenadas em matrizes de caracteres e o nome da matriz, sem índice, é o endereço do primeiro elemento da matriz. Portanto, para ler uma string e carregá-la em **address**, use

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

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

Os itens de entrada de dados devem ser separados por espaços, tabulações ou novas linhas. Sinais de pontuação como vírgula, ponto-e-vírgula etc. não contam como separadores. Isso significa que

```
scanf("%d%d", &r, &c);
```

aceita a entrada **10 20** mas falha com **10,20.**

Um * colocado após o % e antes do especificador de formato lê o dado do tipo especificado, mas suprime sua atribuição. Assim,

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

dada a entrada **10/20,** coloca o valor 10 em **x**, descarta o sinal de divisão e atribui a **y** o valor 20.

Os comandos de formato podem especificar um modificador de largura máxima de campo. É um inteiro colocado entre o % e o código do comando de formato, limitando o número de caracteres lidos para qualquer campo. Por exemplo, se não deseja ler mais que 20 caracteres para **address**, você pode escrever

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

Se a stream de entrada fosse maior que 20 caracteres, uma chamada subseqüente à entrada começaria onde ela terminou. A entrada em um campo pode terminar antes que o comprimento máximo seja atingido, caso um espaço em branco seja encontrado. Nessse caso, **scanf()** move-se para o próximo campo.

Embora espaços, tabulações e novas linhas sejam utilizados como separadores de campo, eles são lidos como qualquer outro caractere quando se lê um único caractere. Por exemplo, com uma stream de entrada "**x y**",

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

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

Tenha cuidado: qualquer outro caractere na string de controle — incluindo espaços, tabulações e novas linhas — é usado para comparar e descartar caracteres da stream de entrada. Qualquer caractere que coincida é descartado. Por exemplo, dada a stream de entrada "**10t20**",

```
scanf("%st%s", &x, &y);
```

coloca 10 em **x** e 20 em **y**. O **t** é descartado por causa do **t** na string de controle.

O padrão C ANSI adicionou a **scanf()** uma nova característica chamada *scanset* que não fazia parte da versão original no UNIX. Um scanset define um conjunto de caracteres que podem ser lidos por **scanf()** e atribuídos à matriz de caracteres correspondente. Você define um scanset colocando uma string com os caracteres que você deseja receber entre colchetes. O colchete inicial deve ser precedido por um sinal de percentagem. Por exemplo, esse "scanset" instrui **scanf()** a ler apenas os caracteres **A**, **B** e **C**.

```
%[ABC]
```

Quando você usa um scanset, **scanf()** continua a ler caracteres e a colocá-los na matriz de caracteres correspondente até que um caractere que não esteja no scanset seja encontrado. A variável correspondente deve ser um ponteiro para uma matriz de caracteres. Quando **scanf()** retornar, a matriz conterá uma string terminada com um nulo e será formada pelos caracteres lidos.

Você pode especificar um conjunto complementar se o primeiro caractere do conjunto for um ^. Quando o ^ está presente, ele informa ao **scanf()** para aceitar qualquer caractere que *não* esteja definido no scanset.

Você pode especificar uma faixa por meio da utilização de um hífen. Por exemplo, isto informa a **scanf()** para aceitar as letras de A a Z.

```
%[A-Z]
```

Lembre-se de que um scanset é sensível à caixa das letras (distingue maiúsculas de minúsculas). Portanto, se deseja ler tanto letras maiúsculas como minúsculas, você deve especificá-las individualmente.

A função **scanf()** devolve um número igual à quantidade de campos que receberam valores com sucesso. Esse número não inclui os campos que foram lidos mas não atribuídos por meio do modificador *. É devolvido **EOF** se ocorre um erro antes que o primeiro campo seja atribuído.

Exemplo

A operação destes comandos **scanf()** é explicada nos comentários:

```
char str[80]; str2[80]
int i;

/* lê uma string e um inteiro */
scanf("%s%d", str, &i);

/* lê até 79 caracteres para str */
scanf("%79s", str);

/* salta o inteiro entre as duas strings */
scanf("%s%*d%s", str, str2);
```

Funções Relacionadas

printf(), fscanf()

#include <stdio.h>
void setbuf(FILE *stream, char *buf);

A função **setbuf()** determina o buffer que a stream especificada usará ou, se chamada com *buf* nulo, desativa o buffer. Se um buffer definido pelo usuário deve ser especificado, ele precisa ter **BUFSIZ** caracteres de comprimento. **BUFSIZ** é definido em STDIO.H.

A função **setbuf()** não devolve nenhum valor.

Exemplo

O fragmento seguinte associa um buffer definido pelo programador à stream apontada por **fp**.

```
char buffer[BUFSIZ];
.
.
.
setbuf(fp, buffer);
```

Funções Relacionadas

fopen(), fclose(), setvbuf()

#include <stdio.h>
int setvbuf(FILE *stream, char *buf, int mode, size_t size);

A função **setvbuf()** permite que o programador determine o buffer, seu tamanho e seu modo para uma stream especificada. A matriz de caracteres apontada por *buf* é usada como o buffer de stream para operações de E/S. O tamanho do buffer é estabelecido por *size*, e *mode* determina como o buffer será manipulado. Se *buf* é nulo, **setvbuf()** alocará seu próprio buffer.

Os valores legais para *mode* são **_IOFBF**, **_IONBF** e **_IOLBF**. Eles são definidos em STDIO.H. Quando o modo é colocado em **_IOFBF**, ocorre um uso total de buffers. Se o modo é **_IOLBF**, a stream terá um buffer de linha, o que significa que o buffer é esvaziado toda vez que um caractere de nova linha é escrito em streams de saída; para streams de entrada, uma solicitação de entrada lerá todos os caracteres até uma nova linha. Nos dois casos, o buffer também é esvaziado quando cheio. Quando o modo é colocado em **_IONBF**, não ocorre nenhuma operação com buffer.

O valor de *size* deve ser maior que zero.

A função **setvbuf()** devolve zero em caso de sucesso e um valor diferente de zero em caso de falha.

Exemplo

Este fragmento de código coloca a stream **fp** no modo de buffer de linha com um buffer de 128 bytes de tamanho:

```
#include <stdio.h>
char buffer[128];
.
.
.
setvbuf(fp, buffer, _IOLBF, 128);
```

Função Relacionada

setbuf()

#include <stdio.h>
int sprintf(char *buf, const char *format,...);

A função **sprintf()** é idêntica a **printf()**, exceto que a saída é colocada na matriz apontada por *buf*. Veja **printf()** para detalhes.

O valor de retorno é igual ao número de caracteres realmente colocado na matriz.

Exemplo

Após a execução desse fragmento de código, str conterá um, 2, 3:

```
char str[80];

sprintf(str, "%s %d %c", "um", 2, 3);
```

Funções Relacionadas

printf(), fsprintf()

#include <stdio.h>
int sscanf(const char *buf, const char *format,...);

A função **sscanf()** é idêntica a **scanf()**, mas os dados são lidos da matriz apontada por *buf* em lugar de **stdin**. Veja **scanf()** para detalhes.

O valor devolvido é igual ao número de variáveis, às quais foram realmente atribuídos valores. Esse número não inclui variáveis que foram saltadas devido ao uso do especificador de formato *. Um valor zero significa que nenhum campo foi atribuído; **EOF** indica que ocorreu um erro antes da primeira atribuição.

Exemplo

Este programa escreve a mensagem **Alo 1** na tela:

```
#include <stdio.h>

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

  sscanf("Alo 1 2 3 4 5", "%s%d", str, &i);
  printf("%s %d", str, i);
}
```

Funções Relacionadas

scanf(), fscanf()

#include <io.h>
long tell(int fd);

A função **tell()** faz parte do sistema de E/S tipo UNIX e não é definida pelo padrão C ANSI.

A função **tell()** devolve o valor atual do indicador de posição de arquivo associado ao descritor de arquivo *fd*. Esse valor é o número de bytes em que o indicador de posição se encontra a partir do início do arquivo. Um valor de retorno de -1 indica um erro.

Exemplo

Esse fragmento de código escreve o valor atual do indicador de posição para o arquivo descrito por **fd**:

```
long pos;
.
.
.
pos = tell(fd);
printf("A posição é %ld bytes a partir do início.", pos);
```

Funções Relacionadas

ftell(), lseek(), open(), close(), read(), write()

#include <stdio.h>
FILE *tmpfile(void);

A função **tmpfile()** abre um arquivo temporário para atualização e devolve um ponteiro para a stream. A função usa automaticamente um nome de arquivo diferente de todos os outros para evitar conflitos com arquivos existentes.

A função **tmpfile()** devolve um ponteiro nulo em caso de falha; caso contrário, devolve um ponteiro para a stream.

O arquivo temporário criado por **tmpfile()** é automaticamente removido quando o arquivo é fechado ou o programa termina.

Exemplo

O fragmento seguinte cria um arquivo de trabalho temporário.

```
FILE *temp;
```

```
if((temp=tmpfile())==NULL) {
  printf("arquivo temporário de trabalho não pode ser aberto.\n");
  exit(1);
}
```

Função Relacionada

tmpnam()

#include <stdio.h>
char *tmpnam(char *name);

A função **tmpnam()** gera um nome de arquivo diferente de todos os outros e armazena-o na matriz apontada por *name*. A principal finalidade de **tmpnam()** é gerar um nome de arquivo temporário que seja diferente de qualquer outro arquivo no diretório.

A função pode ser chamada até **TMP_MAX** vezes. **TMP_MAX** é definido em STDIO.H. Cada vez que **tmpnam()** é chamada, ela gera um novo nome de arquivo temporário.

Um ponteiro para *name* é devolvido em caso de sucesso; caso contrário, é devolvido um ponteiro nulo. Se *name* é NULL, então um ponteiro para uma região de memória alocada estaticamente que contém o nome do arquivo é devolvido.

Exemplo

Este programa mostra três nomes diferentes para arquivos temporários:

```
#include <stdio.h>

void main(void)
{
  char name[40];
  int i;

  for(i=0; i<3; i++) {
    tmpnam(name);
    printf("%s", name);
  }
}
```

Função Relacionada

tmpfile()

#include <stdio.h>
int ungetc(int ch, FILE *stream);

A função **ungetc()** devolve o caractere especificado no byte menos significativo de *ch* para a stream de entrada. Esse caractere é, então, devolvido pela próxima operação de leitura em *stream*. Uma chamada a **fflush()**, **fseek()**, **rewind ()** ou **fsetpos()** desfaz uma operação de **ungetc()** e descarta o caractere.

A devolução de um caractere é garantida, porém algumas implementações aceitam mais.

Você pode usar **"unget"** com um **EOF**.

Uma chamada a **ungetc()** limpa o indicador de fim de arquivo associado à stream. O valor do indicador de posição de arquivo para uma stream de texto é indefinido até que todos os caracteres devolvidos sejam lidos, quando será o mesmo de antes da primeira chamada a **ungetc()**. Para streams binárias, cada chamada a **ungetc()** decrementa o indicador de posição do arquivo.

O valor devolvido é igual a ch, em caso de sucesso, ou **EOF** em caso de falha.

Exemplo

Essa função lê apenas palavras da stream de entrada apontada por **fp.** O caractere de terminação é devolvido à stream para uso posterior. Por exemplo, dada a entrada **count/10**, a primeira chamada a **read_word()** devolve **count** e coloca "/" de volta na stream de entrada.

```
void read_word(FILE *fp, char *token)
{

  while(isalpha(*token=getc(fp))) token++;
  ungetc(fp, *token);
}
```

Função Relacionada

getc()

#include <io.h>
int unlink(const char *fname);

A função **unlink()** faz parte do sistema de E/S tipo UNIX e não é definida pelo padrão C ANSI.

A função **unlink()** remove o arquivo especificado do diretório. Ela devolve zero em caso de sucesso e -1 em caso de falha.

Exemplo

Este programa apaga o arquivo especificado no primeiro argumento da linha de comando.

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

void main(int argc, char *argv[])
{
 if(unlink(argv[1])==-1) printf("O arquivo não pode ser apagado.");
}
```

Funções Relacionadas

open(), close()

#include <stdarg.h>
#include <stdio.h>
int vprintf(const char *format, va_list arg_ptr);
int vfprintf(FILE *stream, const char *format, va_list arg_ptr);
int vsprintf(char *buf, const char *format, va_list arg_ptr);

As funções **vprintf()**, **vfprintf()** e **vsprintf()** são funcionalmente equivalentes a **printf()**, **fprintf()** e **sprintf()**, respectivamente. Porém, a lista de argumentos foi substituída por um ponteiro para uma lista de argumentos. Esse ponteiro deve ser do tipo **va_list** e é definido em STDARG.H. Veja **va_arg()**, **va_start()** e **va_end()**, no Capítulo 18, para mais informações sobre a passagem de argumentos de comprimentos variáveis para funções.

Exemplo

Este fragmento de código mostra como estabelecer uma chamada a **vprintf()**. A chamada a **va_start()** cria um ponteiro a argumentos de comprimento variável para o início da lista de argumentos. Esse ponteiro deve ser usado na chamada a **vprintf()**. A chamada a **va_end()** limpa o ponteiro para o argumento de comprimento variável.

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



```
void print_message(char *format, ...);

void main(void)
{
  print_message("O arquivo não pode ser aberto %s.", "teste");
}

void print_message(char *format, ...);
{
  va_list ptr; /* obtém um ponteiro para arg prt */

  /*inicializa ptr para apontar para o primeiro argumento após
    a string de formato
  */
  va_start(ptr, format);

  /* escreve a mensagem */
  vprintf(format, ptr);

  va_end(ptr);
}
```

Funções Relacionadas

va_list(), va_start(), va_end()

#include <io.h>
int write(int fd, char *buf, unsigned count);

A função **write()** faz parte do sistema de E/S tipo UNIX e não é definida pelo padrão C ANSI.

A função **write()** escreve *count* bytes no arquivo descrito por *fd* do buffer apontado por *buf*. O indicador de posição de arquivo é aumentado de acordo com o número de bytes escritos. Se o arquivo for aberto no modo texto, poderão ocorrer traduções de caracteres.

O valor devolvido será igual ao número de bytes realmente escritos. Esse número pode ser menor que *count* se for encontrado um erro. Um valor -1 significa que ocorreu um erro.

A função **write()** tende a ser dependente da implementação e pode ter um comportamento diferente do exposto aqui. Verifique seu manual do usuário para detalhes.

Exemplo

Esse programa escreve 100 bytes de **buffer** no arquivo TEST.

```
#include <io.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>

void main(void)
{
  int fd;
  char buffer[100];

  if((fd=open("test", O_WRONLY))==-1) {
    printf("O arquivo não pode ser aberto.\n");
    exit(1);
  }

  gets(buffer);

  if(write(fd, buffer, 100)!=100) printf("Erro de escrita");
  close(fd);
}
```

Funções Relacionadas

read(), close(), fwrite(), lseek()

13

Funções de String e de Caracteres

A biblioteca C padrão tem um conjunto rico e variado de funções para manipulação de string e de caracteres. Em C, uma string é uma matriz de caracteres terminada com um nulo. Em uma implementação padrão, as funções de string exigem o arquivo de cabeçalho STRING.H para fornecer seus protótipos. As funções de caracteres usam CTYPE.H como arquivo de cabeçalho. Todas as funções descritas neste capítulo são definidas pelo padrão C ANSI.

Como C não tem verificação de limites em operações com matrizes, é responsabilidade do programador evitar o estouro da matriz. De acordo com o padrão C ANSI, se ocorre um estouro em uma matriz, seu "comportamento é indeterminado" — uma maneira sutil de dizer que seu programa está para quebrar!

Em C, um *caractere que pode ser impresso* é aquele que pode ser mostrado em um terminal. Normalmente são os caracteres entre um espaço (0x20) e o "tilde" (0x7E). Os caracteres de controle têm valores entre 0 e 0x1F além do DEL (0x7F).

Os argumentos para as funções de caractere são tradicionalmente inteiros. No entanto, apenas o byte menos significativo é usado; a função de caractere converte automaticamente o argumento em **unsigned char**. Apesar disso, você pode chamar essas funções com argumentos de tipo caractere, porque caracteres são automaticamente promovidos a inteiros no momento da chamada.

O arquivo de cabeçalho STRING.H define o tipo **size_t**, que é essencialmente o mesmo que um **unsigned**.

#include <ctype.h>
int isalnum(int ch);

A função **isalnum()** devolve um valor diferente de zero se o argumento for uma letra ou um dígito. Se o caractere não for alfanumérico, **isalnum()** devolverá zero.

Exemplo

Este programa verifica cada caractere lido de **stdin** e apresenta todos os caracteres alfanuméricos:

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

void main(void)
{
  char ch;

  for(;;) {
    ch = getc(stdin);
    if(ch==' ') break;
    if(isalnum(ch)) printf("%c é alfanumérico\n", ch);
  }
}
```

Funções Relacionadas

isalpha(), iscntrl(), isdigit(), isgraph(), isprint(), ispunct(), isspace()

#include <ctype.h>
int isalpha(int ch);

A função **isalpha()** devolverá um valor diferente de zero se *ch* for uma letra do alfabeto; caso contrário, devolverá zero. O que constitui uma letra pode variar de língua para língua — em inglês, são letras as maiúsculas e minúsculas de A a Z.

Exemplo

Este programa verifica cada caractere lido do **stdin** e apresenta todas as letras.

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

void main(void)
{
```



```
  char ch;

  for(;;) {
    ch = getchar();
    if(ch==' ') break;
    if(isalpha(ch)) printf("%c é uma letra\n", ch);
  }
}
```

Funções Relacionadas

isalnum(), iscntrl(), isdigit(), isgraph(), isprint(), ispunct(), isspace()

#include <ctype.h>
int iscntrl(int ch);

A função **iscntrl()** devolve um valor diferente de zero se *ch* está entre 0 e 0x1F ou é igual a 0x7F (DEL); caso contrário, devolve zero.

Exemplo

Essa função verifica cada caractere lido de **stdin** e apresenta todos os caracteres de controle:

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

void main(void)
{
  char ch;
  for(;;) {
    ch = getchar();
    if(ch==' ') break;
    if(iscntrl(ch)) printf("%c é um caractere de controle\n", ch);
  }
}
```

Funções Relacionadas

isalnum(), isalpha(), isdigit(), isgraph(), isprint(), ispunct(), isspace()

#include <ctype.h>
int isdigit(int ch);

A função **isdigit()** devolve um valor diferente de zero se *ch* for um dígito (isto é, de zero a 9). Caso contrário, devolve zero.

Exemplo

Este programa verifica cada caractere lido de **stdin** e apresenta todos os dígitos:

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

void main(void)
{
  char ch;

  for(;;) {
    ch = getchar();
    if(ch==' ') break;
    if(isdigit(ch)) printf("%c é um dígito\n", ch);
  }
}
```

Funções Relacionadas

isalnum(), isalpha(), iscntrl(), isgraph(), isprint(), ispunct(), isspace()

#include <ctype.h>
int isgraph(int ch);

A função **isgraph()** devolve um valor diferente de zero se *ch* é qualquer caractere que pode ser impresso, com exceção do espaço; caso contrário, devolve zero. Embora isso dependa da implementação, os caracteres que podem ser impressos estão geralmente na faixa de 0x21 a 0x7E.

Exemplo

Este programa verifica cada caractere lido de **stdin** e apresenta todos os caracteres que podem ser impressos:

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

void main(void)
```



```
{
  char ch;

  for(;;) {
    ch = getchar();
    if(isgraph(ch)) printf("%c pode ser impresso\n", ch);
    if(ch==' ') break;
  }
}
```

Funções Relacionadas

isalnum(), isalpha(), iscntrl(), isdigit(), isprint(), ispunct(), isspace()

#include <ctype.h>
int islower(int ch);

A função **islower()** devolve um valor diferente de zero se *ch* é uma letra minúscula; caso contrário, devolve zero.

Exemplo

Este programa verifica cada caractere lido de **stdin** e apresenta todas as letras minúsculas:

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

void main(void)
{
  char ch;

  for(;;) {
    ch = getchar();
    if(ch==' ') break;
    if(islower(ch)) printf("%c é minúsculo\n", ch);
  }
}
```

Função Relacionada

isupper()

#include <ctype.h>
int isprint(int ch);

A função **isprint()** devolve um valor diferente de zero, se *ch* é um caractere que pode ser impresso, incluindo um espaço; caso contrário, devolve zero. Embora dependentes da implementação, os caracteres que podem ser impressos geralmente estão na faixa de 0x20 a 0x7E.

Exemplo

Este programa verifica cada caractere lido de **stdin** e apresenta todos os caracteres que podem ser impressos:

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

void main(void)
{
  char ch;

  for(;;) {
    ch = getchar();
    if(isprint(ch)) printf("%c pode ser impresso\n", ch);
    if(ch==' ') break;
  }
}
```

Funções Relacionadas

isalnum(), isalpha(), iscntrl(), isdigit(), isgraph(), ispunct(), isspace()

#include <ctype.h>
int ispunct(int ch);

A função **ispunct()** devolve um valor diferente de zero se *ch* é um caractere de pontuação; caso contrário, devolve zero. O termo *pontuação*, como definido pela função, inclui todos os caracteres que podem ser impressos e não sejam alfanuméricos nem espaço.

Exemplo

Este programa verifica cada caractere lido de **stdin** e apresenta todos os caracteres de pontuação:


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

void main(void)
{
  char ch;

  for(;;) {
    ch = getchar();
    if(ch==' ') break;
    if(ispunct(ch)) printf("%c é pontuação\n", ch);
  }
}
```

Funções Relacionadas

isalnum(), isalpha(), iscntrl(), isdigit(), isgraph(), isspace()

#include <ctype.h>
int isspace(int ch);

A função **isspace()** devolverá um valor diferente de zero se *ch* for um espaço, tabulação horizontal, tabulação vertical, alimentação de formulário, retorno de carro ou caractere de nova linha; caso contrário, devolverá zero.

Exemplo

Este programa verifica cada caractere lido de **stdin** e apresenta todos os caracteres de espaço em branco:

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

void main(void)
{
  char ch;

  for(;;) {
    ch = getchar();
    if(isspace(ch)) printf("%c é um espaço em branco\n", ch);
    if(ch==' ') break;
  }
}
```

Funções Relacionadas

isalnum(), isalpha(), iscntrl(), isdigit(), isgraph(), ispunct()

#include <ctype.h>
int isupper(int ch);

A função **isupper()** devolverá um valor diferente de zero se *ch* for uma letra maiúscula; caso contrário, devolverá zero.

Exemplo

Este programa verifica cada caractere lido de **stdin** e apresenta todas as letras maiúsculas:

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

void main(void)
{
  char ch;

  for(;;) {
    ch = getchar();
    if(ch== ' ') break;
    if(isupper(ch)) printf("%c é maiúsculo\n", ch);
  }
}
```

Função Relacionada

islower()

#include <ctype.h>
int isxdigit(int ch);

A função **isxdigit()** devolve um valor diferente de zero se *ch* é um dígito hexadecimal; caso contrário, devolve zero. Um dígito hexadecimal está na faixa de "A" a "F", de "a" "a" "f" ou de 0 a 9.

Exemplo

Este programa verifica cada caractere lido de **stdin** e apresenta todos os dígitos hexadecimais:


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

void main(void)
{
  char ch;

  for(;;) {
    ch = getchar();
    if(ch== ' ') break;
    if(isxdigit(ch)) printf("%c é hexadecimal\n", ch);
  }
}
```

Funções Relacionadas

isalnum(), isalpha(), iscntrl(), isdigit(), isgraph(), ispunct(), isspace()

#include <string.h>
void *memchr(const void *buffer, int ch, size_t count);

A função **memchr()** procura, na matriz apontada por *buffer*, pela primeira ocorrência de *ch* nos primeiros *count* caracteres.

A função **memchr()** devolve um ponteiro para a primeira ocorrência de *ch* em *buffer* ou um ponteiro nulo se *ch* não for encontrado.

Exemplo

Este programa escreve **isto e um teste** na tela:

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

void main(void)
{
  char *p;

  p = memchr("isto é um teste", ' ', 14);
  printf(p);
}
```

Funções Relacionadas

memcpy(), memmove()

#include <string.h>
int memcmp(const void *buf1, const void *buf2, size_t count);

A função **memcmp()** compara os primeiros *count* caracteres das matrizes apontadas por *buf1* e *buf2*. A comparação é feita lexicograficamente.

A função **memcmp()** devolve um inteiro, que é interpretado como indicado a seguir:

Valor	Significado
Menor que zero	*buf1* é menor que *buf2*
Zero	*buf1* é igual a *buf2*
Maior que zero	*buf1* é maior que *buf2*

Exemplo

Este programa mostra o resultado de uma comparação entre seus dois argumentos de linha de comando:

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

void main(int argc, char *argv[])
{
  int outcome, len, l1,  l2;

  if(argc!=3) {
    printf("Número incorreto de parâmetros.");
    exit(1);
  }

  /* encontra o comprimento da menor string */
  l1 = strlen(argv[1]);
  l2 = strlen(argv[2]);
  len = l1 < l2 ? l1 : l2;

  outcome = memcmp(argv[1], argv[2], len);
  if(!outcome) printf ("Iguais");
  else if(outcome<0) printf("Primeiro menor que segundo.");
  else printf("Primeiro maior que segundo.");
}
```


Funções Relacionadas

memchr(), memcpy(), strcmp()

#include <string.h>
void *memcpy(void *to, const void *from, size_t count);

A função **memcpy()** copia *count* caracteres da matriz apontada por *from* para a matriz apontada por *to*. Se as matrizes se sobrepõem é indefinido, o comportamento de **memcpy()**.

A função **memcpy()** devolve um ponteiro para *to*.

Exemplo

Este programa copia o conteúdo de buf1 em buf2 e mostra o resultado:

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

#define SIZE 80

void main(void)
{
  char buf1[SIZE], buf2[SIZE];

  strcpy(buf1, "Quando, no curso do ...");
  memcpy(buf2, buf1, SIZE);
  printf(buf2);
}
```

Função Relacionada

memmove()

#include <string.h>
void *memmove(void *to, const void *from, size_t count);

A função **memmove()** copia *count* caracteres da matriz apontada por *from* para a matriz apontada por *to*. Se as matrizes se sobrepõem, a cópia ocorrerá corretamente, colocando o conteúdo correto em *to*, porém *from* será modificado.

A função **memmove()** devolve um ponteiro para *to*.

Exemplo

Este programa copia o conteúdo de **str1** em **str2** e mostra o resultado.

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

#define SIZE 80

void main(void)
{
  char str1[SIZE], str2[SIZE];

  strcpy(str1, "Quando, no curso do...");
  memmove(str2, str1, SIZE);
  printf(str2);
}
```

Função Relacionada

memcpy()

#include <string.h>
void *memset(void *buf, int ch, size_t count);

A função **memset()** copia o byte menos significativo de *ch* nos primeiros *count* caracteres da matriz apontada por *buf*. Ela devolve *buf*.

O uso mais comum de **memset()** é na inicialização de uma região de memória com algum valor conhecido.

Exemplo

Este fragmento inicializa com nulo os 100 primeiros bytes da matriz apontada por **buf**. Em seguida, coloca X nos 10 primeiros bytes e mostra a string XXXXX XXXXX.

```
memset(buf, '\0', 100);
memset(buf, 'X', 10);
printf(buf);
```

Funções Relacionadas

memcmp(), memcpy(), memmove()

#include <string.h>
char *strcat(char *str1, const char *str2);

A função **strcat()** concatena uma cópia de *str2* em *str1* e termina *str1* com um nulo. O terminador nulo, que originalmente finalizava *str1* é sobreposto pelo primeiro caractere de *str2*. A string *str2* permanece inalterada na operação. Se as matrizes se sobrepõem, o comportamento de **strcat()** é indefinido.

A função **strcat()** devolve *str1*.

Lembre-se de que não ocorre nenhuma verificação de limites. É de sua responsabilidade garantir que *str1* seja suficientemente grande para armazenar seu conteúdo original e o de *str2*.

Exemplo

Este programa acrescenta a primeira string lida de **stdin** à segunda. Por exemplo, assumindo que o usuário tenha digitado **alo** e **aqui**, o programa escreve **aquialo**.

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

void main(void)
{
  char s1[80], s2[80];

  gets(s1);
  gets(s2);

  strcat(s2, s1);
  printf(s2);
}
```

Funções Relacionadas

strchr(), strcmp(), strcpy()

#include <string.h>
char *strchr(const char *str, int ch);

A função **strchr()** devolve um ponteiro à primeira ocorrência do byte menos significativo de *ch* na string apontada por *str*. Se não for encontrada nenhuma coincidência, será devolvido um ponteiro nulo.

Exemplo

Este programa escreve a string **isto é um teste**:

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

void main(void)
{
  char *p;

  p = strchr("isto é um teste", ' ');
  printf(p);
}
```

Funções Relacionadas

strpbrk(), strspn(), strstr(), strtok()

#include <string.h>
int strcoll(const char *str1, const char *str2);

A função **strcoll()** compara a string apontada por *str1* com aquela apontada por *str2*. A comparação é efetuada de acordo com a localidade especificada, usando-se a função **setlocale()**. (Veja **setlocale()** para detalhes.)

A função **strcoll()** devolve um inteiro, que é interpretado como indicado a seguir:

Valor	Significado
Menor que zero	*str1* é menor que *str2*
Zero	*str1* é igual a *str2*
Maior que zero	*str1* é maior que *str2*

Exemplo

Este fragmento de código escreve igual na tela:

```
if(strcoll("oi", "oi")) printf("igual");
```

Funções Relacionadas

memcmp(), strcmp()

#include <string.h>
int strcmp(const char *str1, const char *str2);

A função **strcmp()** compara lexicograficamente duas strings e devolve um inteiro baseado no resultado, como mostrado aqui:

Valor	Significado
Menor que zero	*str1* é menor que *str2*
Zero	*str1* é igual a *str2*
Maior que zero	*str1* é maior que *str2*

Exemplo

Você pode usar a função seguinte como uma rotina de verificação de senha. Ela devolve zero em caso de falha e 1 em caso de sucesso.

```
password(void)
{
  char s[80];

  printf("digite a senha: ")
  gets(s);

  if(strcmp(s, "pass")) {
    printf("senha inválida\n");
    return 0;
  }
  return 1;
}
```

Funções Relacionadas

strchr(), strcpy(), strncmp()

#include <string.h>
char *strcpy(char *str1, const char *str2);

A função **strcpy()** copia o conteúdo de *str2* em *str1*. *str2* deve ser um ponteiro para uma string terminada com um nulo. A função **strcpy()** devolve um ponteiro para *str1*.

Se *str1* e *str2* se sobrepõem, o comportamento de **strcpy()** é indefinido.

Ainda, a área de memória apontada por *str1* deve ser grande o suficiente para conter a string apontada por *str2*.

Exemplo

O fragmento de código seguinte copia **alo** na string str:

```
char str[80];
strcpy(str, "alo");
```

Funções Relacionadas

memcpy(), strchr(), strcmp(), strncmp()

#include <stdio.h>
size_t strcspn(const char *str1, const char *str2);

A função **strcspn()** devolve o comprimento da substring inicial da string apontada por *str1*, que é formada apenas pelos caracteres não contidos na string apontada por *str2*. Expondo de forma diferente, **strcspn()** devolve o índice do primeiro caractere da string apontada por *str1* que coincide com qualquer um dos caracteres da string apontada por *str2*.

Exemplo

Este programa escreve o número **7**.

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

void main(void)
{
  int len;

  len = strcspn("isto é um teste", "uz");
  printf("%d", len);
}
```

Funções Relacionadas

strbrk(), strrchr(), strstr(), strtok()

#include <string.h>
char *strerror(int errnum);

A função **strerror()** devolve um ponteiro para uma string definida pela implementação, que é associada ao valor de *errnum*. Sob nenhuma circunstância você deve modificar a string.

Exemplo

Este fragmento de código escreve na tela uma mensagem de erro definida pela implementação.

```
printf(strerror(10));
```

#include <string.h>
size_t strlen(const char *str);

A função **strlen()** devolve o comprimento da string terminada por um nulo apontada por *str*. O nulo não é contado.

Exemplo

O fragmento de código seguinte escreve **3** na tela:

```
printf("%d", strlen("alo"));
```

Funções Relacionadas

memcpy(), strchr(), strcmp(), strncmp()

#include <string.h>
char *strncat(char *str1, const char *str2, size_t count);

A função **strncat()** concatena não mais que *count* caracteres da string apontada por *str2* à string apontada por *str1* e termina *str1* com um nulo. O terminador nulo que originalmente finalizava *str1* é sobreposto pelo primeiro caractere de *str2*. A string *str2* permanece inalterada com a operação. Se as strings se sobrepõem, o comportamento de **strncat()** é indefinido.

A função **strncat()** devolve *str1*.

Lembre-se de que não ocorre nenhuma verificação de limite, portanto é de sua responsabilidade assegurar que *str1* seja suficientemente grande para armazenar seu conteúdo original como também o de *str2*.

Exemplo

Este programa acrescenta a primeira string à segunda e evita que ocorra o estouro da matriz **s1**. Por exemplo, assumindo que o usuário digite **alo** e **aqui**, o programa escreve **aquialo**.

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

void main(void)
{
  char s1[80], s2[80];
  unsigned int len;

  gets(s1);
  gets(s2);

  /* calcula quantos caracteres caberão */
  len = 79-strlen(s2);

  strncat(s2, s1, len);
  printf(s2);
}
```

Funções Relacionadas

strcat(), strnchr(), strncmp(), strncpy()

#include <string.h>
int strncmp(const char *str1, const char *str2, size_t count);

A função **strncmp()** compara lexicograficamente não mais que *count* caracteres das duas strings terminadas com nulo e devolve um inteiro baseado no resultado, como mostrado aqui:

Valor	**Significado**
Menor que zero	*str1* é menor que *str2*
Zero	*str1* é igual a *str2*
Maior que zero	*str1* é maior que *str2*

Se há menos do que *count* caracteres em uma das strings, a comparação termina quando o primeiro nulo for encontrado.

Exemplo

A função seguinte compara os oito primeiros caracteres de dois argumentos da linha de comando e indica se são iguais:

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

void main(int argc, char *argv[])
{
  if(argc!=3) {
    printf("Número incorreto de parâmetros.");
    exit(1);
  }

  if(!strncmp(argv[1], argv[2], 8))
    printf("Os nomes de arquivo são iguais.\n");
}
```

Funções Relacionadas

strcmp(), strnchr(), strncpy()

#include <string.h>
char *strncpy(char *str1, const char *str2, size_t count);

A função **strncpy()** copia até *count* caracteres da string apontada por *str2* na string apontada por *str1*. *str2* deve ser um ponteiro para uma string terminada com um nulo.

Se *str1* e *str2* se sobrepõem, o comportamento de **strncpy()** é indefinido.

Se a string apontada por *str2* tiver menos que *count* caracteres, serão acrescidos nulos a *str1* até que um total de *count* caracteres tenham sidos copiados.

Alternativamente, se a string apontada por *str2* for maior que *count* caracteres, então a string resultante, apontada por *str1*, não é terminada em nulo.

A função **strncpy()** devolve um ponteiro para *str1*.

Exemplo

O fragmento de código seguinte copia no máximo 79 caracteres de **str1** em **str2**, garantindo, assim, que não ocorrerá nenhum estouro de limite de matriz.

```
char str1[128], str2[80];

gets(str1);
strncpy(str2, str1, 79);
```

Funções Relacionadas

memcpy(), strchr(), strncat(), strncmp()

#include <string.h>
char *strpbrk(const char *str1, const char *str2);

A função **strpbrk()** devolve um ponteiro para o primeiro caractere da string apontada por *str1* que coincide com qualquer caractere da string apontada por *str2*. Os terminadores nulos não são incluídos. Se não há nenhuma coincidência, é devolvido um ponteiro nulo.

Exemplo

Este programa escreve a mensagem **o é um teste** na tela:

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

void main(void)
{
  char *p;

  p = strpbrk("isto é um teste", "obkj");
  printf(p);
}
```

Funções Relacionadas

strrchr(), strspn(), strstr(), strtok()

#include <string.h>
char *strrchr(const char *str, int ch);

A função **strrchr()** devolve um ponteiro para a última ocorrência do byte menos significativo de *ch* na string apontada por *str*. Se não for encontrada nenhuma coincidência, um ponteiro nulo será devolvido.

Exemplo

Este programa escreve a mensagem **o é um teste**.

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

void main(void)
{
  char *p;

  p = strrchr("isto é um teste", 'o');
  printf(p);
}
```

Funções Relacionadas

strpbrk(), strspn(), strstr(), strtok()

#include <string.L>
size_t strspn(const char *str1, const char *str2);

A função **strspn()** devolve o comprimento da substring inicial da string apontada por *str1*, que consiste apenas em caracteres contidos na string apontada por *str2*. Exposto de forma diferente, **strspn()** devolve um índice para o primeiro caractere na string apontada por *str1* que não coincide com nenhum dos caracteres da string apontada por *str2*.

Exemplo

Este programa escreve 8:

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

void main(void)
{
  int len;

  len = strspn("isto é um teste", "otsi ");
  printf("%d", len);
}
```

Funções Relacionadas

strpbrk(), strrchr(), strstr(), strtok()

#include <string.h>
char *strstr(const char *str1, const char *str2);

A função **strstr()** devolve um ponteiro para a primeira ocorrência da string apontada por *str2* na string apontada por *str1*. Ela devolve um ponteiro nulo se não for encontrada nenhuma coincidência.

Exemplo

Este programa mostra a mensagem **to é um teste**.

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

void main(void)
{
  char *p;

  p = strstr("isto é um teste", "to");
  printf(p);
}
```

Funções Relacionadas

strchr(), strcspn(), strpbrk(), strrchr(), strspn(), strtok()

#include <string.h>
char *strtok(char *str1, const char *str2);

A função **strtok()** retorna um ponteiro para a próxima palavra na string apontada por *str1*. Os caracteres que compõem a string *str2* definem os delimitadores que separam cada palavra da seguinte. Por exemplo, dada a string:

Um, dois, e três.

As palavras são **um**, **dois**, **e**, e **três**. Os delimitadores são os espaços, a vírgula e o ponto.

strtok() retorna um ponteiro nulo quando não há mais palavras em *str1*.

Na primeira vez em que **strtok()** é chamada, *str1* é realmente utilizada na chamada. Chamadas subseqüentes devem usar um ponteiro nulo como primeiro argumento.

É importante observar que a função **strtok()** modifica a string apontada por *str1*. Toda vez que uma palavra é encontrada, é colocado um nulo onde o delimitador foi encontrado. Dessa forma, **strtok()** pode continuar a avançar pela string.

Você pode usar um conjunto diferente de delimitadores para cada chamada a **strtok()**.

Exemplo

Este programa separa as palavras da string "O soldado de verão, o patriota da luz do dia", com espaços e vírgulas como delimitadores. O resultado é

"O¦soldado¦de¦verão¦o¦patriota¦da¦luz¦do¦dia"

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

void main(void)
{
  char *p;

  p = strtok("O soldado de verão, o patriota da luz do dia", " ");
  printf(p);
  do {
    p = strtok('\0', ", ");
    if(p) printf("¦%s", p);
  } while (p);
}
```

Funções Relacionadas

strchr(), strcspn(), strpbrk(), strrchr(), strspn()

#include <string.h>
size_t strxfrm(char *str1, const char *str2, size_t count);

A função **strxfrm()** transforma os primeiros *count* caracteres da string apontada por *str2* de forma que ela possa ser usada pela função **strcmp()**. **strxfrm()** coloca, em seguida, o resultado na string apontada por *str1*. Após a transformação, o resultado de um **strcmp()**, usando *str1*, e de um **strcoll()**, usando a string original, apontada por *str2* será igual. O principal uso da função **strxfrm()** é em ambientes de língua estrangeira, que não usam a seqüência ASCII.

A função **strxfrm()** devolve o comprimento da matriz transformada.

Exemplo

A linha seguinte transforma os dez primeiros caracteres da string apontada por s2 e coloca o resultado na string apontada por s1.

```
strxfrm(s1, s2, 10);
```

Função Relacionada

strcoll()

#include <ctype.h>
int tolower(int ch);

A função **tolower()** devolve o equivalente minúsculo de *ch* se *ch* é uma letra; caso contrário, *ch* é devolvido sem alteração.

Exemplo

Este fragmento de código mostra q.

```
putchar(tolower('Q'));
```

Função Relacionada

toupper()

#include <ctype.h>
int toupper(int ch);

A função **toupper()** devolve o equivalente maiúsculo de *ch* se *ch* é uma letra; caso contrário, *ch* é devolvido sem alteração.

Exemplo

Este código mostra A.

```
putchar(toupper('a'));
```

Função Relacionada

tolower()

MAKRON Books

14

Funções Matemáticas

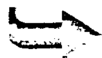
O padrão C ANSI define 22 funções matemáticas que se encontram nas seguintes categorias:

- Funções trigonométricas
- Funções hiperbólicas
- Funções exponenciais e logarítmicas
- Miscelâneas

Estas funções estão descritas neste capítulo. Mesmo que seu compilador não siga completamente o padrão, as funções matemáticas descritas aqui são provavelmente aplicáveis.

Todas as funções matemáticas exigem o arquivo de cabeçalho MATH.H. Além de declarar os protótipos das funções matemáticas, esse cabeçalho define a macro **HUGE_VAL**. As funções matemáticas utilizam com freqüência as macros **EDOM** e **ERANGE**, que são definidas no arquivo de cabeçalho **ERRNO.H**. Se um argumento para uma função matemática não está no domínio para o qual ele é definido, um valor definido pela implementação é devolvido e a variável global inteira **errno** é ajustada para que seja igual a **EDOM**. Se a rotina produz um resultado muito grande para ser representado por um **double**, ocorrerá um estouro. Isso faz com que a rotina devolva **HUGE_VAL** e ajuste **errno** para que seja igual a **ERANGE**, indicando um erro de escala. Se seu compilador não segue o padrão C ANSI, a operação exata das rotinas em situação de erro pode ser diferente.

No que se refere às funções matemáticas, todos os ângulos são expressos em radianos.



*NOTA: Para ter acesso a **errno** e às macros EDOM e ERANGE, você deve incluir ERRNO.H em seu programa.*

#include <math.h>
double acos(double arg);

A função **acos()** devolve o arco co-seno de *arg*. O argumento de **acos()** deve estar na faixa de -1 a 1; caso contrário, ocorrerá um erro de domínio.

Exemplo

Este programa escreve os arcos co-senos dos valores de -1 a 1 em incrementos de um décimo.

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

void main(void)
{
  double val=-1.0;

  do {
    printf("Arco co-seno de %f é %f.\n", val, acos(val));
    val += 0.1;
  } while(val<=1.0);
}
```

Funções Relacionadas

asin(), atan(), atan2(), cos(), cosh(), sin(), sinh(), tan(), tanh()

#include <math.h>
double asin(double arg);

A função **asin()** devolve o arco seno de *arg*. O argumento de **asin()** deve estar na faixa de -1 a 1; caso contrário, ocorrerá um erro de domínio.

Exemplo

Este programa escreve os arcos senos dos valores de -1 a 1 em incrementos de um décimo.

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



```
void main(void)
{
  double val=-1.0;

  do {
    printf("Arco seno de %f é %f.\n", val, asin(val));
    val += 0.1;
  } while(val<=1.0);
}
```

Funções Relacionadas

acos(), atan(), atan2(), cos(), cosh(), sin(), sinh(), tan(), tanh()

#include <math.h>
double atan(double arg);

A função **atan()** devolve o arco tangente de *arg*.

Exemplo

Este programa escreve os arcos tangentes dos valores de -1 a 1 em incrementos de um décimo.

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

void main(void)
{
  double val=-1.0;

  do {
    printf("Arco tangente de %f é %f.\n", val, atan(val));
    val += 0.1;
  } while(val<=1.0);
}
```

Funções Relacionadas

acos(), asin(), atan2(), cos(), cosh(), sin(), sinh(), tan(), tanh()

#include <math.h>
double atan2(double y, double x);

A função **atan2()** devolve o arco tangente de *y/x*. Ela usa o sinal dos argumentos para calcular o quadrante do valor devolvido.

Exemplo

Este programa escreve os arcos tangentes de *y*, de -1 a 1, em incrementos de um décimo.

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

void main(void)
{
  double val=-1.0;

  do {

    printf("Atan2 de %f é %f.\n", val, atan2(val,1.0));
    val += 0.1;
  } while(val<=1.0);
}
```

Funções Relacionadas

acos(), asin(), atan(), cos(), cosh(), sin(), sinh(), tan(), tanh()

#include <math.h>
double ceil(double num);

A função **ceil()** devolve o menor inteiro, representado como um double, que não seja menor que *num*. Por exemplo, dado 1.02, **ceil()** devolverá **2.0**. Dado -1.02, **ceil()** devolverá **-1**.

Exemplo

Este fragmento de código escreve **10** na tela:

```
printf("%f", ceil(9.9));
```

Funções Relacionadas

floor(), fmod()

#include <math.h>
double cos(double arg);

A função **cos()** devolve o co-seno de *arg*. O valor de *arg* deve estar em radianos.

Exemplo

Este programa escreve os co-senos dos valores de -1 a 1 em incrementos de um décimo:

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

void main(void)
{
  double val=-1.0;

  do {
    printf("Co-seno de %f é %f.\n", val, cos(val));
    val += 0.1;
  } while(val<=1.0);
}
```

Funções Relacionadas

acos(), asin(), atan(), atan2(), cosh(), sin(), sinh(), tan(), tanh()

#include <math.h>
double cosh(double arg);

A função **cosh()** devolve o co-seno hiperbólico de *arg*.

Exemplo

O programa seguinte escreve os co-senos hiperbólicos dos valores de -1 a 1 em incrementos de um décimo:

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

void main(void)
{
  double val=-1.0;

  do {
    printf("Co-seno hiperbólico de %f é %f.\n", val, cosh(val));
    val += 0.1;
  } while(val<=1.0);
}
```

Funções Relacionadas

acos(), asin(), atan(), atan2(), cos(), sin(), tan(), tanh()

#include <math.h>
double exp(double arg);

A função **exp()** devolve o logaritmo natural **e** elevado à potência *arg*.

Exemplo

Este fragmento de código mostra **e** (arredondado para 2.718282):

```
printf("valor de e à primeira: %f", exp(1.0));
```

Função Relacionada

log()

#include <math.h>
double fabs(double num);

A função **fabs()** devolve o valor absoluto de *num*.

Exemplo

Este programa mostra **1.0 1.0** na tela:

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

void main(void)
{
  printf("%1.1f %1.1f", fabs(1.0), fabs(-1.0));
}
```

Função Relacionada

abs()

#include <math.h>
double floor(double num);

A função **floor()** devolve o maior inteiro (representado como um **double**) que não seja maior que *num*. Por exemplo, dado 1.02, **floor()** devolve **1.0**. Dado -1.02, **floor()** devolve **-2.0**.

Exemplo

Este fragmento de código escreve **10** na tela:

```
printf("%f", floor(10.9));
```

Funções Relacionadas

fceil(), fmod()

#include <math.h>
double fmod(double x, double y);

A função **fmod()** devolve o resto de *x/y*.

Exemplo

O programa seguinte escreve **1.0** na tela — o resto de 10/3.

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

void main(void)
{
  printf("%1.1f", fmod(10.0, 3.0));
}
```

Funções Relacionadas

ceil(), fabs(), floor()

#include <math.h>
double frexp(double num, int *exp);

A função **frexp()** decompõe o número *num* em uma mantissa, na faixa de 0.5 a 1, e em um expoente inteiro tal que $num = mantissa * 2^{exp}$. A mantissa é devolvida pela função e o expoente é colocado na variável apontada por *exp*.

Exemplo

Este fragmento de código escreve **0.625** para a mantissa e **4** para o expoente:

```
int e;
double f;

f = frexp(10.0, &e);
printf("%f %d", f, e);
```

Função Relacionada

ldexp()

#include <math.h>
double ldexp(double num, int exp);

A função **ldexp()** devolve o valor de *num* * 2^{exp}. Se ocorre um estouro, HUGE_VAL é devolvido.

Exemplo

Este programa mostra o número 4.

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

void main(void)
{
  printf("%f", ldexp(1, 2));
}
```

Funções Relacionadas

frexp(), modf()

#include <math.h>
double log(double num);

A função **log()** devolve o logaritmo natural de *num*. Ocorrerá um erro de domínio se *num* for negativo e um erro de escala se o argumento for zero.

Exemplo

Este programa escreve os logaritmos naturais dos números de 1 a 10.


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

void main(void)
{
  double val=1.0;

  do {
    printf("%f %f\n", val, log(val));
    val++;
  } while(val<11.0);
}
```

Função Relacionada

log10()

#include <math.h>
double log10(double num);

A função **log10()** devolve o logaritmo de base 10 de *num*. Ocorrerá um erro de domínio se *num* for negativo e um erro de escala se o argumento for zero.

Exemplo

Este programa escreve os logaritmos de base 10 dos números de 1 a 10:

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

void main(void)
{
  double val=1.0;

  do {
    printf("%f %f\n", val, log10(val));
    val++;
  } while(val<11.0);
}
```

Função Relacionada

log()

#include <math.h>
double modf(double num, double *i);

A função **modf()** decompõe *num* nas suas partes inteira e fracionária. Ela devolve a parte fracionária e coloca a parte inteira na variável apontada por *i*.

Exemplo

Este fragmento de código imprime **10** e **0.123** na tela:

```
double i;
double f;

f = modf(10.123, &i);
printf("%f %f",i, f);
```

Funções Relacionadas

frexp(), ldexp()

#include <math.h>
double pow(double base, double exp);

A função **pow()** devolve *base* elevada à potência *exp* ($base^{exp}$). Ocorrerá um erro de domínio se *base* for zero e *exp* for menor ou igual a zero. Também ocorrerá um erro de domínio se *base* for negativa e *exp* não for um inteiro. Um estouro produzirá um erro de escala.

Exemplo

O programa seguinte imprime as dez primeiras potências de 10:

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

void main(void)
{
  double x = 10.0, y=0.0;

  do {
    printf("%f\n", pow(x, y));
    y++;
  } while(y<11.0);
}
```


Funções Relacionadas

exp(), log(), sqrt()

#include <math.h>
double sin(double arg);

A função **sin()** devolve o seno de *arg*. O valor de *arg* deve estar em radianos.

Exemplo

Este programa escreve os senos dos valores de -1 a 1 em incrementos de um décimo:

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

void main(void)
{
  double val=-1.0;

  do {
    printf("Seno de %f é %f.\n", val, sin(val));
    val += 0.1;
  } while(val<=1.0);
}
```

Funções Relacionadas

acos(), asin(), atan(), atan2(), cos(), cosh(), sinh(), tan(), tanh()

#include <math.h>
double sinh(double arg);

A função **sinh()** devolve o seno hiperbólico de *arg*. O valor de *arg* deve estar em radianos.

Exemplo

Este programa escreve os senos hiperbólicos dos valores -1 a 1 em incrementos de um décimo:

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

```
void main(void)
{
  double val=-1.0;

  do {
    printf("Seno hiperbólico de %f é %f.\n", val, sinh(val));
    val += 0.1;
  } while(val<=1.0);
}
```

Funções Relacionadas

acos(), asin(), atan(), atan2(), cos(), cosh(), sin(),tan(), tanh()

#include <math.h> double sqrt(double num);

A função **sqrt()** devolve a raiz quadrada de *num*. Se você chamar com um argumento negativo, ocorrerá um erro de domínio.

Exemplo

Este fragmento de código imprime 4 na tela:

```
printf("%f", sqrt(16.0));
```

Funções Relacionadas

exp(), log(), pow()

#include <math.h> double tan(double arg);

A função **tan()** devolve a tangente de *arg*.

Exemplo

Este programa escreve as tangentes dos valores de -1 a 1 em incrementos de um décimo:

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

void main(void)
{
```



```
  double val=-1.0;

  do {
    printf("Tangente de %f é %f.\n", val, tan(val));
    val += 0.1;
  } while(val<=1.0);
}
```

Funções Relacionadas

acos(), asin(), atan(), atan2(), cos(), cosh(), sin() sinh(), tanh()

#include <math.h> double tanh(double arg);

A função **tanh()** devolve a tangente hiperbólica de *arg*.

Exemplo

Este programa escreve as tangentes hiperbólicas dos valores de -1 a 1 em incrementos de um décimo:

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

void main(void)
{
  double val=-1.0;

  do {
    printf("Tangente hiperbólica de %f é %f.\n", val, tanh(val));
    val += 0.1;
  } while(val<=1.0);
}
```

Funções Relacionadas

acos(), asin(), atan(), atan2(), cos(), cosh(), sin(), sinh(), tan()

MAKRON Books

15

Funções de Hora, Data e Outras Relacionadas com o Sistema

Este capítulo aborda as funções que são mais sensíveis ao sistema operacional. As funções definidas pelo padrão C ANSI incluem as de hora e data bem como **setlocale()** e **localeconv()**. Essas funções utilizam as informações de hora e data do sistema operacional ou, no caso de **setlocale()**, suas informações políticas.

Este capítulo também discute uma categoria de funções que realiza uma interface direta com o sistema operacional. Nenhuma dessas funções é definida pelo padrão C ANSI, porque cada ambiente operacional é diferente. Este capítulo usa o DOS porque ele é o sistema operacional mais amplamente utilizado. Porém, mesmo os compiladores C que operam sob o DOS podem ter modelos ligeiramente diferentes para as funções de interface. Este capítulo usa as funções definidas pelos compiladores C/C++ da Microsoft para DOS, e assume que você sabe como as chamadas ao sistema DOS são acessadas. No entanto, você deve ser capaz de generalizar para as funções do seu próprio compilador.

Este capítulo discute, ainda, as funções definidas pelo Microsoft C, que realiza a interface com o ROM-BIOS do PC. O BIOS fornece o suporte, em nível mais baixo, para os vários dispositivos de hardware do computador. Em certo sentido, o BIOS é o nível mais baixo de qualquer sistema operacional para PC. As funções que acessam o BIOS não foram definidas pelo padrão C ANSI, mas são uma amostra representativa do tipo de funções fornecido por muitos compiladores C baseados em PC e em DOS.

O padrão C ANSI define diversas funções que manipulam a data e a hora do sistema como também o tempo transcorrido. Essas funções exigem o arquivo de cabeçalho TIME.H. Esse cabeçalho define quatro tipos: **size_t**, **clock_t**, **time_t** e **tm**. **size_t** é alguma variedade de inteiro sem sinal. Os tipos **clock_t** e

time_t podem representar o horário e a data do sistema como um inteiro longo. O padrão C ANSI refere-se a isso como *horário de calendário*. O tipo de estrutura **tm** contém a data e a hora decompostas em seus elementos. A estrutura **tm** é definida como:

```
struct tm {
  int tm_sec;    /* segundos, 0-59 */
  int tm_min;    /* minutos, 0-59 */
  int tm_hour;   /* horas, 0-23 */
  int tm_mday;   /* dia do mês, 1-31 */
  int tm_mon;    /* meses a partir de jan, 0-11 */
  int tm_year;   /* anos a partir de 1900 */
  int tm_wday;   /* dias a partir de domingo, 0-6 */
  int tm_yday;   /* dias a partir de 1 de jan, 0-365 */
  int tm_isdst;  /* Indicador de horário de verão, */
};
```

O valor de **tm_isdst** é positivo se estiver vigorando o horário de verão; zero se não estiver, e negativo se não há informação disponível. O padrão C ANSI refere-se a esse tipo de hora e data como *hora decomposta*.

Além disso, TIME.H define a macro **CLOCKS_PER_SEC**, que é o número de "tiques" do sistema por segundo.

As funções de localização exigem o arquivo de cabeçalho LOCALE.H.

As funções de interface com o DOS, definidas pelo Microsoft C++, exigem o cabeçalho DOS.H. Ele define uma união que corresponde aos registradores da família 8086 das CPUs e é utilizado por algumas funções de interface com o sistema. Essa união é definida como a união de duas estruturas, que permitem que cada registrador seja acessado por palavra ou byte. As estruturas e a união são mostradas aqui, como definidas pelo Microsoft:

```
/* Copyright (c) 1985-1992, Microsoft Corporation.
   All rights reserved.
*/

/* registradores de palavras */
struct_WORDREGS {
    unsigned int ax;
    unsigned int bx;
    unsigned int cx;
    unsigned int dx;
    unsigned int si;
    unsigned int di;
```

```
    unsigned int cflag;
};

/* registradores de bytes */
struct_BYTEREGS {
    unsigned char al, ah;
    unsigned char bl, bh;
    unsigned char cl, ch;
    unsigned char dl, dh;
};

/* união dos registradores de uso geral -
   sobrepõe os registradores de palavras e bytes correspondentes.
*/

union _REGS {
    struct_WORDREGS x;
    struct_BYTEREGS h;
};
```

DOS.H também define o tipo de estrutura **_SREGS**, utilizada por algumas funções para estabelecer os registradores de segmento. Essa estrutura é definida pelo Microsoft como:

```
/*  Copyright (c) 1985-1992, Microsoft Corporation.
    All rights reserved.
*/

/* registradores de segmento */

struct _SREGS {
    unsigned int es;
    unsigned int cs;
    unsigned int ss;
    unsigned int ds;
};
```

As funções de interface com o BIOS exigem o arquivo de cabeçalho BIOS.H.

Muitas das funções descritas neste capítulo atribuem à variável global inteira **errno** um valor de código de erro quando ocorre uma falha. (Você deve incluir ERRNO.H para acessar **errno**.) Consulte seu manual do usuário para detalhes.

#include <time.h>
char *asctime (const struct tm *ptr);

A função **asctime()** devolve um ponteiro para uma string que converte a informação armazenada na estrutura apontada por *ptr* à seguinte forma:

dia mês data horas:minutos:segundos ano\n\0

Por exemplo:

Wed Jun 19 12:05:34 1999

asctime() retorna um ponteiro para a string convertida.

O ponteiro de estrutura passado para **asctime()** é obtido geralmente de **localtime()** ou **gmtime()**.

O buffer usado por **asctime()** para guardar a string de saída formatada é uma matriz de caracteres alocada estaticamente e sobrescrita toda vez que a função é chamada. Para salvar o conteúdo da string, você deve copiá-la para outro lugar.

Exemplo

Este programa mostra o horário local definido pelo sistema:

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

void main(void)
{
  struct tm *ptr;
  time_t lt;

  lt = time(NULL);
  ptr = localtime(&lt);
  printf(asctime(ptr));
}
```

Funções Relacionadas

ctime(), gmtime(), localtime(), time()

#include <dos.h>
int _bdos(int fnum, unsigned dx, unsigned al);

A função _bdos() não é definida pelo padrão C ANSI e aplica-se apenas a compiladores C baseados em DOS, podendo aparecer com um nome ligeiramente diferente. Verifique seu manual do usuário.

A função **_bdos()** acessa a chamada do sistema DOS especificada por *fnum*. Primeiro, o valor *dx* é colocado no registrador **DX** e *al* no registrador **AL**; em seguida, é executada a instrução INT 21H.

A função **_bdos()** devolve o valor do registrador **AX**, que é usado pelo DOS para devolver informação.

A função **_bdos()** só pode ser usada para acessar chamadas ao sistema que não usam argumentos ou que necessitem apenas de **DX** e/ou **AL** como seus argumentos.

Exemplo

Este programa lê caracteres diretamente do teclado, contornando todas as funções de E/S de C, até que o usuário pressione ENTER:

```
/* Faz leitura do teclado. */
#include <dos.h>

void main(void)
{
  while((255 & _bdos(1, 0, 0)) != '\r');
}
```

Funções Relacionadas

intdos(), intdosx()

#include <bios.h>
unsigned_bios_disk(unsigned cmd, struct_diskinfo_t *info);

A função **_bios_disk()** não é definida pelo padrão C ANSI e aplica-se apenas a compiladores C baseados em DOS, podendo aparecer com um nome ligeiramente diferente. Verifique seu manual do usuário.

A função **_bios_disk()** realiza operações de disco em nível de BIOS usando a interrupção 0x13. Essas operações ignoram a estrutura lógica do disco, incluindo arquivos. Todas as operações ocorrem nos setores.

A Microsoft define a estrutura **_diskinfo_t** como:

```
struct _diskinfo_t {
    unsigned drive;     /* unidade acionadora de disco */
    unsigned head;      /* cabeça */
    unsigned track;     /* trilha */
    unsigned sector;    /* setor */
    unsigned nsectors;  /* números de setores */
    void _ _far *buffer;  /* buffer */
};
```

O acionador afetado é especificado em **drive**, com 0 correspondendo a A, 1 a B etc., para acionadores de disco flexível. O primeiro acionador de disco fixo é 0x80, o segundo 0x81 e assim por diante. A parte do disco operada é especificada em **head**, **track** e **sector**. O campo **nsectors** especifica o número de setores a ser lido ou escrito e **buffer** aponta para um buffer que armazena a informação lida ou escrita no disco. Consulte o *Manual Técnico de Referência* do IBM PC para detalhes na operação das rotinas de disco em nível de BIOS. Lembre-se de que o controle direto do disco exige um conhecimento íntimo e completo tanto do hardware como do DOS. Ele deve ser evitado, exceto em situações especiais.

Funções Relacionadas

fread(), fwrite()

#include <bios.h>
unsigned _bios_equiplist(void);

A função **_bios_equiplist()** não é definida pelo padrão C ANSI e aplica-se apenas a compiladores C baseados em DOS, podendo aparecer com um nome ligeiramente diferente. Verifique seu manual do usuário.

A função **_bios_equiplist()** devolve um valor que especifica quais equipamentos se encontram no computador. Esse valor é codificado como mostrado a seguir:

Bit	Equipamento
0	Boot realizado por meio do disco flexível
1	Co-processador 80x87 instalado, se zero
2,3	Tamanho da RAM na *motherboard* (placa-mãe)
	0 0: 16k
	0 1: 32k

	1 0: 48k
	1 1: 64k
4,5	Modo inicial de vídeo
	0 0: não usado
	0 1: 40x25 BW, adaptador colorido
	1 0: 80x25 BW, adaptador colorido
	1 1: 80x25, adaptador monocromático
6,7	Números de acionadores de disco flexível
	0 0: um
	0 1: dois
	1 0: três
	1 1: quatro
8	Chip de DMA instalado
9,10,11	Número de portas seriais
	0 0 0: zero
	0 0 1: um
	0 1 0: dois
	0 1 1: três
	1 0 0: quatro
	1 0 1: cinco
	1 1 0: seis
	1 1 1: sete
12	Adaptador de jogos instalado
13	Modem instalado
14,15	Número de impressoras
	0 0: zero
	0 1: uma
	1 0: duas
	1 1: três

Exemplo

Este programa mostra o número de acionadores de disco flexível instalado no computador:

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

void main(void)
{
  unsigned eq;

  eq = _bios_equiplist();
```



```
  eq >>= 6; /* desloca os bits 6 e 7 para a posição mais baixa */

  printf("Número de acionadores de disco: %d", (eq & 3) + 1);
}
```

Função Relacionada

_bios_serialcom()

#include <bios.h>
unsigned _bios_keybrd(unsigned cmd);

A função **_bios_keybrd()** não é definida pelo padrão C ANSI e aplica-se apenas a compiladores C baseados em DOS, podendo aparecer com um nome ligeiramente diferente. Verifique seu manual do usuário.

A função **_bios_keybrd()** executa operações diretamente no teclado. O valor de *cmd* determina qual operação será executada.

Se *cmd* é zero, **_bios_keybrd()** devolve a próxima tecla pressionada. (Ela espera até que uma tecla seja pressionada.) A função devolve uma quantidade de 16 bits, que consiste em dois valores diferentes. O byte menos significativo conterá o código ASCII do caractere se uma tecla normal for pressionada; contém zero se uma tecla especial for pressionada. As teclas especiais incluem as teclas de movimentação de cursor e de função. O byte menos significativo contém o código de varredura da tecla, que corresponde imprecisamente à posição da tecla no teclado.

Se *cmd* é 1, **_bios_keybrd()** verifica se uma tecla foi pressionada. Em caso afirmativo, ela devolve um valor diferente de zero; caso contrário, devolve zero.

Quando *cmd* vale 2, é devolvido o estado de maiúsculas. O estado de diversas teclas que alteram o significado de outras é codificado na porção menos significativa do valor devolvido, como mostrado aqui:

Bit	Significado
0	Tecla SHIFT direita pressionada
1	Tecla SHIFT esquerda pressionada
2	Tecla CTRL pressionada
3	Qualquer tecla ALT pressionada
4	Tecla SCROLL LOCK ligada
5	Tecla NUM LOCK ligada
6	Tecla CAPS LOCK ligada

7 Tecla INSERT ligada
8 Tecla CTRL esquerda pressionada
9 Tecla ALT esquerda pressionada
10 Tecla CTRL direita pressionada
11 Tecla ALT direita pressionada
12 Tecla SCROLL LOCK pressionada
13 Tecla NUM LOCK pressionada
14 Tecla CAPS LOCK pressionada
15 Tecla SYSRQ pressionada

Exemplo

Este fragmento de código gera números randômicos (aleatórios) até que uma tecla seja pressionada. Isso permite inicializar **rand()**, o gerador de números randômicos, a partir de um ponto randômico.

```
while(!_bios_keybrd(1)) rand();
```

Funções Relacionadas

getche(), kbhit()

#include <bios.h>
unsigned _bios_memsize(void);

A função **_bios_memsize()** não é definida pelo padrão C ANSI e aplica-se apenas a compiladores C baseados em DOS, podendo aparecer com um nome ligeiramente diferente. Verifique seu manual do usuário.

A função **_bios_memsize()** devolve a quantidade de memória (em unidades de 1K) instalada no sistema. (O valor retomado nunca é maior que 640K.)

Exemplo

Este programa apresenta a quantidade de memória do sistema.

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

void main(void)
{
  printf("%uK bytes de ram", _bios_memsize());
}
```


Função Relacionada

_bios_equiplist()

#include <bios.h>
unsigned _bios_printer(unsigned cmd, unsigned port, unsigned data);

A função **_bios_printer()** não é definida pelo padrão C ANSI e aplica-se apenas a compiladores C baseados em DOS, podendo aparecer com um nome ligeiramente diferente. Verifique seu manual do usuário.

A função **_bios_printer()** controla a porta da impressora especificada em *port*. Se *port* é zero, LPT1 é usado; se *port* é 1, LPT2 é acessado. A função exata que é executada depende do valor de *cmd*. Os valores legais para *cmd* são mostrados aqui:

Valor	Significado
0	Imprime o caractere passado em *data*
1	Inicializa a porta de impressora
2	Devolve o estado da porta

O estado da porta de impressora é codificado no byte menos significativo do valor devolvido, conforme mostrado aqui:

Bit	Significado
0	Erro de *time-out* (temporização)
1	Não usado
2	Não usado
3	Erro de E/S
4	Impressora selecionada
5	Erro de falta de papel
6	Byte recebido
7	Impressora NÃO ocupada (pronta)

Exemplo

Este fragmento escreve a string **ola** na impressora conectada a LPT1:

```
char p[] = "ola"
while(*p) _bios_printer(0, 0, *p++);
```

Função Relacionada

_bios_serialcom()

#include <bios.h>
unsigned _bios_serialcom(unsigned cmd, port, unsigned data);

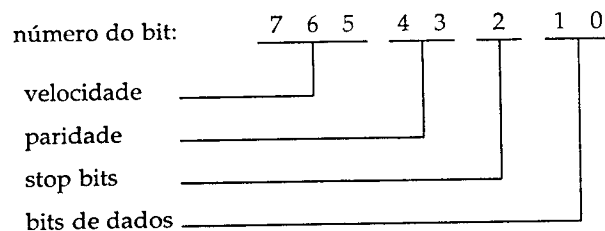
A função **_bios_serialcom()** não é definida pelo padrão C ANSI e aplica-se apenas a compiladores C baseados em DOS, podendo aparecer com um nome ligeiramente diferente. Verifique seu manual do usuário.

A função **_bios_serialcom()** manipula a porta de comunicação assíncrona RS232 especificada em *port*. A sua operação é determinada pelo *cmd*, cujos valores são mostrados a seguir:

cmd	Significado
0	Inicializa a porta
1	Envia um caractere
2	Recebe um caractere
3	Devolve o estado da porta

Para acessar COM1, *port* deve ser zero. Para acessar COM2, *port* deve ser 1.

Antes de utilizar a porta serial, talvez seja necessário inicializá-la com atribuições diferentes do padrão. Para fazer isso, chame **_bios_serialcom()** com *cmd* igual a 0. A forma como a porta é colocada é determinada pelo valor do byte menos significativo de *data*, que é codificado com parâmetros de inicialização, conforme mostrado a seguir:



A velocidade (em *bauds*) é codificada da seguinte forma:

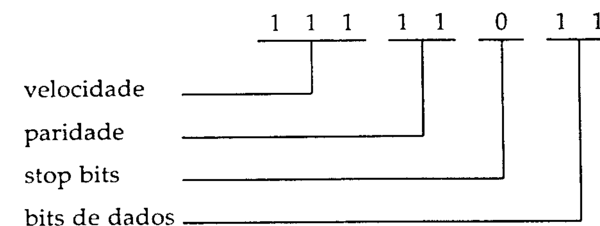
Velocidade	Padrão dos bits
9600	111
4800	110
2400	101
1200	100
600	011
300	010
150	001
110	000

Os bits de paridade são codificados como mostrado aqui:

Paridade	Padrão dos bits
Sem paridade	00 ou 10
Ímpar	01
Par	11

O número de stop bits é determinado pelo bit 2 do byte de inicialização da porta serial. Se for 1, dois stop bits serão utilizados; caso contrário, apenas um stop bit será utilizado. Finalmente, o número de bits de dados é estabelecido pelo código nos bits 0 e 1 do byte de inicialização. Dos quatro padrões de bits possíveis, apenas dois são válidos. Se os bits 1 e 0 contêm o padrão 1 0, são utilizados sete bits de dados. Se contêm 1 1, são utilizados oito bits de dados.

Por exemplo, para estabelecer a porta em 9600 bauds, paridade par, um stop bit e oito bits de dados, seria utilizado o seguinte padrão de bits:



Em decimal, isso fica 251.

O valor devolvido por **_bios_serialcom()** é sempre uma quantidade de 16 bits. O byte mais significativo contém os bits de estado, que recebem estes valores:

Significado quando ligado	Bit
Dados prontos	0
Erro de excesso de caracteres	1
Erro de paridade	2
Erro de enquadramento	3
Erro de detecção de interrupção	4
Registro de armazenamento de envio vazio	5
Registro de deslocamento de envio vazio	6
Erro de *time-out* (temporização)	7

Se *cmd* é colocado em 0, 1 ou 3, o byte menos significativo é codificado conforme mostrado a seguir:

Significado quando ligado	Bit
Alteração em clear-to-send	0
Alteração em data-set-ready	1
Detector de chamada na borda de descida	2
Alteração no sinal de linha	3
Clear-to-send	4
Data-set-ready	5
Indicador de chamada	6
Sinal de linha detectado	7

Quando *cmd* tem o valor 2, o byte menos significativo contém o valor recebido pela porta.

Exemplo

Isso inicializa a porta 0 com 9600 bauds, paridade par, um stop bit e oito bits de dados:

```
_bios_serialcom(0, 0, 251);
```

Função Relacionada

bioskey()

#include <bios.h>
unsigned _bios_timeofday(unsigned cmd, long *newtime);

A função **_bios_timeofday()** não é definida pelo padrão C ANSI e aplica-se apenas a compiladores C baseados em DOS, podendo aparecer com um nome ligeiramente diferente. Verifique seu manual do usuário.

A função **_bios_timeofday()** lê ou ajusta o relógio do sistema. O relógio do sistema "bate" a uma razão de 18,2 vezes por segundo. Seu valor é zero à meia-noite e aumenta até ser zerado novamente à meia-noite ou até ser colocado manualmente em algum valor. Se *cmd* é zero, **_bios_timeofday()** devolve o valor atual do relógio na variável ajustada para *newtime*. Se *cmd* é 1, o relógio é ajustado para o valor de *newtime*. A função retorna o valor de registro AX como definido pela rotina da BIOS

Exemplo

Este programa escreve o valor atual do relógio:

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

void main(void)
{
  long t;

  _bios_timeofday(0, &t);
  printf("Valor do relógio: %ld", t);
}
```

Funções Relacionadas

ctime(), time()

#include <time.h>
clock_t clock(void);

A função **clock()** devolve um valor aproximado do tempo de execução do programa que a chama. Para transformar esse valor em segundos, divida-o por CLOCKS_PER_SEC. Um valor devolvido de -1 indica que o tempo não está disponível.

Exemplo

A função seguinte mostra o tempo de execução, em segundos, para o programa que a chama:

```
void elapsed_time(void)
{
  printf("Tempo transcorrido: %u segs.\n", clock()/CLOCK_PER_SEC);
}
```

Funções Relacionadas

asctime(), ctime(), time()

#include <time.h>
char *ctime(const time_t *time);

Dado um ponteiro para o horário de calendário, a função **ctime()** devolve um ponteiro para uma string na forma:

dia mês ano horas:minutos:segundos ano\n\0

Eis um exemplo dessa string: Mon Dec 5 12:03:03 1996. O horário de calendário normalmente é obtido por meio de uma chamada a **time()**.

O buffer usado por **ctime()** para guardar a string de saída formatada é uma matriz de caracteres alocada estaticamente e sobrescrita toda vez que a função é chamada. Para salvar o conteúdo da string, você precisa copiá-lo em outro lugar.

Exemplo

Este programa mostra a hora local definida pelo sistema:

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

void main(void)
{
  time_t lt;

  lt = time(NULL);
  printf(ctime(&lt));
}
```

Funções Relacionadas

asctime(), gmtime(), localtime(), time()

#include <time.h>
double difftime(time_t time2, time_t time1);

A função **difftime()** devolve a diferença, em segundos, entre *time1* e *time2*. Ou seja, ela devolve *time2 - time1*.

Exemplo

Este programa mede o tempo em segundos que o laço **for** vazio gasta para ir de 0 a 500.000:

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

void main(void)
{
  time_t start, end;
  long unsigned t;

  start = time(NULL);
  for(t=0; t<500000; t++);
  end = time(NULL);
  printf("O laço usou %f segundos.\n", difftime(end, start));
}
```

Funções Relacionadas

asctime(), gmtime(), localtime(), time()

#include <dos.h>
void _disable(void);

A função **_disable()** não é definida pelo padrão C ANSI e aplica-se apenas a compiladores C baseados em DOS, podendo aparecer com um nome ligeiramente diferente. Verifique seu manual do usuário.

A função **_disable()** desabilita as interrupções. A única interrupção que continua habilitada é a NMI (interrupção não-mascarável). Empregue esta função com cuidado porque muitos dispositivos do sistema usam interrupções.

Função Relacionada

_enable()

#include <dos.h>
unsigned _dos_allocmem(unsigned size, unsigned *seg);

A função **_dos_allocmem()** não é definida pelo padrão C ANSI e aplica-se apenas a compiladores C baseados em DOS, podendo aparecer com um nome ligeiramente diferente. Verifique seu manual do usuário.

A função **_dos_allocmem()** aloca um bloco de memória alinhado por parágrafo. Ela põe o endereço do segmento do bloco no inteiro sem sinal apontado por *seg*. O argumento *size* especifica o número de parágrafos a serem alocados (um parágrafo tem 16 bytes).

Se a memória requisitada pode ser alocada, é devolvido zero. Se não existe memória suficiente, não é feita nenhuma atribuição ao inteiro sem sinal apontado por *seg* e um valor diferente de zero é devolvido. Em caso de erro, **errno** recebe **ENOMEM** (memória insuficiente).

Exemplo

Este fragmento de código aloca 100 parágrafos de memória:

```
unsigned i;

i = 0;

if((_dos_allocmem(100, &i))==0)
    printf("Alocação bem-sucedida\n");
else
    printf("Falha na alocação\n");
```

Funções Relacionadas

_dos_freemem(), _dos_setblock()

#include <dos.h>
unsigned _dos_close(int fd);

A função **_dos_close()** não é definida pelo padrão C ANSI e aplica-se apenas a compiladores C baseados em DOS, podendo aparecer com um nome ligeiramente diferente. Verifique seu manual do usuário.

A função **_dos_close()** fecha o arquivo especificado pelo descritor de arquivo *fd*. Ela é funcionalmente equivalente à função tipo UNIX **close()**. Recorde que os descritores de arquivo são utilizados pelo sistema de arquivo tipo UNIX e não têm relação com o sistema de arquivo C ANSI. A função devolve zero se bem-sucedida. Caso contrário, ela devolve um valor diferente de zero e **errno** receberá **EBADF** (descritor de arquivo ruim).

Exemplo

Este fragmento fecha o arquivo associado ao descritor de arquivo **fd**:

```
_dos_close(fd);
```


Funções Relacionadas

_dos_creat(), _dos_open()

#include <dos.h>
unsigned _dos_creat(char *fname, unsigned attr, int *fd);
unsigned _dos_creatnew(char *fname, unsigned attr, int *fd);

As funções **_dos_creat()** e **_dos_creatnew()** não são definidas pelo padrão C ANSI e aplicam-se apenas a compiladores C baseados em DOS, podendo aparecer com nomes ligeiramente diferentes. Verifique seu manual do usuário.

A função **_dos_creat()** cria um arquivo cujo nome é apontado por *fname* com os atributos especificados por *attr*. Ela devolve um descritor de arquivo no inteiro apontado por *fd*. (Descritores de arquivo são usados pelo sistema de arquivo tipo UNIX, não pelo sistema de arquivo C ANSI.) Se o arquivo já existe, é apagado. A função **_dos_creatnew()** é igual a **_dos_creat()**, exceto que, se o arquivo já existe, ele não é apagado e **_dos_creatnew()** devolve um erro.

Os valores válidos para *attr* são mostrados aqui (as macros são definidas em DOS.H):

Macro	**Significado**
_A_NORMAL	Arquivo normal
_A_RDONLY	Arquivo de apenas leitura
_A_HIDDEN	Arquivo oculto
_A_SYSTEM	Arquivo de sistema
_A_VOLID	Rótulo de volume
_A_SUBDIR	Subdiretório
_A_ARCH	Bit de arquivo alterado

As duas funções devolvem zero quando bem-sucedidas e um valor diferente de zero em caso de falha. Em caso de falha, **errno** contém um destes valores: **ENOENT** (arquivo não encontrado), **EMFILE** (muitos arquivos abertos), **EACCES** (acesso negado) ou **EEXIST** (arquivo já existe).

As funções **_dos_creat()** e **_dos_creatnew()** são semelhantes à função tipo UNIX **creat()**.

Exemplo

Este fragmento de código abre um arquivo chamado TEST.TST para saída:

```
int fd;

if(_dos_creat("test.tst",_A_NORMAL, &fd))
  printf("O arquivo não pode ser aberto.");
```

Função Relacionada

_dos_open()

#include <dos.h>
int _dosexterr(struct_DOSERROR *err);

A função **_dosexterr()** não é definida pelo padrão C ANSI e aplica-se apenas a compiladores C baseados em DOS, podendo aparecer com um nome ligeiramente diferente. Verifique seu manual do usuário.

A função **_dosexterr()** preenche a estrutura apontada por *err* com informações estendidas de erro quando uma chamada ao DOS falha. A estrutura **_DOSERROR** é definida desta forma:

```
struct _DOSERROR {
  int exterror; /* código do erro */
  char class;   /* classe do erro */
  char action;  /* ação sugerida */
  char locus;   /* local do erro */
};
```

Para uma interpretação apropriada da informação devolvida pelo DOS, veja a referência técnica do DOS.

Função Relacionada

ferror()

#include <dos.h>
unsigned _dos _findfirst(char *fname, unsigned attr, struct _find_t *ptr);
unsigned _dos_findnext(struct _find_t *ptr);

As funções **_dos_findfirst()** e **_dos_findnext()** não são definidas pelo padrão C ANSI e aplicam-se apenas a compiladores C baseados em DOS, podendo aparecer com nomes ligeiramente diferentes. Verifique seu manual do usuário.

A função **_dos_findfirst()** busca pelo primeiro nome de arquivo que coincide com aquele apontado por *fname*. O nome de arquivo pode incluir um especificador do acionador e um nome de caminho (*path*). Além disso, o nome do arquivo pode incluir os caracteres-chave * e ?. Caso seja encontrada uma coincidência, a estrutura apontada por *ptr* é preenchida com informações sobre o arquivo.

O Microsoft define a estrutura **_find_t** como segue:

```
struct _find_t {
  char reserved[21]; /* para uso do DOS */
  char attrib;       /* atributo do arquivo */
  unsigned wr_time;  /* horário da última alteração no arquivo */
  unsigned wr_date;  /* dia da última alteração no arquivo */
  long size;         /* tamanho em bytes */
  char name[13];     /* nome do arquivo */
};
```

O parâmetro *attrib* determina que tipo de arquivos será pesquisado por **_dos_findfirst()**. *attrib* pode ser uma das seguintes macros (definidas em DOS.H):

Macro	Significado
_A_NORMAL	Arquivo normal
_A_RDONLY	Arquivo de apenas leitura
_A_HIDDEN	Arquivo oculto
_A_SYSTEM	Arquivo de sistema
_A_VOLID	Rótulo de volume
_A_SUBDIR	Subdiretório
_A_ARCH	Bit de arquivo alterado

A função **_dos_findnext()** continua a busca iniciada por **_dos_findfirst()**. O buffer apontado por *ptr* deve ser o mesmo utilizado na chamada a **_dos_findfirst()**.

As funções **_dos_findfirst()** e **_dos_findnext()** devolvem zero em caso de sucesso e um valor diferente de zero em caso de falha ou quando não é mais encontrada nenhuma coincidência. Em caso de falha, **errno** recebe **ENOENT** (nome de arquivo não encontrado).

Exemplo

Este programa mostra todos os arquivos com extensão .C e seus respectivos tamanhos no diretório corrente:

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

void main(void)
{
  struct _find_t f;
  register int done;

  done = _dos_findfirst("*.c", _A_NORMAL, &f);
  while(!done) {
    printf("%s %ld\n", f.name, f.size);
    done = _dos_findnext(&f);
  }
}
```

#include <dos.h> unsigned _dos_freemem(unsigned seg);

A função **_dos_freemem()** não é definida pelo padrão C ANSI e aplica-se apenas a compiladores C baseados em DOS, podendo aparecer com um nome ligeiramente diferente. Verifique seu manual do usuário.

A função **_dos_freemem()** libera o bloco de memória cujo segmento está em *seg*. Essa memória deve ter sido alocada anteriormente usando **_dos_ allocmem()**. A função devolve zero em caso de sucesso; em caso de falha, ela devolve um valor diferente de zero e ajusta **errno** para ENOMEM (memória insuficiente).

Exemplo

Este fragmento de código ilustra como alocar e liberar memória usando **_dos_allocmem()** e **_dos_freemem()**.

```
unsigned i;

if(_dos_allocmem(some, &i)!=0)
  printf("Erro de alocação");
else
  _dos_freemem(i);
```

Funções Relacionadas

_dos_allocmem(), _dos_setblock()

#include <dos.h> void _dos_getdate(struct _dosdate_t *d); void _dos_gettime(struct _dostime_t *t);

As funções **_dos_getdate()** e **_dos_gettime()** não são definidas pelo padrão C ANSI e aplicam-se apenas a compiladores C baseados em DOS, podendo aparecer com nomes ligeiramente diferentes. Verifique seu manual do usuário.

A função **_dos_getdate()** preenche a estrutura apontada por *d* com o formato do DOS para a data atual do sistema. A função **_dos_gettime()** preenche a estrutura apontada por *t* com o formato do DOS para o horário atual do sistema.

A Microsoft define a estrutura **_dosdate_t** como:

```
struct _dosdate_t {
  unsigned char day;       /* dia */
  unsigned char month;     /* mês */
  unsigned char ano;       /* ano */
  unsigned char dayofweek; /* dia da semana, domingo é zero */
};
```

O Microsoft define a estrutura **_dostime_t** conforme mostrado aqui:

```
struct _dostime_t {
  unsigned char hour;    /* hora */
  unsigned char minute;  /* minuto */
  unsigned char second;  /* segundo */
  unsigned char hsecond; /* centésimos de segundo */
};
```

Exemplo

Este programa mostra a data e a hora usando chamadas ao sistema:

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

void main(void)
{
  struct _dostime_t t;
  struct _dosdate_t d;

  _dos_getdate(&d);
```

```
  _dos_gettime(&t);

  printf("data: %d/%d/%d\n", d.day, d.month, d.year);
  printf("hora: %d:%d:%d\n", t.hour, t.minute, t.second);
}
```

Funções Relacionadas

_dos_setdate(), _dos_settime()

#include <dos.h>
unsigned _dos_getdiskfree(unsigned drive, struct _diskfree_t *dfptr);

A função **_dos_getdiskfree()** não é definida pelo padrão C ANSI e aplica-se apenas a compiladores C baseados em DOS, podendo aparecer com um nome ligeiramente diferente. Verifique seu manual do usuário.

A função **_dos_getdiskfree()** devolve o total de espaço livre no disco na estrutura apontada por *dfptr* para o acionador especificado por *drive*. Os acionadores são numerados a partir de 1, começando por A. Você pode especificar o acionador padrão chamando **_dos_getdiskfree()** com o valor zero. A estrutura **_diskfree_t** é definida pelo Microsoft como segue:

```
struct _diskfree_t {
  unsigned total_clusters;      /* total de clusters no disco */
  unsigned avail_clusters;      /* clusters disponíveis */
  unsigned sectors_per_cluster; /* setores por cluster */
  unsigned bytes_per_sector;    /* bytes por setor */
};
```

A função retornará zero no caso de sucesso e um valor diferente de zero se ocorrer um erro. No caso de falha, **errno** recebe o valor **EINVAL** (unidade incorreta).

Exemplo

O programa seguinte escreve o número de clusters disponíveis para uso no acionador C:

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

void main(void)
{
  struct _diskfree_t p;

  _dos_getdiskfree(3, &p); /* drive C */

  printf("Número de clusters livres é %u.", p.avail_clusters);
}
```


Função Relacionada

_dos_getftime()

#include <dos.h>
void _dos_getdrive(unsigned *drive);

A função **_dos_getdrive()** não é definida pelo padrão C ANSI e aplica-se apenas a compiladores C baseados em DOS, podendo aparecer com um nome ligeiramente diferente. Verifique seu manual do usuário.

A função **_dos_getdrive()** devolve o número do acionador de disco atual no inteiro apontado por *drive*. O acionador A é codificado como 1, o acionador B, como 2, e assim por diante.

Exemplo

Este fragmento de código mostra o acionador de disco atual.

```
unsigned d;

_dos_getdrive(&d);
printf("drive é %c", d+'A'-1);
```

Função Relacionada

_dos_setdrive()

#include <dos.h>
unsigned _dos_getfileattr(const char *fname, unsigned *attrib);

A função **_dos_getfileattr()** não é definida pelo padrão C ANSI e aplica-se apenas a compiladores C baseados em DOS, podendo aparecer com um nome ligeiramente diferente. Verifique seu manual do usuário.

A função **_dos_getfileattr()** devolve o atributo do arquivo especificado por *fname* no inteiro sem sinal apontado por *attrib*, que pode ser um dos seguintes valores (as macros são definidas pelo Microsoft em DOS.H.):

Macro	**Significado**
_A_NORMAL	Arquivo normal
_A_RDONLY	Arquivo de apenas leitura
_A_HIDDEN	Arquivo oculto
_A_SYSTEM	Arquivo de sistema
_A_VOLID	Rótulo de volume
_A_SUBDIR	Subdiretório
_A_ARCH	Bit de arquivo alterado

A função **_dos_getfileattr()** devolve zero se bem-sucedida; caso contrário, um valor diferente de zero é devolvido. Se ocorre uma falha, **errno** recebe **ENOENT** (arquivo inválido).

Exemplo

Este fragmento de código determina se TEST.TST é um arquivo normal:

```
unsigned attr;
if(_dos_getfileattr("test.tst", &attr))
  printf("Erro de arquivo");

if(attr & _A_NORMAL) printf("O arquivo é normal.");
```

Função Relacionada

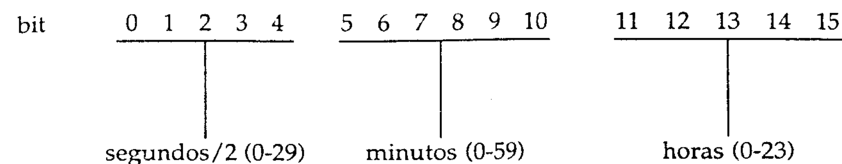
_dos_setfileattr()

#include <dos.h>
unsigned _dos_getftime(int fd, unsigned *fdate, unsigned *ftime);

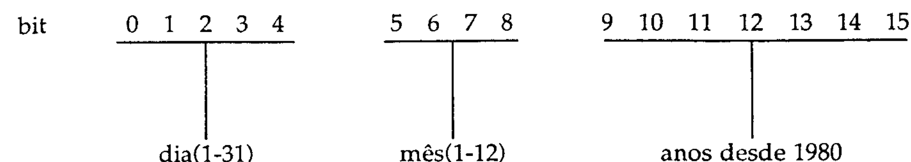
A função **_dos_getftime()** não é definida pelo padrão C ANSI e aplica-se apenas a compiladores C baseados em DOS, podendo aparecer com um nome ligeiramente diferente. Verifique seu manual do usuário.

A função **_dos_getftime()** devolve a hora e a data de criação do arquivo associado ao descritor de arquivo *fd* nos inteiros apontados por *ftime* e *fdate*.

Os bits no objeto apontado por *ftime* são codificados conforme mostrado a seguir:



Os bits no objeto apontado por *fdate* são codificados como segue:



Como indicado, o ano é representado como o número de anos desde 1980. Assim, se o ano é 2000, o valor dos bits de 9 a 15 é 20.

A função **_dos_getftime()** devolve zero caso seja bem-sucedida. Se ocorre um erro, ela devolve um valor diferente de zero e ajusta **errno** para que seja igual a **EBADF** (número de arquivo ruim).

Lembre-se de que os arquivos associados a descritores de arquivo utilizam o sistema de E/S tipo UNIX, que não é definido ou relacionado com o sistema de arquivo C ANSI.

Exemplo

Este programa escreve o ano em que o arquivo TEST.TST foi criado.

```
#include <io.h>
#include <dos.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>

void main(void)
{
  struct {
    unsigned day: 5;
    unsigned month: 4;
    unsigned year: 7;
  } d;
```

```
  unsigned t;
  int fd;

  if((fd=open("TEST.TST", O_RDONLY))==-1) {
    printf("O arquivo não pode ser aberto.");
    exit(1);
  }

  _dos_getftime(fd, (unsigned *) &d, &t);

  printf("Data de criação: %u", d.year+1980);

}
```

Função Relacionada

_dos_setftime()

#include <dos.h>
void (_ _interrupt _ _far *_dos_getvect(unsigned intr))(void);

A função **_dos_getvect()** não é definida pelo padrão C ANSI e aplica-se apenas a compiladores C baseados em DOS, podendo aparecer com um nome ligeiramente diferente. Verifique seu manual do usuário.

A função **_dos_getvect()** devolve o endereço da rotina do serviço de interrupção especificado em *intr*. Esse valor é devolvido como um ponteiro far.

Exemplo

O fragmento de código seguinte devolve o endereço da função de impressão na tela (que é associado à interrupção 5).

```
void (_ _interrupt _ _far *p)(void);

p = _dos_getvect(5);
```

Função Relacionada

_dos_setvect()

#include <dos.h>
void _dos_keep(unsigned status, unsigned size);

A função **_dos_keep()** não é definida pelo padrão C ANSI e aplica-se apenas a compiladores C baseados em DOS, podendo aparecer com um nome ligeiramente diferente. Verifique seu manual do usuário.

A função **_dos_keep()** executa uma interrupção 0x31, que faz com que o programa atual termine, mas permaneça residente. O valor de *status* é devolvido ao DOS como um código de retorno. O tamanho do programa que deve ficar residente é especificado em *size*. O tamanho é especificado em parágrafos (16 bytes). O restante da memória é liberada para uso pelo DOS.

Pelo fato de os programas que terminam e permanecem residentes serem um tanto complexos, não há nenhum exemplo aqui. No entanto, meu livro *The Craft of C* (Calif.: Osborne/McGraw-Hill, 1992), aborda este importante tópico em profundidade.

#include <dos.h>
unsigned _dos_open(const char *fname, unsigned mode, int *fd);

A função **_dos_open()** não é definida pelo padrão C ANSI e aplica-se apenas a compiladores C baseados em DOS, podendo aparecer com um nome ligeiramente diferente. Verifique seu manual do usuário.

A função **_dos_open()** abre o arquivo cujo nome é apontado por *fname*, no modo especificado por *mode*, e devolve um descritor de arquivo no inteiro apontado por *fd*. Lembre-se de que descritores de arquivo são utilizados pelo sistema de arquivo tipo UNIX e não têm relação com o sistema de arquivo C ANSI.

Os valores mais comuns para *mode* são:

Valor	Significado
O_RDONLY	Apenas leitura
O_WRONLY	Apenas escrita
O_RDWR	Leitura/escrita

Essas macros estão definidas em FCNTL.H

A função **_dos_open()** devolve zero caso seja bem-sucedida e um valor diferente de zero em caso de falha. Se ocorre um erro, **errno** recebe **EINVAL** (modo de acesso inválido), **EACCES** (acesso negado), **EMFILE** (arquivos abertos em excesso) ou **ENOENT** (arquivo não encontrado).

Exemplo

O fragmento de código a seguir abre um arquivo chamado TEST.TST para operações de leitura/escrita.

```
int fd;

if(_dos_open("test.tst", O_RDWR, &fd))
  printf("Erro ao abrir Arquivo.");
```

Funções Relacionadas

_dos_close(), _dos_creat(), _dos_creatnew()

#include <dos.h>
unsigned _dos_read(int fd, void __far *buf, unsigned count, unsigned *numread);

A função **_dos_read()** não é definida pelo padrão C ANSI e aplica-se apenas a compiladores C baseados em DOS, podendo aparecer com um nome ligeiramente diferente. Verifique seu manual do usuário.

A função **_dos_read()** lê *count* bytes do arquivo especificado pelo descritor de arquivo *fd* para o buffer apontado por *buf*. O número de bytes realmente lidos é devolvido em *numread*.

Em caso de sucesso, **_dos_read()** devolve zero; em caso de falha, devolve um valor diferente de zero. O valor devolvido é determinado pelo DOS. Você precisará da documentação técnica do DOS para determinar a natureza de quaisquer erros que ocorram. Além disso, no caso de falha, **errno** recebe ou **EACCES** (acesso negado) ou **EBADF** (arquivo não existe).

Exemplo

Este fragmento lê 128 caracteres do arquivo descrito por fd:

```
unsigned count;
char *buf[128];

if(_dos_read(fd, buf, 128, &count))
  printf("Erro na leitura do arquivo.");
```


Função Relacionada

_dos_write()

#include <dos.h>
unsigned _dos_setblock(unsigned size, unsigned seg, unsigned *max);

A função **_dos_setblock()** não é definida pelo padrão C ANSI e aplica-se apenas a compiladores C baseados em DOS, podendo aparecer com um nome ligeiramente diferente. Verifique seu manual do usuário.

A função **_dos_setblock()** altera o tamanho do bloco de memória cujo endereço de segmento é *seg*. O novo tamanho (*size*) é especificado em parágrafos (16 bytes). O bloco de memória deve ter sido previamente alocado com **_dos_allocmem()**.

Se o ajuste no tamanho não pode ser feito, **_dos_setblock()** devolve o maior bloco (em parágrafos) que pode ser alocado no objeto apontado por *max*.

Em caso de sucesso, **_dos_setblock()** devolve zero; em caso de falha, é devolvido um valor diferente de zero e **errno** recebe **ENOMEM** (memória insuficiente).

Exemplo

Este fragmento de código tenta redimensionar para 100 parágrafos o bloco de memória cujo endereço de segmento está em **seg**:

```
unsigned max;

if(_dos_setblock(seg, 100, &max)!=0)
  printf("Erro de redimensionamento, maior bloco é %u.", max);
```

Funções Relacionadas

_dos_allocmem(), _dos_freemem()

#include <dos.h>
unsigned _dos_setdate(struct _dosdate_t *d);
unsigned _dos_settime(struct _dostime_t *t);

As funções **_dos_setdate()** e **_dos_settime()** não são definidas pelo padrão C ANSI e aplicam-se apenas a compiladores C baseados em DOS, podendo aparecer com nomes ligeiramente diferentes. Verifique seu manual do usuário.

A função **_dos_setdate()** define a data do sistema como sendo a especificada na estrutura apontada por *d*. A função **_dos_settime()** estabelece a hora do sistema conforme especificada pela estrutura apontada por *t*.

A Microsoft define a estrutura **_dosdate_t** na seguinte listagem:

```
struct _dosdate_t {
  unsigned char day;        /* dia */
  unsigned char month;      /* mês */
  unsigned int year;        /* ano */
  unsigned char dayofweek; /* dia da semana, domingo é zero */
};
```

A Microsoft define a estrutura **_dostime_t** conforme mostrado aqui:

```
struct _dostime_t {
  unsigned char hour;           /* hora */
  unsigned char minute;         /* minuto */
  unsigned char second;         /* segundo */
  unsigned char hsecond;        /* centésimos de segundo */
};
```

As funções retornarão zero em caso de sucesso e um valor diferente de zero se ocorrer um erro.

Exemplo

Este código ajusta a hora do sistema para 10:10:10.0:

```
struct dostime_t t;

t.hour = 10;
t.minute = 10;
t.second = 10;
t.hsecond = 0;

_dos_settime(&t);
```


Funções Relacionadas

_dos_getdate(),_dos_gettime()

#include <dos.h>
void _dos_setdrive(unsigned drive, unsigned *num);

A função **_dos_setdrive()** não é definida pelo padrão ANSI e aplica-se apenas a compiladores C baseados em DOS, podendo aparecer com um nome ligeiramente diferente. Verifique seu manual do usuário.

A função **_dos_setdrive()** modifica o acionador de disco atual para aquele especificado por *drive*. O acionador A corresponde a 1, o acionador B, a 2 e assim por diante. O número de acionadores do sistema é devolvido no inteiro apontado por *num*.

Exemplo

Este fragmento torna B o acionador corrente:

```
unsigned num;

_dos_setdrive(2, &num);
```

Função Relacionada

_dos_getdrive()

#include <dos.h>
unsigned _dos_setfileattr(char *fname, unsigned *attrib);

A função **_dos_setfileattr()** não é definida pelo padrão C ANSI e aplica-se apenas a compiladores C baseados em DOS, podendo aparecer com um nome ligeiramente diferente. Verifique seu manual do usuário.

A função **_dos_setfileattr()** estabelece o atributo do arquivo especificado por *fname* para o atributo especificado por *attrib*, que deve ser um dos seguintes valores (as macros estão definidas pelo Microsoft em DOS.H):

Macro	Significado
_A_NORMAL	Arquivo normal
_A_RDONLY	Arquivo de apenas leitura
_A_HIDDEN	Arquivo oculto

_A_SYSTEM Arquivo de sistema
_A_VOLID Rótulo de volume
_A_SUBDIR Subdiretório
_A_ARCH Bit de arquivo alterado

A função **_dos_setfileattr()** devolve zero se bem-sucedida e um valor diferente de zero, caso contrário. Se ocorre uma falha, **errno** recebe **ENOENT** (arquivo não encontrado) ou **EACCES** (acesso negado).

Exemplo

O fragmento seguinte fixa o arquivo TEST.TST para apenas leitura.

```
unsigned attr;

attr = _A_RDONLY;

if(_dos_setfileattr("test.tst", &attr))
  printf("Erro de arquivo");
```

Função Relacionada

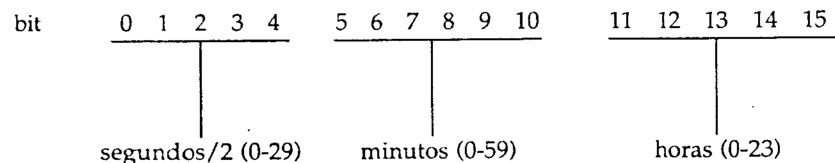
_dos_getfileattr()

#include <dos.h>
unsigned _dos_setftime(int fd, unsigned fdate, unsigned ftime);

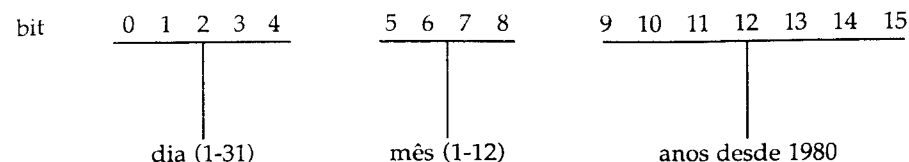
A função **_dos_setftime()** não é definida pelo padrão C ANSI e aplica-se apenas a compiladores C baseados em DOS, podendo aparecer com um nome ligeiramente diferente. Verifique seu manual do usuário.

A função **_dos_setftime()** estabelece a data e a hora do arquivo especificado por *fd*, que deve ser um descritor de arquivo válido.

Os bits no objeto apontado por *ftime* são codificados conforme mostrado na página seguinte:



Os bits no objeto apontado por *fdate* são codificados como segue:



Como indicado, o ano é representado como o número de anos desde 1980. Assim, se o ano é 2000, o valor dos bits de 9 a 15 é 20.

A função **_dos_setftime()** devolve zero caso seja bem-sucedida. Se ocorre um erro, ela devolve um valor diferente de zero e ajusta **errno** para que seja igual a **EBADF** (número de arquivo ruim).

Lembre-se de que os arquivos associados a descritores de arquivo utilizam o sistema de E/S tipo UNIX, que não é definido ou relacionado com o sistema de arquivo C ANSI.

Exemplo

Este programa muda o ano em que o arquivo TEST.TST foi criado para 2000:

```
#include <io.h>
#include <dos.h>
#include <fnctl.h>
#include <stdlib.h>
#include <stdio.h>

void main(void)
{
union {
    struct {
      unsigned day: 5;
      unsigned month: 4;
      unsigned year: 7;
    } d;
    unsigned u;
   } date;

   unsigned t;
   int fd;
```

```
    if((fd=open("TEST.TST", O_RDONLY))==-1) {
      printf("Arquivo não pode ser aberto.");
      exit(1);
    }

    _dos_getftime(fd, &date.u, &t);
    date.year =20;

    _dos_setftime(fd, date.u, t);

    close(fd);
  }
```

Função Relacionada

_dos_getftime()

#include <dos.h>
void _dos_setvect(unsigned intr, void (__interrupt __far *isr)( ));

A função **_dos_setvect()** não é definida pelo padrão C ANSI e aplica-se apenas a compiladores C baseados em DOS, podendo aparecer com um nome ligeiramente diferente. Verifique seu manual do usuário.

A função **_dos_setvect()** coloca o endereço da rotina de serviço de interrupção apontada por *isr* na tabela de vetores de interrupção na posição especificada por *intr*.

Função Relacionada

_dos_getvect()

#include <dos.h>
unsigned _dos_write(int fd, void__far *buf, unsigned count, unsigned *numwritten);

A função **_dos_write()** não é definida pelo padrão C ANSI e aplica-se apenas a compiladores C baseados em DOS, podendo aparecer com um nome ligeiramente diferente. Verifique seu manual do usuário.

A função **_dos_write()** escreve *count* bytes no arquivo especificado pelo descritor de arquivo *fd* do buffer apontado por *buf*. O número de bytes realmente escritos é devolvido em *numwritten*.

Caso seja bem-sucedida, **_dos_write()** devolve zero; um valor diferente de zero é devolvido em caso de falha. O valor devolvido é determinado pelo DOS; você precisará da documentação técnica do DOS para determinar a natureza de quaisquer erros que ocorram. Além disso, em caso de falha, **errno** recebe ou **EACCES** (acesso negado) ou **EBADF** (arquivo não existe).

Exemplo

Este fragmento escreve 128 caracteres no arquivo descrito por **fd**:

```
unsigned count;
char *buf[128];
.
.
.
if(_dos_write(fd, buf, 128, &count))
  printf("Erro ao escrever no arquivo.");
```

Função Relacionada

_dos_read()

#include <dos.h>
void _enable(void);

A função **_enable()** não é definida pelo padrão C ANSI e aplica-se apenas a compiladores C baseados em DOS, podendo aparecer com um nome ligeiramente diferente. Verifique seu manual do usuário.

A função **_enable()** habilita as interrupções.

Função Relacionada

_disable()

#include <dos.h>
unsigned FP_OFF(void __far *ptr);
unsigned FP_SEG(void __far *ptr);

As macros **FP_OFF()** e **FP_SEG()** não são definidas pelo padrão C ANSI e aplicam-se apenas a compiladores C baseados em DOS, podendo aparecer com nomes ligeiramente diferentes. Verifique seu manual do usuário.

A macro **FP_OFF()** devolve o *offset* do ponteiro far *ptr*. A macro **FP_SEG** devolve o segmento do ponteiro far *ptr*.

Exemplo

Este programa escreve o segmento e o *offset* do ponteiro far **ptr**

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

void main(void)
{
  char __far *ptr;

  ptr = (char __far *) malloc(100);

  printf("segmento:offset de ptr: %u %u", FP_SEG(ptr),
          FP_OFF(ptr));
}
```

#include <time.h>
struct tm *gmtime(const time_t *time);

A função **gmtime()** devolve um ponteiro para a forma decomposta de *time* na forma de uma estrutura **tm**. O horário é representado pela hora universal (Universal Time Coordinated — UTC). O valor de *time* geralmente é obtido por meio de uma chamada a **time()**. Se UTC não for suportado pelo sistema, um ponteiro nulo é retornado.

A estrutura usada por **gmtime()** para guardar a hora decomposta é alocada estaticamente e sobrescrita toda vez que a função é chamada. Para salvar o conteúdo da estrutura, você precisa copiá-la em algum outro lugar.

Exemplo

Este programa imprime tanto a hora local como a UTC (hora universal) do sistema:

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

/* Imprime a hora local e de UTC */
void main(void)
```



```
{
  struct tm *local, *gm;
  time_t t;

  t = time(NULL);
  local = localtime(&t);
  printf("Hora local e a data: %s\n", asctime(local));
  gm = gmtime(&t);
  printf("Hora universal e data: %s", asctime(gm));
}
```

Funções Relacionadas

asctime(), localtime(), time()

#include <dos.h>
void _harderr(void (__far *int_handler)());
void _hardresume(int code);
void _hardretn(int code);

As funções **_harderr()**, **_hardresume()** e **_hardretn()** não são definidas pelo padrão C ANSI e aplicam-se apenas a compiladores C baseados em DOS, podendo aparecer com nomes ligeiramente diferentes. Verifique seu manual do usuário.

A função **_harderr()** permite que você substitua o tratador de erro de _hardware padrão por um criado por você. A função é chamada com o endereço da função que se tornará a nova rotina de tratamento de erro. Ela será executada toda vez que ocorrer uma interrupção 0x24.

O tratador da interrupção de erro pode terminar de uma entre três formas. Primeiro, a função **hardresume()** faz com que o tratador vá para o DOS, devolvendo o valor de *code*. Segundo, o tratador pode executar um **return**, que provoca uma saída para o DOS. Terceiro, o tratador pode retornar ao programa via chamada a **hardretn()**, com o valor de retorno de *code*.

As funções de serviço de interrupção são complexas, por isso nenhum exemplo é mostrado. Além disso, a implementação dessas funções varia enormemente de compilador para compilador. Veja seu manual do usuário para detalhes.

#include <dos.h>
int _int86(int int_num, union _REGS *in_regs, union _REGS *out_regs);
int _int86x(int int_num, union _REGS *in_regs, union _REGS *out_regs, struct _SREGS *sregs);

As funções **_int86()** e **_int86x()** não são definidas pelo padrão C ANSI e aplicam-se apenas a compiladores C baseados em DOS, podendo aparecer com nomes ligeiramente diferentes. Verifique seu manual do usuário.

A função **_int86()** executa uma interrupção de software especificada por *int_num*. O conteúdo da união *in_regs* é primeiro copiado nos registradores do processador; em seguida, a interrupção adequada é executada.

Ao retornar, a união *out_regs* contém os valores dos registradores que a CPU tem ao retornar da interrupção. O valor devolvido por **_int86()** é o valor do registrador **AX** após a interrupção.

A união **_REGS** é definida no cabeçalho DOS.H.

A função **_int86x()** é idêntica à função **int86()**, exceto por ser possível determinar os valores dos registradores de segmento do 8086, **ES** e **DS**, usando o parâmetro *sregs*. Ao retornar da chamada, o conteúdo do objeto apontado por *sregs* contém os valores dos registradores de segmento atuais. O registrador **DS** é automaticamente restaurado ao seu valor anterior à chamada a **int86x()**.

Exemplo

A função **_int86()** geralmente é utilizada para chamar rotinas em ROM no BIOS. Por exemplo, esta função executa uma função INT 10H código 0, que estabelece o modo de vídeo para aquele especificado pelo argumento **mode**.

```
#include <dos.h>

set_mode(char mode)
{
  union _REGS in, out;

  in.h.al = mode;
  in.h.ah = 0; /* define número da função */

  _int86(0x10, &in, &out);
}
```


Funções Relacionadas

_bdos(), _intdos()

#include <dos.h>
int _intdos(union _REGS *in_regs, union _REGS *out_regs);
int _intdosx(union _REGS *in_regs, union _REGS *out_regs, struct _SREGS *segregs);

As funções **_intdos()** e **_intdosx()** não são definidas pelo padrão C ANSI e aplicam-se apenas a compiladores C baseados em DOS, podendo aparecer com nomes ligeiramente diferentes. Verifique seu manual do usuário.

A função **_intdos()** efetua uma chamada à função do DOS especificada pelo conteúdo da união apontada por *in_regs*. Ela executa uma instrução INT 21H e o resultado da operação é colocado na união apontada por *out_regs*. A função **intdos()** devolve o valor do registrador **AX**, que é utilizado pelo DOS para devolver informações. Ao retornar, se o indicador de carry estiver ativado, isso significa que ocorreu um erro.

A função **_intdos()** acessa as chamadas ao sistema que precisam de argumentos em registradores diferentes de **DX** e **AL** ou que devolvem informações em um registrador diferente de **AX**.

A união **_REGS** define os registradores da família dos processadores 8088/8086 e encontra-se no arquivo de cabeçalho DOS.H.

Para **_intdosx()**, o valor de *segregs* especifica os registradores **DS** e **ES**. Essa função é utilizada principalmente em programas compilados para o modelo large de dados.

Exemplo

O programa seguinte lê a hora diretamente do relógio do sistema, contornando todas as funções de horário de C.

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

void main(void)
{
  union _REGS in, out;

  in.h.ah = 0x2c; /* número da função que obtém o horário */
```

```
  intdos(&in, &out);
  printf("hora é: %.2d:%.2d:%.2d", out.h.ch, out.h.cl, out.h.dh);
}
```

Funções Relacionadas

_bdos(), _int86()

#include <locale.h>
struct lconv *localeconv(void);

A função **localeconv()** obtém as definições correntes de localização que se referem a valores numéricos, colocando-as em uma estrutura alocada estaticamente do tipo **lconv**. Ela retorna um ponteiro para esta estrutura. Esta estrutura não deve ser modificada por seu programa.

A estrutura **lconv** é definida assim:

```
struct lconv {
  char *decimal_point;     /* caractere de ponto decimal
                              para valores não-monetários */
  char *thousands_sep;     /* separador de milhares
                              para valores não-monetários */
  char *grouping;          /* especifica o agrupamento
                              para valores não-monetários */
  char *int_curr_symbol; /* símbolo internacional de moeda */
  char *currency_symbol; /* símbolo da moeda local */
  char *mon_decimal_point;/* caractere de ponto decimal
                              para valores monetários */
  char *mon_thousands_sep;/* separador de milhar
                              para valores monetários */
  char *mon_grouping;      /* especifica o agrupamento
                              para valores monetários */
  char *positive_sign;     /* indicador de valor positivo
                              para valores monetários */
  char *negative_sign;     /* indicador de valor negativo
                              para valores monetários */
  char int_frac_digits;    /* número de dígitos exibidos
                              à direita do ponto decimal
                              para valores monetários
                              exibidos usando o formato
                              internacional */
  char frac_digits;        /* número de dígitos exibidos
                              à direita do ponto decimal
                              para valores monetários
                              exibidos usando o formato
                              local */
  char p_cs_precedes;      /*1 se o símbolo da moeda
                              precede valores positivos,
                              0 se o símbolo segue o valor */
  char p_sep_by_space;     /*1 se o símbolo da moeda é
                              separado do valor por um espaço,
                              0 caso contrário */
  char n_cs_precedes;      /* 1 se o símbolo da moeda
                              precede valores negativos,
                              0 se o símbolo segue o valor */
  char n_sep_by_space;     /* 1 se o símbolo da moeda é
                              separado de valores negativos por
                              um espaço, 0 caso contrário */
  char p_sign_posn;        /* indica a posição do símbolo de
                              valor positivo */
  char n_sign_posn;        /* indica a posição do símbolo de
                              valor negativo */
};
```


Exemplo

O próximo programa exibe o caractere de ponto decimal usado pela localização corrente:

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

void main(void)
{
  struct lconv lc;

  lc = *localeconv();

  printf("Símbolo decimal é: %s\n", lc.decimal_point);
}
```

Função Relacionada

setlocale()

#include <time.h>
struct tm *localtime(const time_t *time);

A função **localtime()** devolve um ponteiro para a forma decomposta de *time* em uma estrutura **tm**. O horário é representado em hora local. O valor de *time* geralmente é obtido por meio de uma chamada a **time()**.

A estrutura usada por **localtime()** para guardar a hora decomposta é alocada estaticamente e sobrescrita toda vez que a função é chamada. Para salvar o conteúdo da estrutura, você precisa copiá-la em algum outro lugar.

Exemplo

Este programa imprime tanto a hora local como a hora universal (UTC):

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

/* Imprime a hora local e universal. */
void main(void)
{
  struct tm *local;
  time_t t;

  t = time(NULL);
  local = localtime(&t);
  printf("Hora local e a data: %s\n", asctime(local));
  local = gmtime(&t);
  printf("Hora de UTC e data: %s\n", asctime(local));
}
```

Funções Relacionadas

asctime(), gmtime(), time()

#include <time.h>
time_t mktime(struct tm *time);

A função **mktime()** devolve o horário de calendário equivalente ao horário decomposto da estrutura apontada por *time*. Esta função é utilizada principalmente para inicializar o horário do sistema. Os elementos **tm_wday** e **tm_yday** são estabelecidos pela função, assim, eles não precisam ser definidos no momento da chamada.

Se **mktime()** não puder representar a informação como um horário válido de calendário, ela devolverá -1.

Exemplo

Este programa lhe diz o dia da semana de 3 de janeiro de 1999:

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

void main(void)
{
  struct tm t;
  time_t t_of_day;

  t.tm_year = 1999-1900;
  t.tm_mon = 0;
  t.tm_mday = 3;
  t.tm_hour = 0; /* hora, minuto e segundo não importam */
  t.tm_min = 0;  /* contanto que eles não façam com que */
  t.tm_sec = 1;  /* mude o dia */
  t.tm_isdst = 0;

  t_of_day = mktime(&t);
  printf(ctime(&t_of_day));
}
```

Funções Relacionadas

asctime(), ctime(), gmtime(), time()

#include <dos.h>
void _segread(struct _SREGS *sregs);

A função **_segread()** não é definida pelo padrão C ANSI e aplica-se apenas a compiladores C baseados em DOS, podendo aparecer com um nome ligeiramente diferente. Verifique seu manual do usuário.

A função **_segread()** copia os valores atuais dos registradores de segmento (da família) na estrutura **SREGS** apontada por *sregs*.

#include <locale.h>
char *setlocale(int type, const char *locale);

A função **setlocale()** permite que você examine ou estabeleça certos parâmetros que são sensíveis à localização geopolítica do programa. Por exemplo, na Europa, a vírgula é utilizada no lugar do ponto decimal.

Se *locale* é nulo, **setlocale()** devolve um ponteiro para a string de localização atual. Caso contrário, **setlocale()** tenta utilizar a string de localização especificada para estabelecer os parâmetros da localidade como especificado por *type*.

No momento da chamada, *type* deve ser uma das seguintes macros:

LC_ALL
LC_COLLATE
LC_CTYPE
LC_MONETARY
LC_NUMERIC
LC_TIME

LC_ALL refere-se a todas as categorias de localização. **LC_COLLATE** afeta a operação da função **strcoll()**. **LC_CTYPE** altera a maneira pela qual as funções de caracteres operam. **LC_MONETARY** determina o formato monetário. **LC_NUMERIC** muda o caractere de ponto decimal para as funções de entrada/saída. Finalmente, **LC_TIME** determina o comportamento da função **strftime()**.

O padrão C ANSI define duas strings possíveis para *locale*. A primeira é "C", que especifica um ambiente mínimo para compilação C. A segunda é "", a string nula, que especifica o ambiente padrão definido pela implementação. Todos os outros valores para *locale* são definidos pela implementação e afetam a portabilidade.

A função **setlocale()** devolve um ponteiro para uma string associada ao parâmetro *type*. Se a solicitação não pode ser atendida, **setlocale()** retorna nulo.

Exemplo

Este programa mostra a disposição da localidade atual:

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

void main(void)
{
  printf(setlocale(LC_ALL,""));
}
```


Funções Relacionadas

localeconv(), strcoll(), strftime(), time()

#include <time.h>
size_t strftime(char *str, size_t maxsize, const char *fmt, const struct tm *time);

A função **strftime()** coloca as informações de horário e de data, junto a outras informações, na string apontada por *str*. **strftime()** utiliza os comandos de formato encontrados na string apontada por *fmt* e o horário decomposto *time*. Um máximo de caracteres *maxsize* é colocado em *str*.

Tabela 15.1 Os comandos de formato de **strftime()**.

Comando	Substituído por
%a	Nome do dia da semana abreviado
%A	Nome do dia da semana completo
%b	Nome do mês abreviado
%B	Nome do mês completo
%c	String de hora e data padrão
%d	Dia do mês em decimal (1-31)
%H	Hora, na faixa (0-23)
%I	Hora, na faixa (1-12)
%j	Dia do ano em decimal (1-366)
%m	Mês em decimal (1-12)
%M	Minutos em decimal (0-59)
%p	Equivalente local de AM e PM
%S	Segundos em decimal (0-61)
%U	Semana do ano, domingo sendo o primeiro dia (0-52)
%w	Dia da semana em decimal (0-6, domingo sendo 0)
%W	Semana do ano, segunda-feira sendo o primeiro dia (0-53)
%x	String de data padrão
%X	String de hora padrão
%y	Ano em decimal sem século (00-99)
%Y	Ano incluindo século em decimal
%Z	Nome da zona de tempo
%%	O sinal de porcentagem

A função **strftime()** opera de forma semelhante à função **sprintf()**. Ela reconhece um conjunto de comandos de formato que começa com um sinal de porcentagem (%) e coloca sua saída formatada na string. Os comandos de formato especificam a maneira exata em que as diversas informações de horário e data são representadas em *str*. Qualquer outro caractere encontrado na string é colocado, sem alteração, em *str*. A hora e a data mostradas estão em hora local. Os comandos de formato são mostrados na Tabela 15.1. Observe que muitos deles são sensíveis à diferença entre maiúsculas e minúsculas.

A função **strftime()** devolve o número de caracteres colocado na string apontada por *str*. Se ocorre um erro, a função devolve zero.

Exemplo

Assumindo que **ltime** aponta para uma estrutura que contém 10:00:00 AM Jan 2, 1994, o fragmento seguinte escreve **Agora são 10 AM.**

```
strftime(str, 100, "Agora são %H %p", ltime);
printf(str);
```

Funções Relacionadas

gmtime(), localtime(), time()

#include <time.h>
time_t time(time_t *time);

A função **time()** devolve o horário atual de calendário do sistema. Se o sistema não tem horário, **time()** devolve -1.

A função **time()** pode ser chamada com um ponteiro nulo ou com um ponteiro para uma variável do tipo **time_t**. Nesse caso, o argumento também recebe o horário de calendário.

Exemplo

Este programa mostra a hora local definida pelo sistema:

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

void main(void)
{
  struct tm *ptr;
  time_t lt;
```



```
  lt = time(NULL);
  ptr = localtime(&lt);
  printf(asctime(ptr));
}
```

Funções Relacionadas

ctime(), gmtime(), localtime(), strftime()

MAKRON Books

16

Alocação Dinâmica

Existem duas maneiras fundamentais de um programa em C poder armazenar informações na memória principal do computador. O primeiro método usa variáveis locais e globais — incluindo matrizes e estruturas. No caso de variáveis globais e locais estáticas, o armazenamento é fixo durante todo o tempo de execução do programa. Para variáveis locais, o armazenamento é alocado do espaço da pilha do computador. Embora essas variáveis sejam implementadas eficientemente em C, elas exigem que o programador saiba, de antemão, a quantidade de armazenamento necessária para todas as situações em que o programa possa se encontrar — algo que nem sempre é possível. De fato, alguns programas precisam ser capazes de ajustar as suas necessidades de armazenamento como resposta a eventos que só serão conhecidos durante a execução do programa.

Para oferecer um meio pelo qual o programa possa obter espaço para armazenamento em tempo de execução, o C inclui um subsistema de *alocação dinâmica*. A alocação dinâmica é a segunda maneira pela qual um programa pode obter espaço de armazenamento para dados. Nesse método, o armazenamento para a informação é alocado da memória livre, também chamada de *heap*, conforme suas necessidades. A região de memória livre está situada entre seu programa, com sua área de armazenamento permanente, e a pilha. A Figura 16.1 mostra conceitualmente como um programa em C aparece na memória. (Para a família 8086 de processadores, a posição do heap muda dependendo do modelo de memória utilizado. Os modelos de memória do 8086 serão discutidos resumidamente.) A pilha cresce para baixo conforme ela é usada; assim, a quantidade de memória de que ela precisa depende de como seu programa é construído. Por exemplo, um programa com muitas funções recursivas tem uma demanda muito maior de memória de pilha do que um que não tem funções recursivas,

pois as variáveis locais são armazenadas na pilha. A memória necessária para o programa e as variáveis globais são fixas durante a execução do programa. A memória para satisfazer uma solicitação de alocação é retirada da área de memória livre, começando logo acima das variáveis globais e crescendo na direção da pilha. Como você poderia supor, sob casos relativamente extremos, a pilha pode colidir com o heap. O fato de o heap poder se esgotar implica que um pedido de alocação de memória pode falhar. (E, de fato, falha.)

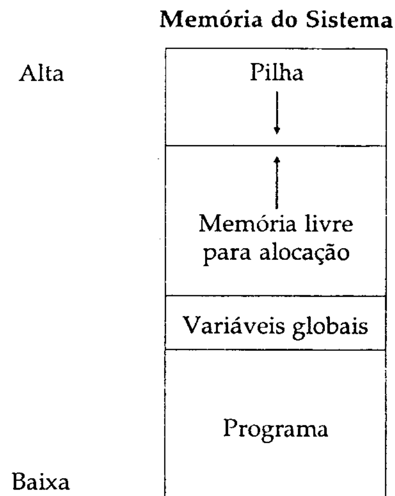


Figura 16.1 O uso da memória de um programa em C.

No núcleo do sistema de alocação dinâmica de C estão as funções **malloc()** e **free()** — parte da biblioteca C padrão. Toda vez que **malloc()** requisita memória, uma porção da memória livre restante é alocada. Toda vez que é feita uma chamada a **free()** para liberar memória, memória é devolvida para o sistema. A maneira mais comum de implementar **malloc()** e **free()** é organizar a memória livre em uma lista encadeada. No entanto, o método de gerenciamento de memória depende da implementação.

O padrão C ANSI especifica que os protótipos para as funções de alocação dinâmica definidas pelo padrão estão em STDLIB.H.

O padrão C ANSI define apenas quatro funções para o sistema de alocação dinâmica: **calloc()**, **malloc()**, **free()** e **realloc()**. No entanto, este livro examina várias outras que estão em uso de forma massiva — especialmente na

família 8086 de ambientes de CPU. A base para as funções não-padrões é o Microsoft C versão 5.1, mas muitos compiladores terão funções similares, embora com nomes diferentes. As funções não-padrões usam o arquivo de cabeçalho MALLOC.H. Algumas dessas funções adicionais são necessárias para suportar a arquitetura segmentada da família de processadores 8086 de forma eficiente. (Essas funções não dizem respeito a compiladores projetados para outros processadores ou para processadores mais novos da família 8086 quando não estiverem executando no modo de compatibilidade DOS). Por causa da memória segmentada da família 8086 de processadores, geralmente são suportados pelos compiladores 3 modificadores de tipo fora do padrão, específicos para estes processadores. Para muitos compiladores estes novos tipos são chamados **near**, **far** e **huge**. No entanto, a Microsoft os chama **__near**, **__far** e **__huge**. Neste capítulo discutimos as funções de alocação baseadas em 8086 na versão Microsoft, usaremos estas palavras-chaves. (Os sublinhados iniciais garantem compatibilidade como padrão ANSI). Estes tipos são usados para criar ponteiros de um tipo diferente daquele que seria usado normalmente em função do modelo de memória usado para compilar o programa. A seguinte discussão explica os modelos de memória segmentada do 8086.

Os Modelos de Memória Segmentada do 8086

Quando operada no modo segmentado, a famíla 8086 de processadores vê a memória como um conjunto de fatias de 64 K; cada fatia é chamada de *segmento*. Cada byte na memória é definido por seu endereço de segmento (contido em um registrador de segmento da CPU) e seu offset, ou deslocamento (contido em outro registrador), dentro desse segmento. Tanto o segmento como o offset usam valores de 16 bits. Quando um endereço que está sendo acessado pertence ao segmento atual, apenas os 16 bits do offset precisam ser carregados para acessar um byte específico na memória. No entanto, se o endereço de memória está fora do segmento atual, os 16 bits do segmento e os 16 bits do offset precisam ser carregados. Dessa forma, para acessar a memória dentro do segmento, o compilador C pode tratar um ponteiro ou uma instrução de call ou jump como um objeto de 16 bits. Para acessar a memória fora do segmento atual, o compilador deve tratar um ponteiro ou uma instrução de call ou jump como uma entidade de 32 bits.

Em razão da natureza segmentada da família 8086, você pode organizar a memória em um destes seis modelos (mostrados em ordem crescente de tempo de execução):

Tiny (minúsculo) Todos os registradores de segmento são colocados no mesmo valor e todo o endereçamento é feito usando 16 bits. Isso significa que o código, os dados e a pilha devem todos encaixar-se no mesmo segmento de 64 K. A mais rápida execução do programa.

Small (pequeno) Todo o código se enquadra em um segmento de 64 K e todos os dados devem caber em um segundo segmento de 64 K. Todos os ponteiros são de 16 bits. Tão veloz quanto o modelo *tiny*.

Medium (médio) Todos os dados devem caber em um segmento de 64 K, mas o código pode usar segmentos múltiplos. Todos os ponteiros para os dados são de 16 bits, mas todos os jumps e calls exigem endereços de 32 bits. Rápido acesso aos dados, execução mais lenta do código.

Compact (compacto) Todo o código deve caber em um segmento de 64 K, mas os dados podem usar múltiplos segmentos. No entanto, nenhum item de dado pode exceder 64 K. Todos os ponteiros para dados são de 32 bits, mas os jumps e calls podem usar endereços de 16 bits. Acesso lento aos dados, execução mais rápida do código.

Large (grande) Tanto código como dados podem usar segmentos múltiplos. Todos os ponteiros são de 32 bits. No entanto, nenhum item único de dado pode exceder 64 K. Execução mais lenta do programa.

Huge (enorme) Tanto código como dados podem usar segmentos múltiplos. Todos os ponteiros são de 32 bits. Itens únicos podem exceder 64 K. Execução mais lenta do programa.

Como você poderia supor, é muito mais rápido acessar a memória via ponteiros de 16 bits do que com de 32, pois metade dos bits devem ser carregados na CPU para cada referência à memória.

Ocasionalmente, você precisará fazer referência a um ponteiro que é diferente do fornecido pelo modelo de memória. Muitos compiladores C baseados em 8086 permitem que ponteiros de 16 ou 32 bits sejam criados explicitamente pelo seu programa, contornando, assim, o modelo padrão de memória. Isso normalmente ocorre quando um programa exige muitos dados para uma operação específica. Nesse caso, um ponteiro **far** é criado e a memória é alocada com a versão não-padrão de **malloc()**, que aloca memória fora do segmento de dados padrão. Dessa forma, todos os outros acessos à memória permanecem rápidos e o tempo de execução não sofre tanto como se tivesse sido usado um modelo maior. O inverso também pode acontecer: um programa que usa um modelo maior pode estabelecer um ponteiro **near** para uma porção da memória freqüentemente acessada para aumentar o desempenho. Como os métodos reais de sobrepor o modelo de memória diferem de compilador para compilador, verifique seu manual do usuário.

Muitos compiladores C baseados em 8086 restringem o tamanho de um item único de dados para 64 K — o tamanho de um segmento. No entanto, você pode usar o modificador **_huge** para criar um ponteiro que pode ser usado com objetos maiores que 64 K.

#include <malloc.h>
void *_alloca(size_t size);

A função **_alloca()** não é definida pelo padrão C ANSI. Ela pode, ainda, ter um nome ligeiramente diferente; verifique seu manual do usuário.

A função **_alloca()** aloca *size* bytes de memória da pilha do sistema (não do heap) e devolve um ponteiro de caracteres para essa região. Um ponteiro nulo é devolvido se a solicitação não puder ser cumprida.

A memória alocada com **_alloca()** é automaticamente liberada quando a função que a chamou retorna. Isso significa que você nunca deve utilizar um ponteiro gerado por **_alloca()** como um argumento para **free()**.

Exemplo

O código seguinte aloca 80 bytes da pilha usando **_alloca()**:

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

void main(void)
{
  char *str;

  if(!(str = _alloca(80))) {
    printf("Erro de alocação - abortando.");
    exit(1);
  }
  .
  .
  .
}
```

Funções Relacionadas

malloc(), stackavail()

#include <stdlib.h>
void *calloc(size_t num, size_t size);

A função **calloc()** aloca uma quantidade de memória igual a *num* * *size*. Isto é, **calloc()** aloca memória suficiente para uma matriz de *num* objetos de tamanho *size*.

A função **calloc()** devolve um ponteiro para o primeiro byte da região alocada. Se não houver memória suficiente para satisfazer a solicitação, é devolvido um ponteiro nulo. Você sempre deve verificar se o valor devolvido não é um ponteiro nulo antes de usá-lo. Memória alocada usando **calloc()** deve ser liberada usando **free()**.

Exemplo

Esta função devolve um ponteiro para uma matriz de 100 **floats** alocada dinamicamente:

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

float *get_mem(void)
{
  float *p;

  p = calloc(100, sizeof(float));
  if(!p) {
    printf("Erro de alocação - abortando.");
    exit(1);
  }
  return p;
}
```

Funções Relacionadas

free(), malloc(), realloc()

#include <malloc.L>
void__far *_fcalloc(size_t num, size_t size);

A função **_fcalloc()** não é definida pelo padrão ANSI e se aplica à maioria dos compiladores para a CPU 8086. Ela também pode ter um nome ligeiramente diferente, portanto verifique seu manual de usuário.

A função **_fcalloc()** aloca a quantidade de memória equivalente a num*size. Em outras palavras, **_fcalloc()** aloca memória suficiente para um vetor de num elementos, onde cada elemento ocupa size bytes. A memória é sempre alocada fora do segmento padrão de dados. (Nos demais aspectos é semelhante a **calloc()**. Veja **calloc()** para ver um exemplo.)

A função **_fcalloc()** retorna um ponteiro para o primeiro byte da região alocada. Se não houver memória suficiente para atender o pedido, será retornado um ponteiro nulo. Você deveria sempre verificar que o valor de retorno não seja nulo antes de utilizá-lo.

Funções Relacionadas

_ffree(), _fmalloc(), _frealloc()

#include <malloc.h>
void _ffree(void __far *ptr);

A função **_ffree()** não é definida pelo padrão C ANSI e aplica-se à maioria dos compiladores baseados em 8086. Ela pode, ainda, ter um nome ligeiramente diferente; verifique seu manual do usuário.

A função **_ffree()** devolve ao sistema a memória apontada pelo ponteiro **far** *ptr*. Isso torna a memória disponível para futura alocação. O ponteiro deve ter sido previamente alocado utilizando-se **_fmalloc()**, **_frealloc ()** ou **_fcalloc ()**.

Ela não pode liberar ponteiros alocados por outras funções de alocação. Se for utilizado um ponteiro inválido na chamada, geralmente ocorre a destruição do mecanismo de gerenciamento da memória, provocando uma quebra do sistema.

Exemplo

Este programa aloca memória fora do segmento de dados padrão e, em seguida, libera-a. Observe a utilização de **__far** para estabelecer um ponteiro **far**. Lembre-se de que **__far** não faz parte da linguagem C e só diz respeito a compiladores baseados na família 8086 de processadores.

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

void main(void)
{
  char __far *str;

  if((str = _fmalloc(128))==NULL) {
     printf("Erro de alocação - abortando.");
     exit(1);
  }

  /* agora libera a memória */
  _ffree(str);
}
```


Funções Relacionadas

_fcalloc(), _fmalloc(), _frealloc()

#include <malloc.h>
void __far *_fmalloc(size_t size);

A função **_fmalloc()** não é definida pelo padrão C ANSI e aplica-se à maioria dos compiladores baseados em 8086. Ela pode, ainda, ter um nome ligeiramente diferente; verifique seu manual do usuário.

A função **_fmalloc()** devolve um ponteiro **far** para o primeiro byte de uma região de memória de tamanho *size* que foi alocada fora do segmento de dados padrão. Se não há memória suficiente fora do segmento de dados padrão, o heap será tentado. Se ambos falham em satisfazer a solicitação de **_fmalloc()**, um ponteiro nulo é, então, devolvido. Você deve sempre verificar se o valor devolvido não é um ponteiro nulo antes de utilizá-lo.

Exemplo

Esta função aloca memória suficiente, fora do segmento de dados padrão, para guardar estruturas do tipo **addr**. Observe a utilização de **__far** para estabelecer um ponteiro **far**. Lembre-se de que **__far** não faz parte da linguagem C e só diz respeito a compiladores baseados na família 8086 de processadores.

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

struct addr {
  char name[40];
  char street[40];
  char city[40];
  char state[3];
  char zip[10];
};

struct addr __far *get_struct(void)
{
  struct addr __far *p;

  if((p = _fmalloc(sizeof(struct addr)))==NULL) {
    printf("Erro de alocação - abortando.");
    exit(1);
  }
  return p;
}
```

Funções Relacionadas

_fcalloc(), _ffree(), _frealloc()

#include <malloc.h>
size_t _fmsize(void __far *ptr);

A função **_fmsize()** não é definida pelo padrão C ANSI e aplica-se à maioria dos compiladores baseados em 8086. Ela pode, ainda, ter um nome ligeiramente diferente; verifique seu manual do usuário.

A função **_fmsize()** devolve o número de bytes do bloco de memória alocado apontado pelo ponteiro **far** *ptr*. Essa memória deve ter sido alocada com **_fmalloc()**, **_frealloc()** ou **_fcalloc()**.

Exemplo

Este programa mostra o tamanho do bloco de memória necessário para conter a estrutura **addr**:

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

struct addr {
  char name[40];
  char street[40];
  char city[40];
  char state[3];
  char zip[10];
};

void main(void)
{
  struct addr __far *p;

  p = _fmalloc(sizeof(struct addr));

  printf("Tamanho do bloco é %u.", _fmsize(p));
}
```

Função Relacionada

_fmalloc()

#include <malloc.h>
void __far*_frealloc(void __far *ptr, size_t size);

A função **_frealloc()** não é definida pelo padrão C ANSI e aplica-se principalmente a compiladores baseados em 8086. Pode ser também que tenha um nome ligeiramente diferente, portanto verifique seu manual do usuário.

A função **_frealloc()** modifica o tamanho da área de memória alocada anteriormente e apontada por *ptr* para o tamanho *size*. O valor de *size* pode ser maior ou menor do que o tamanho original. A memória apontada por *ptr* deve ter sido alocada anteriormente usando **_fmalloc()** ou **_fcalloc()**.

_frealloc() pode precisar mover o bloco original de memória para poder aumentar seu tamanho. Se isto ocorrer, o conteúdo do bloco antigo é copiado para o bloco novo — nenhuma informação é perdida. Se *ptr* for nulo, **_frealloc()** simplesmente alocará *size* bytes de memória e retorna um ponteiro para eles. Se *size* for zero, a memória apontada por *ptr* será liberada.

_frealloc() retorna um ponteiro para o bloco de memória já mudado de tamanho, que será alocado do heap **far**. Se não houver memória suficiente no heap para alocar *size* bytes, será retornado um ponteiro nulo e o bloco original não será modificado.

_frealloc() é a versão **far** de **realloc()**. Veja **realloc()** para um exemplo.

Funções relacionadas

_fcalloc(), _ffree(), fmalloc()

#include <stdlib.h>
void free(void *ptr);

A função **free()** devolve ao heap a memória apontada por *ptr*, tornando a memória disponível para alocação futura.

free() deve ser chamada somente com um ponteiro que foi previamente alocado com uma das funções do sistema de alocação dinâmica (**malloc()**, **realloc()** ou **calloc()**). A utilização de um ponteiro inválido na chamada provavelmente destruirá o mecanismo de gerenciamento de memória e provocará uma quebra do sistema.

Exemplo

Este programa aloca espaço para as strings digitadas pelo usuário e, em seguida, libera a memória.

Funções Relacionadas

calloc(), _hfree(), malloc(), realloc()

#include <malloc.h>
void _hfree(void __huge *ptr);

A função **_hfree()** não é definida pelo padrão C ANSI e aplica-se à maioria dos compiladores baseados em 8086. Ela pode ainda ter um nome ligeiramente diferente; verifique seu manual do usuário.

A função **_hfree()** devolve ao sistema a memória apontada pelo ponteiro **huge** *ptr*, tornando a memória disponível para alocação futura. O ponteiro deve ter sido previamente alocado com **halloc()**, que aloca blocos de memória maiores que 64 K. A utilização de um ponteiro inválido na chamada provavelmente destruirá o mecanismo de gerenciamento de memória e provocará uma quebra do sistema.

Exemplo

Este programa primeiro aloca memória externa ao segmento de dados padrão e depois libera-a:

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

void main(void)
{
  char __huge *large;

  if((large = _halloc(1, 128000))==NULL) {
    printf("Erro de alocação - abortando.");
    exit(1);
  }

  /* agora libera a memória */
  _hfree(large);
}
```

Funções Relacionadas

calloc(), _halloc(), malloc(), realloc()

#include <stdlib.h>
void *malloc(size_t size);

A função **malloc()** devolve um ponteiro para o primeiro byte de uma região de memória de tamanho *size* que foi alocada do heap. Caso não haja memória suficiente no heap para satisfazer a solicitação, **malloc()** devolve um ponteiro nulo. Você deve sempre verificar se o valor devolvido não é um ponteiro nulo antes de utilizá-lo. A tentativa de usar um ponteiro nulo resultará geralmente numa quebra do sistema.

Exemplo

Esta função aloca memória suficiente para conter uma estrutura do tipo **addr**:

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

struct addr *get_struct(void)
{
  struct addr *p;

  if((p = malloc(sizeof(struct addr)))==NULL) {
    printf("Erro de alocação - abortando.");
    exit(1);
  }
  return p;
}
```

Funções Relacionadas

calloc(), free(), realloc()

#include <malloc.h>
size_t _memavl(void);

A função **_memavl()** não é definida pelo padrão C ANSI. Ela pode, ainda, ter um nome ligeiramente diferente; verifique seu manual do usuário.

A função **_nfree()** devolve ao sistema a memória apontada pelo ponteiro **near** *ptr*, tornando a memória disponível para alocação futura. O ponteiro deve ter sido previamente alocado com **_nmalloc()**, **_ncalloc()** ou **_nrealloc()**. A utilização de um ponteiro inválido na chamada provavelmente destruirá o mecanismo de gerenciamento de memória e provocará uma quebra do sistema.

Exemplo

Este programa primeiro aloca memória interna ao segmento de dados padrão e depois libera-a. Observe a utilização de **_ _near** para estabelecer um ponteiro _ _near. Lembre-se de que **near** não faz parte da linguagem C e só diz respeito a compiladores baseados na família 8086 de processadores.

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

void main(void)
{
  char __near *str;

  if((str = _nmalloc(128))==NULL) {
    printf("Erro de alocação - abortando.");
    exit(1);

  }
  /* agora libera a memória */
  _nfree(str);
}
```

Funções Relacionadas

ncalloc(), _nmalloc(), nrealloc()

#include <malloc.h>
char _ _near *_nmalloc(size_t size);

A função **_nmalloc()** não é definida pelo padrão C ANSI e aplica-se à maioria dos compiladores baseados em 8086. Ela pode, ainda, ter um nome ligeiramente diferente; verifique seu manual do usuário.

A função **_nmalloc()** devolve um ponteiro **near** para o primeiro byte de uma região de memória de tamanho *size* que foi alocada do interior do segmento de dados padrão. Isso somente é significativo para programas compilados em um dos modelos grandes de dados do 8086 e permite a utilização de um ponteiro de 16 bits. Se a solicitação de alocação falha por memória livre insuficiente, **_nmalloc()** devolve um ponteiro nulo. Você deve sempre verificar se o valor devolvido não é um ponteiro nulo antes de usá-lo.

Exemplo

Esta função aloca memória suficiente, dentro do segmento de dados padrão, para conter 128 bytes. Observe a utilização de **_ _near** para estabelecer um ponteiro near. Lembre-se de que **_ _near** não faz parte da linguagem C e só diz respeito a compiladores baseados na família 8086 de processadores.

```
char __near *get_near_ram(void)
{
  char __near *p;

  if((p = _nmalloc(128))==NULL) {
    printf("Erro de alocação - abortando.");
    exit(1);
  }
  return p;
}
```

Funções Relacionadas

_ncalloc(), _nfree(), nrealloc()

#include <malloc.h>
size_t _nmsize(void _ _near *ptr);

A função **_nmsize()** não é definida pelo padrão C ANSI e aplica-se à maioria dos compiladores baseados em 8086. Ela pode, ainda, ter um nome ligeiramente diferente; verifique seu manual do usuário.

A função **_nmsize()** devolve o número de bytes no bloco de memória alocado apontado pelo ponteiro **near** *ptr*. Esta memória deve ter sido alocada com **_nmalloc()**, **_nrealloc ()** ou **_ncalloc()**.

Exemplo

Este programa mostra o tamanho do bloco de memória que é necessário para conter a estrutura **addr**:

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

struct addr {
  char name[40];
  char street[40];
  char city[40];
  char state[3];
  char zip[10];
};

void main(void)
{
  struct addr __near *p;

  p = _nmalloc(sizeof(struct addr));

  printf("O tamanho do bloco é: %u.", _nmsize(p));
}
```

Função Relacionada

_nmalloc()

#include <malloc.h>
void __near*_nrealloc(void __near *ptr, size_t size);

A função **_nrealloc()** não é definida pelo padrão C ANSI e aplica-se principalmente a compiladores baseados em 8086. Pode ser também que tenha um nome ligeiramente diferente, portanto verifique seu manual do usuário.

A função **_nrealloc()** modifica o tamanho da área de memória alocada anteriormente e apontada por *ptr* para o tamanho *size*. O valor de *size* pode ser maior ou menor do que o tamanho original. A memória apontada por *ptr* deve ter sido alocada anteriormente usando **_nmalloc()** ou **_ncalloc()**.

_nrealloc() pode precisar mover o bloco original de memória para poder aumentar seu tamanho. Se isto ocorrer, o conteúdo do bloco antigo é copiado para o bloco novo — nenhuma informação é perdida. Se *ptr* for nulo, **_nrealloc()** simplesmente alocará *size* bytes de memória e retornará um ponteiro para eles. Se *size* for zero, a memória apontada por *ptr* será liberada.

_nrealloc() retorna um ponteiro para o bloco de memória já mudado de tamanho, que será alocado do heap **near**. Se não houver memória suficiente no heap para alocar *size* bytes, será retornado um ponteiro nulo e o bloco original não será modificado.

_nrealloc() é a versão **near** de **realloc()**. Veja **realloc()** para um exemplo.

Funções Relacionadas

_ncalloc(), _nfree(), _nmalloc()

#include <stdlib.h>
void *realloc(void *ptr, size_t size);

A função **realloc()** modifica o tamanho da memória previamente alocada apontada por *ptr* para aquele especificado por *size*. O valor de *size* pode ser maior ou menor que o original.

Um ponteiro para o bloco de memória é devolvido porque **realloc()** pode precisar mover o bloco para aumentar seu tamanho. Se isso ocorre, o conteúdo do bloco antigo é copiado no novo bloco; nenhuma informação é perdida. Se *ptr* é um nulo, **realloc()** simplesmente aloca *size* bytes de memória e devolve um ponteiro para a memória alocada. Se size é zero, a memória apontada por *ptr* é liberada.

realloc() retorna um ponteiro para o bloco redimensionado de memória. Se não há memória livre suficiente no heap para alocar *size* bytes, é devolvido um ponteiro nulo e o bloco original é deixado inalterado.

Exemplo

Este programa primeiro aloca 23 caracteres, copia a string "isso são 22 caracteres" neles e, em seguida, usa **realloc()** para aumentar o tamanho para 24 e, assim, pôr um ponto no final.

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

void main(void)
{
  char *p;

  p = malloc(23);
  if(!p) {
    printf("Erro de alocação - abortando.");
    exit(1);
  }

  strcpy(p, "isso são 22 caracteres");

  p = realloc(p, 24);
  if(!p) {
```

```
        printf("Erro de alocação - abortando.");
        exit(1);
      }

    strcat(p, ".");

    printf(p);

    free(p);
  }
```

Funções Relacionadas

calloc(), free(), malloc()

#include <malloc.h>
size_t _stackavail(void)

A função **_stackavail()** não é definida pelo padrão C ANSI. Ela pode, ainda, ter um nome ligeiramente diferente; verifique seu manual do usuário.

A função **_stackavail()** devolve o número aproximado de bytes disponível na pilha para alocação com **_alloca()**.

Você também pode usar **_stackavail()** para predizer uma possível colisão entre o heap e a pilha que poderia ser gerada por rotinas recursivas, como ilustrado pelo exemplo a seguir.

Exemplo

A função **recurse()** chama a si mesma indefinidamente, usando mais espaço da pilha a cada chamada até que o tamanho da pilha tenha atingido um nível perigosamente baixo.

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

void recurse(void);

void main(void)
{
  recurse();
}
```



```
/* Esta rotina chamará a si mesma até que uma colisão
   entre o heap e a pilha se torne uma "ameaça".
*/

void recurse(void)
{
  printf("%u\n", stackavail());
  if(stackavail() < 1000) return;
  recurse()
}
```

Funções Relacionadas

_alloca(), _freect(), _memavl()

MAKRON Books

17

Funções Gráficas e de Texto

Apesar de o padrão C ANSI não definir funções de telas gráficas ou de texto, elas são importantes para a maioria dos trabalhos de programação atual. O padrão C ANSI não define essas funções em razão das grandes diferenças entre as capacidades e interfaces dos diferentes tipos de hardware. Mesmo os compiladores projetados para o mesmo ambiente normalmente implementam as funções gráficas e de texto de forma bastante diferente. Este capítulo descreve as funções gráficas e de texto mais comuns, que são fornecidas com o Microsoft C/C++. Essas funções operam apenas no DOS. Lembre-se: as funções gráficas fornecidas pelo seu compilador podem ser diferentes, mas os princípios são semelhantes.

NOTA: As funções descritas neste capítulo não se aplicam ao Windows. O Windows fornece seu próprio conjunto de funções gráficas, que são parte da API (Application Program Interface) do Windows. Se você criar programas gráficos em um ambiente Windows, precisará usar as funções gráficas da API.

Os protótipos para as funções gráficas e de texto do Microsoft estão contidos em GRAPH.H, juntamente com vários tipos de estrutura, que são discutidos conforme se tornarem necessários.

Este capítulo discute, primeiro, os diversos modos de vídeo disponíveis para a linha PC de computadores. Você precisa dessa informação auxiliar para entender certas funções gráficas.

Modos de Vídeo do PC

Como você provavelmente sabe, há diversos tipos de adaptadores de vídeo atualmente disponíveis para PCs. Os mais comuns são o monocromático, o CGA (Adaptador Gráfico Colorido), o PC*jr*, o EGA (Adaptador Gráfico Estendido) e o VGA (Matriz Gráfica de Vídeo). Juntos, estes adaptadores suportam diversos modos diferentes de operação em vídeo. A Tabela 17.1 resume esses modos de vídeo. Como você pode ver, alguns modos são para texto e alguns para gráficos. Em um modo texto, apenas texto pode ser apresentado. Em um modo gráfico, tanto texto como gráficos podem ser apresentados.

NOTA: Se seu equipamento possui uma placa de vídeo Super VGA, então ela pode suportar modos de alta resolução além dos relacionados na Tabela 17.1. Esses modos estendidos são não-padrões e podem diferir de uma placa Super VGA para outra. Você precisará consultar o manual do seu compilador para obter detalhes sobre o suporte a estes modos estendidos.

Tabela 17.1 Modos de tela para os diversos adaptadores de vídeo.

Modo	Tipo	Dimensões gráficas	Dimensões de texto
0	Texto, b/p	n/d	40x25
1	Texto, 16 cores	n/d	40x25
2	Texto, b/p	n/d	80x25
3	Texto, 16 cores	n/d	80x25
4	Gráficos, 4 cores	320x200	40x25
5	Gráficos, 4 tons de cinza	320x200	40x25
6	Gráficos, 2 cores	640x200	80x25
7	Texto, b/p	n/d	80x25
8	Gráficos PC*jr* 16 cores (obsoleto)	160x200	20x25
8	Hercules Graphics, 2 cores	720x348	80x25
9	Gráficos PC*jr* 16 cores (obsoleto)	320x200	40x25
10	Reservado		
11	Reservado		
12	Reservado		
13	Gráficos, 16 cores	320x200	40x25
14	Gráficos, 16 cores	640x200	80x25
15	Gráficos, 2 cores	640x350	80x25
16	Gráficos, 16 cores	640x350	80x25
17	Gráficos, 2 cores	640x480	80x30
18	Gráficos, 16 cores	640x480	80x30
19	Gráficos, 256 cores	320x200	40x25

A menor parte da tela endereçável pelo usuário no modo texto é um caractere. A menor parte da tela endereçável pelo usuário em um modo gráfico é um pixel. Na verdade, o termo *pixel* referia-se, originalmente, ao menor elemento individual de fósforo em um monitor de vídeo que podia ser energizado individualmente pelo feixe de varredura. No entanto, o termo foi generalizado para se referir ao menor ponto endereçável de um visor gráfico.

Em um modo texto, as posições individuais dos caracteres na tela são referenciadas pelo número de linha e de coluna. Para muitas implementações, as coordenadas do canto superior esquerdo são 1, 1 ao operar no modo texto. Em um modo gráfico, os pixels individuais são referenciados por suas coordenadas X,Y, sendo X o eixo horizontal. Em qualquer um dos modos gráficos, o canto superior esquerdo da tela é a posição 0,0. (As funções de tela e gráficas do Microsoft descritas aqui refletem um caso comum: a origem para o texto é 1,1 enquanto a origem para gráficos é 0, 0.)

Os exemplos de tela de texto deste capítulo utilizam o modo de vídeo 3, o modo de 80 colunas colorido. As rotinas gráficas usam o modo 18. Se seu hardware não suporta um desses modos, você terá de efetuar as alterações apropriadas nos exemplos.

Quando em um modo de texto colorido, você pode especificar a cor na qual o texto será apresentado. As cores e seus equivalentes inteiros são mostrados aqui:

Cor do texto	Valor
Preto	0
Azul	1
Verde	2
Ciano	3
Vermelho	4
Magenta	5
Marrom	6
Cinza-claro	7
Cinza-escuro	8
Azul-claro	9
Verde-claro	10
Ciano-claro	11
Vermelho-claro	12
Magenta-claro	13
Amarelo	14
Branco	15

Ativar o bit mais significativo por meio da soma do número 128 à cor faz o texto piscar.

As cores de texto para o fundo são mostradas a seguir:

Cor de fundo	Valor
Preto	0
Azul	1
Verde	2
Ciano	3
Vermelho	4
Magenta	5
Marrom	6

Para os modos de gráfico coloridos, as cores de fundo são as seguintes:

Cor de fundo	Valor
Preto	0
Azul	1
Verde	2
Ciano	3
Vermelho	4
Magenta	5
Marrom	6
Cinza-claro	7
Cinza-escuro	8
Azul-claro	9
Verde-claro	10
Ciano-claro	11
Vermelho-claro	12
Magenta-claro	13
Amarelo	14
Branco	15

Em um modo gráfico colorido, a cor de frente é determinada tanto pelo valor da cor como pelo da paleta atualmente selecionada. No modo de vídeo 4 da CGA (gráfico com quatro cores), você tem quatro cores por paleta e quatro paletas a escolher. As cores são numeradas de 0 a 3, com 0 sendo a cor de fundo. As paletas também são numeradas de 0 a 3. As paletas e suas cores associadas são mostradas na Tabela 17.2. A função **_selectpalette()** permite mudar as paletas.

No modo de 16 cores da EGA/VGA, uma paleta consiste em 16 cores selecionadas entre 64 possíveis. Os valores padrão são mostrados a seguir:

Cor	Valor
Preto	0
Azul	1
Verde	2

Cor	Valor
Ciano	3
Vermelho	4
Magenta	5
Marrom	6
Cinza-claro	7
Cinza-escuro	8
Azul-claro	9
Verde-claro	10
Ciano-claro	11
Vermelho-claro	12
Magenta-claro	13
Amarelo	14
Branco	15

Tabela 17.2 As paletas e cores do modo de vídeo 4.

Paleta	Número da cor		
	1	2	3
0	Verde	Vermelho	Marrom
1	Ciano	Magenta	Branco
2	Verde-claro	Vermelho-claro	Amarelo
3	Ciano-claro	Magenta-claro	Branco

Para mudar uma paleta na EGA/VGA, utilize **_remapallpalette()**, que define as cores que você selecionou na paleta.

#include <graph.h>
short __far _arc(short x1, short y1, short x2, short y2, short x3, short y3, short x4, short y4);

A função **_arc()** desenha um arco cujo centro é o centro do retângulo de demarcação definido por *x1,y1* e *x2,y2*. (O retângulo não é mostrado.) O arco começa no ponto definido por *x3,y3* e termina em *x4,y4*. (Este processo é exibido na Figura 17.1.) O arco é mostrado na cor atual de desenho.

A função **_arc()** devolve verdadeiro se bem-sucedida; devolve zero caso ocorra um erro.

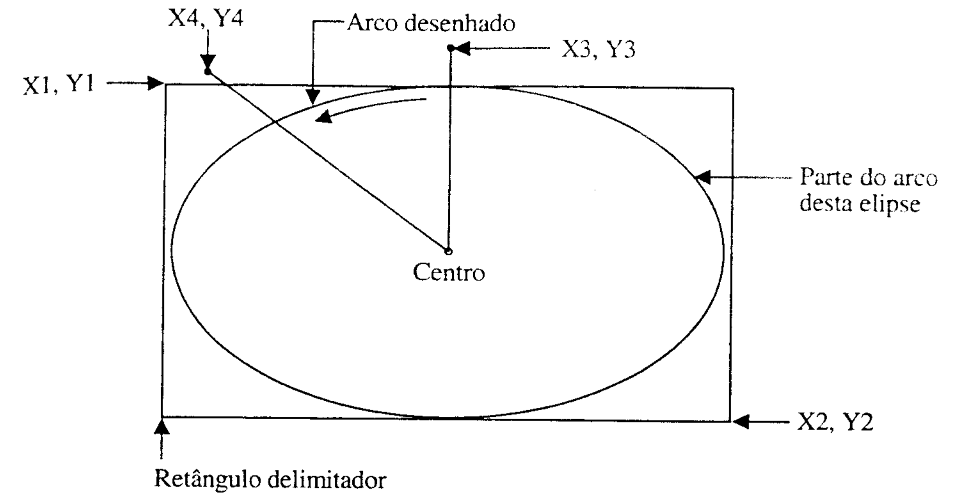


Figura 17.1 O funcionamento da função arc().

Exemplo

Este programa mostra um arco:

```
#include <graph.h>
#include <conio.h>

void main(void)
{
  _setvideomode(_VRES16COLOR);

  _arc(100, 100, 200, 200, 100, 100, 200, 200);
  getche();

  _setvideomode(_DEFAULTMODE);
}
```

Funções Relacionadas

_ellipse(), _lineto(), _rectangle()

#include <graph.h>
void __far _clearscreen(short region);

A função **_clearscreen()** limpa a região especificada da tela usando a cor atual de fundo. Essa função opera nos modos gráficos e de texto. Os valores de *region* são **_GCLEARSCREEN**, que limpa a tela inteira, **_GVIEWPORT**, que limpa uma janela (*viewport*) gráfica, e **_GWINDOW**, que limpa uma janela de texto. Essas macros estão definidas pelo Microsoft em GRAPH.H.

Exemplo

Este programa limpa a tela:

```
#include <graph.h>

void main(void)
{
  _clearscreen(_GCLEARSCREEN);
}
```

Funções Relacionadas

_settextwindow(), _setviewport()

#include <graph.h>
short __far _ellipse(short fill, short x1, short y1, short x2, short y2);

A função **_ellipse()** desenha uma elipse delimitada pelo retângulo definido por *x1,y1* e *x2,y2*, utilizando a cor atual de desenho. (O retângulo não é mostrado.) Se *fill* tem o valor **_GFILLINTERIOR**, a elipse é preenchida utilizando-se a cor e estilo de preenchimento atuais. Se *fill* tem o valor **_GBORDER**, a elipse não é preenchida. Essas macros estão definidas pela Microsoft em GRAPH.H.

Se a elipse pode ser desenhada, **_ellipse()** devolve verdadeiro. Caso contrário, devolve zero.

Exemplo

O programa seguinte desenha uma elipse.

```
#include <graph.h>
#include <conio.h>

void main(void)
```



```
{
  _setvideomode(_VRES16COLOR);
  _setcolor(3);
  _ellipse(_GBORDER, 100, 100, 300, 200);
  getche();

  _setvideomode(_DEFAULTMODE);
}
```

Funções Relacionadas

_arc(), _lineto(), _rectangle()

#include <graph.h>
short __far _floodfill(short x, short y, short color);

A função **_floodfill()** preenche uma região com a cor e o estilo de preenchimento atuais. A região preenchida deve estar completamente fechada pela cor especificada em *color* (Isto é, *color* especifica a cor da área que delimita a região). O ponto especificado por *x,y* deve estar contido na região a ser preenchida.

A função **_floodfill()** devolve verdadeiro caso seja bem-sucedida e zero se ocorre um erro.

Exemplo

Este programa desenha uma elipse e, em seguida, utiliza **_floodfill()** para preenchê-la.

```
#include <graph.h>
#include <conio.h>

void main(void)
{
   short color;

  _setvideomode(_VRES16COLOR);
  color = _getcolor();
  _ellipse(_GBORDER, 100, 100, 300, 200);
  _setcolor(2);
  _floodfill(150, 150, color);
  getche();

  _setvideomode(_DEFAULTMODE);
}
```

Função Relacionada

_setfillmask()

#include <graph.h>
long _ _far _getbkcolor(void);

A função **_getbkcolor()** devolve o valor da cor de fundo atual.

Exemplo

Este programa mostra a cor de fundo padrão.

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

void main(void)
{
  _setvideomode(_VRES16COLOR);

  printf("A cor de fundo é %ld.", _getbkcolor());
  getche();

  _setvideomode(_DEFAULTMODE);
}
```

Função Relacionada

_setbkcolor()

#include <graph.h>
short _ _far _getcolor(void);

A função **_getcolor()** devolve o valor da cor de desenho atual. Por padrão, a cor de desenho atual é o maior valor permitido pelo modo de vídeo atualmente em uso. Você pode usar esse fato para determinar a gama de cores válida (começando de zero) para um dado modo.

Exemplo

Este programa mostra o valor máximo para cores de desenho:

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



```
void main(void)
{
  short color;

  _setvideomode(_VRES16COLOR);

  color = _getcolor();
  printf("A cor é %hd.", color);
  getche();

  _setvideomode(_DEFAULTMODE);
}
```

Função Relacionada

_setcolor()

#include <graph.h>
struct _xycoord _ _far _getcurrentposition(void);

A função **_getcurrentposition()** devolve as coordenadas *x,y* da posição gráfica atual. A posição gráfica atual é o ponto no qual o próximo evento de saída gráfica começará. A posição gráfica atual não está de forma alguma relacionada com a posição de texto atual.

O Microsoft define a estrutura **_xycoord** desta forma:

```
struct _xycoord {
  short xcoord;
  short ycoord;
};
```

Exemplo

Este fragmento mostra a posição gráfica atual:

```
struct _xycoord xy;
.
.
.
xy = _getcurrentposition();
printf("X,Y atuais são %d %d.", xy.xcoord, xy.ycoord);
```

Funções Relacionadas

_gettextposition(), _moveto()

#include <graph.h>
unsigned char __far *__far_getfillmask(unsigned char __far *buf);

A função **_getfillmask()** copia o padrão de preenchimento atual no buffer apontado por *buf*. O buffer deve ter 8 bytes de extensão.

O padrão de preenchimento define a maneira como um objeto será preenchido por **_floodfill()** ou uma das outras funções que podem preencher um objeto. A máscara é tratada como uma matriz de 8 bytes por 8 bits. Essa matriz é, então, mapeada repetidamente na região a ser preenchida. Quando um bit é ativado, o pixel correspondente é colocado na cor de preenchimento atual. Se o bit está desligado, o pixel não é modificado.

A função **_getfillmask()** devolve **NULL** se nenhuma máscara está disponível.

Exemplo

O programa seguinte salva a máscara de preenchimento atual, gera uma nova randomicamente, preenche uma elipse usando a nova máscara e, finalmente, repõe a máscara de preenchimento no seu valor anterior.

```
#include <graph.h>
#include <conio.h>
#include <stdlib.h>

void main(void)
{
  short color, i;
  unsigned char oldmask[8];
  unsigned char newmask[8];

  /* obtém alguns valores randômicos para newmask */
  for(i=0; i<8; i++) newmask[i] = rand()%255;

  _setvideomode(_VRES16COLOR);

  color = _getcolor();
```



```
  _ellipse(_GBORDER, 100, 100, 300, 200);
  _setcolor(2);
  _getfillmask(oldmask);
  _setfillmask(newmask);
  _floodfill(150, 150, color);
  _setfillmask(oldmask);
  getche();

  _setvideomode(_DEFAULTMODE);
}
```

Função Relacionada

_setfillmask()

#include <graph.h>
void __far _getimage(short x1, short y1, short x2, short y2, char __huge *buf);

A função **_getimage()** copia o conteúdo do retângulo definido por *x1,y1* e *x2,y2* no buffer apontado por *buf*. Para determinar o tamanho em bytes que o buffer deve ter, deve ser utilizada a função **_imagesize()**.

Exemplo

Este programa copia a imagem de uma elipse de uma parte para outra da tela:

```
#include <stdio.h>
#include <graph.h>
#include <conio.h>
#include <stdlib.h>

void main(void)
{
  long size;
  char *buf;

  _setvideomode(_VRES16COLOR);

  size = _imagesize(100, 100, 300, 200);
  buf = malloc((size_t) size);
  if(!buf) {
    print("Erro de alocação.\n");
```

```
    exit(1);
  }

  _ellipse(_GBORDER, 100, 100, 300, 200);
  _getimage(100, 100, 300, 200, buf);
  _putimage(0, 0, buf, _GPSET);
  getche();

  _setvideomode(_DEFAULTMODE);
}
```

Funções Relacionadas

_imagesize(), _putimage()

#include <graph.h>
unsigned short __far _getlinestyle(void);

A função **_getlinestyle()** devolve o estilo de linha atual. Essa máscara determina como as linhas aparecerão. A máscara de estilo de linha atual tem 16 bits de comprimento. Se um bit é ativado, o pixel correspondente a esse bit é colocado na cor de desenho atual. Se o bit está desligado, o pixel não é modificado.

Exemplo

O programa seguinte salva o estilo de linha atual, usa um novo estilo para desenhar um retângulo, restaura o estilo de linha antigo e desenha outro retângulo.

```
#include <graph.h>
#include <conio.h>

void main(void)
{
  short oldstyle;

  _setvideomode(_VRES16COLOR);

  oldstyle = _getlinestyle();
  _moveto(0, 0);

  /* estilo de linha padrão */
  _lineto(100, 100);

  /* novo estilo de linha */
```



```
  _setlinestyle(12345);
  _rectangle(_GBORDER, 100, 100, 200, 200);

  /* estilo antigo restaurado */
  _setlinestyle(oldstyle);
  _rectangle(_GBORDER, 200, 200, 300, 300);
  getche();

  _setvideomode(_DEFAULTMODE);
}
```

Função Relacionada

_setlinestyle()

#include <graph.h>
short __far _getpixel(short x, short y);

A função **_getpixel()** devolve a cor do pixel especificado por *x,y*. Se o valor de *x* ou *y* for inválido, **_getpixel()** devolverá -1.

Exemplo

Este fragmento de código escreve a cor atual do pixel em 0,0:

```
printf("%d", _getpixel(0, 0));
```

Função Relacionada

_setpixel()

#include <graph.h>
short __far _gettextcolor(void);

A função **_gettextcolor()** devolve a cor de texto atual, isto é, a cor em que as saídas de texto serão escritas.

Exemplo

O fragmento de código seguinte mostra um texto na tela na cor padrão e em uma nova cor. A cor de texto é, então, recolocada na cor padrão.

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

```
void main(void)
{
  int color;

  _setvideomode(_VRES16COLOR);

  color = _gettextcolor();
  printf("A cor de texto padrão é %d.", color);
  _settextcolor(3);
  _settextposition(10, 1);
  _outtext("Isto está em uma cor diferente.\n");
  getche();

  _setvideomode(_DEFAULTMODE);
}
```

Função Relacionada

_settextcolor()

#include <graph.h>
struct _rccoord __far _gettextposition(void);

A função **_gettextposition()** devolve a posição de texto atual. A posição de texto atual especifica as coordenadas em que a próxima saída de texto começará. A posição de texto não está relacionada à posição gráfica atual.

A função **_gettextposition()** devolve as coordenadas de linha e coluna em uma estrutura do tipo **_rccoord**, definida pelo Microsoft como mostrado aqui:

```
struct _rccoord {
  short row;
  short col;
};
```

Exemplo

Este fragmento mostra a posição de texto atual.

```
struct _rccoord loc;

loc = _gettextposition();

printf("%d,%d", loc.row, loc.col);
```


Função Relacionada

_settextposition()

#include <graph.h>
struct _videoconfig __far *__far _getvideoconfig (struct _videoconfig __far *buf);

A função **_getvideoconfig()** copia a configuração atual de vídeo do sistema na estrutura apontada por *buf*. A informação sobre a configuração inclui a dimensão da tela em pixels, o número de colunas e linhas de texto, o número de cores diferentes, os bits por pixel e o número de páginas de vídeo.

A Microsoft define a estrutura **_videoconfig** como segue:

```
struct _videoconfig {
  short numxpixels;      /* número de pixels horizontalmente */
  short numypixels;      /* número de pixels verticalmente */
  short numtextcols;     /* número de colunas de texto */
  short numtextrows;     /* número de linhas de texto */
  short numcolors;       /* número de cores */
  short bitsperpixel;    /* número de bits em um pixel */
  short numvideopages;   /* número de páginas de vídeo */
  short mode;            /* modo de vídeo */
  short adapter;         /* placa de vídeo */
  short monitor;         /* monitor de vídeo */
  short memory;          /* kilobytes de memória de vídeo */
}
```

Exemplo

Este fragmento mostra o número de colunas de texto disponível no modo de vídeo atual:

```
struct _videoconfig c;

_getvideoconfig(&c);

printf("Colunas de texto: %d.", c.numtextcols);
```

Função Relacionada

_setvideomode()

#include <graph.h>
long __far _imagesize(short x1, short y1, short x2, short y2);

A função **_imagesize()** devolve em bytes o tamanho de memória necessário para conter a região retangular da tela definida por *x1,y1* e *x2,y2*, relativo ao modo de vídeo atual. É utilizada principalmente em união com **_getimage()**.

Exemplo

O fragmento de código seguinte devolve a quantidade de bytes da memória que é necessária para armazenar uma imagem.

```
size = _imagesize(0, 0, 10, 10);
```

Funções Relacionadas

_getimage(), _putimage()

#include <graph.h>
short __far _lineto(short x, short y);

A função **_lineto()** desenha uma linha na cor de desenho atual da posição gráfica atual até aquela especificada por *x,y*. A posição gráfica atual é, então, colocada em *x,y*.

A função **_lineto()** devolve verdadeiro caso seja bem-sucedida. Se as coordenadas especificadas por *x,y* são inválidas para o modo de vídeo atual, **_lineto()** devolve zero.

Exemplo

Este programa mostra uma linha diagonal começando no canto superior esquerdo da tela:

```
#include <graph.h>
#include <conio.h>

void main(void)
{
  _setvideomode(_VRES16COLOR);
  _moveto(0, 0);
  _setcolor(3);
  _lineto(600,400);
  getche();

  _setvideomode(_DEFAULTMODE);
}
```


Funções Relacionadas

_ellipse(), _moveto(), _rectangle()

#include <graph.h>
struct _xycoord __far _moveto(short x, short y);

A função **_moveto()** modifica a posição gráfica atual para aquela especificada por *x,y*. Ela não afeta a posição de texto atual. A função **_moveto()** devolve uma estrutura do tipo **_xycoord**, que contém as coordenadas da posição gráfica anterior.

A estrutura **_xycoord** é definida pela Microsoft como:

```
struct _xycoord {
  short xcoord;
  short ycoord;
};
```

Exemplo

Este fragmento retorna o canto superior esquerdo da tela à posição gráfica atual:

```
_moveto(0, 0);
```

Funções Relacionadas

_lineto(), _settextposition()

#include <graph.h>
void __far _outtext(const char __far *str);

A função **_outtext()** apresenta a string apontada por *str* na tela, a partir da posição de texto atual, nas cores de texto de frente e de fundo atuais. Ela também reconhece os limites do viewport e da janela.

Exemplo

O fragmento seguinte apresenta **Ola** na tela.

```
_outtext("Ola");
```

Funções Relacionadas

_gettextposition(), _settextposition()

#include <graph.h>
short __far _pie(short fill, short x1, short y1, short x2, short y2, short x3, short y3, short x4, short y4);

A função **_pie()** desenha uma fatia de um gráfico de *pizza*. A fatia é uma porção de uma elipse definida pelo seu retângulo delimitador cujo canto superior esquerdo está em x1, y1, cujo canto inferior direito está em x2, y2. O centro da pizza está no centro do retângulo delimitador. O retângulo não é exibido. A fatia começa no ponto em que uma linha desenhada a partir do centro da elipse até x3, y3 *intersecciona* a elipse e termina no ponto em que uma linha desenhada do centro da elipse até x4, y4 intersecciona a elipse. (Este processo é exibido na Figura 17.2.) Ele é mostrado na cor de desenho atual. Se *fill* tem o valor **_GBORDER**, o setor não é preenchido. Se *fill* tem o valor **_GFILLINTERIOR**, ele é preenchido usando a cor e o estilo de preenchimento atuais. (Essas macros estão definidas pelo Microsoft em GRAPH.H.)

Exemplo

Este programa mostra um setor:

```
#include <graph.h>
#include <conio.h>

void main(void)
{
  _setvideomode(_VRES16COLOR);

  _pie(_GBORDER, 100, 100, 300, 300, 300, 10, 100, 300);
  getche();

  _setvideomode(_DEFAULTMODE);
}
```

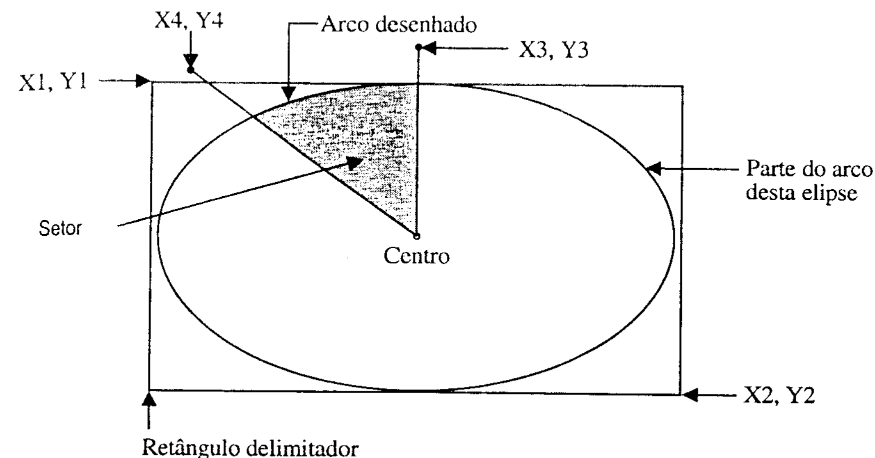



Figura 17.2 O funcionamento da função pie().

Funções Relacionadas

_ellipse(), _rectangle()

#include <graph.h>
void __far _putimage(short x, short y, const char __huge *buf, short how);

A função **_putimage()** copia uma imagem para a tela. Com freqüência esta imagem foi gravada anteriormente por **_getimage()** na tela. A memória que contém a imagem é apontada por *buf*. A imagem é copiada na tela com o seu canto superior esquerdo posicionado nas coordenadas *x,y*.

how determina como a imagem será copiada na tela. O Microsoft define, em GRAPH.H, os cinco valores que *how* pode ter. Caso seja **_GPSET**, a imagem é copiada na tela, sobrepondo qualquer conteúdo anterior. Caso *how* seja **_GPRESET**, a imagem é copiada na tela em vídeo inverso. Caso *how* seja **_GXOR**, é realizada uma operação XOR entre cada pixel da imagem e o conteúdo atual da tela. Caso seja **_GOR**, é realizada uma operação OR entre cada pixel da imagem e o conteúdo atual da tela. Finalmente, se *how* for **_GAND**, é realizada uma operação AND entre cada pixel da imagem e o conteúdo da tela.

Exemplo

Este programa copia a imagem de uma elipse de uma parte a outra da tela:

```
#include <stdio.h>
#include <graph.h>
#include <conio.h>
#include <stdlib.h>

void main(void)
{
  long size;
  char *buf;

  _setvideomode(_VRES16COLOR);

  size = _imagesize(100, 100, 300, 200);
  buf = malloc((size_t) size);
  if(!buf) {
    print("erro de alocação\n");
    exit(1);
  }

  _ellipse(_GBORDER, 100, 100, 300, 200);
  _getimage(100, 100, 300, 200, buf);
  _putimage(0, 0, buf, _GPSET);
  getche();

  _setvideomode(_DEFAULTMODE);
}
```

Função Relacionada

_getimage()

#include <graph.h> short _ _far _rectangle(short fill, short x1, short y1, short x2, short y2);

A função **_rectangle()** desenha um retângulo na cor de desenho atual. O canto superior esquerdo do retângulo é especificado por *x1,y1* e o canto inferior direito, por *x2,y2*.

Se *fill* tiver o valor **_GFILLINTERIOR**, o retângulo será preenchido usando a cor atual. Se *fill* tiver o valor **_GBORDER**, o retângulo não será preenchido. (Essas macros são definidas pelo Microsoft em GRAPH.H.)

A função **_rectangle()** devolve verdadeiro se as coordenadas estão dentro da faixa e zero, caso contrário.

Exemplo

Este fragmento de código desenha um retângulo na tela:

```
#include <graph.h>
#include <conio.h>

void main(void)
{
  _setvideomode(_VRES16COLOR);
  _rectangle(_GBORDER, 0, 0, 100, 100);
  getche();
  _setvideomode(_DEFAULTMODE);
}
```

Funções Relacionadas

_ellipse(), _lineto()

#include <graph.h> short _ _far _remapallpalette(long _ _far *colors); long _ _far _remappallette(short index, short newcolor);

A função **_remapallpalette()** troca os valores das cores suportadas pelo modo de vídeo atual pelos valores especificados na matriz apontada por *colors*, que deve conter tantas cores quantas existem na paleta atual. A função **_remappalette()** troca o valor de uma cor, especificada por *index*, por aquela especificada por *newcolor*.

As duas funções exigem uma placa de vídeo EGA ou melhor.

Quando bem-sucedida, **_remappalette()** devolve a cor anteriormente associada a *oldcolor*. Quando bem-sucedida, **_remapallpalette()** devolve um valor diferente de zero. Em caso de falha, as duas funções devolvem -1.

Exemplo

O fragmento seguinte define a cor 4 na cor 3:

```
_remappalette(4, 3);
```

Exemplo

Este programa cria uma nova máscara de preenchimento gerada randomicamente e usa-a para preencher uma elipse:

```
#include <graph.h>
#include <conio.h>
#include <stdlib.h>

void main(void)
{
  short  i;
  unsigned char newmask[8];

  /* obtém alguns valores randômicos para newmask */
  for(i=0; i<8; i++) newmask[i] = rand()%255;

  _setvideomode(_VRES16COLOR);

  _setfillmask(newmask);
  _ellipse(_GFILLINTERIOR, 100, 100, 300, 200);
  getche();

  _setvideomode(_DEFAULTMODE);
}
```

Função Relacionada

_getfillmask()

#include <graph.h>
void __far _setlinestyle(unsigned short mask);

A função **_setlinestyle()** estabelece o estilo de linha atual conforme especificado por *mask*. Uma máscara de estilo de linha tem 16 bits de extensão. Se um bit está ativado, o pixel correspondente àquele bit é colocado na cor de desenho atual. Se o bit está desligado, o pixel é deixado inalterado.

Exemplo

Este programa salva o estilo de linha atual, usa um novo estilo para desenhar um retângulo, restaura o estilo de linha antigo e desenha outro retângulo:


```
#include <graph.h>
#include <conio.h>

void main(void)
{
  short oldstyle;

  _setvideomode(_VRES16COLOR);

  oldstyle = _getlinestyle();
  _moveto(0, 0);

  /* estilo de linha padrão */
  _lineto(100, 100);

  /* novo estilo de linha */
  _setlinestyle(12345);
  _rectangle(_GBORDER, 100, 100, 200, 200);

  /* estilo anterior restaurado */
  _setlinestyle(oldstyle);
  _rectangle(_GBORDER, 200, 200, 300, 300);
  getche();

  _setvideomode(_DEFAULTMODE);
}
```

Funções Relacionadas

_getlinestyle(), _setfillmask()

#include <graph.h>
short __far _setpixel(short x, short y);

A função **_setpixel()** modifica o pixel especificado por *x,y* de acordo com a cor de desenho atual. Ela devolve a cor anterior. Se uma coordenada especificada está fora dos limites, **_setpixel()** devolve -1.

Exemplo

Este fragmento altera o pixel na posição 10,20 para a cor de desenho atual:

```
_setpixel(10, 20);
```

Função Relacionada

_getpixel()

#include <graph.h>
short __far _settextcolor(short color);

A função **_settextcolor()** troca a cor de texto atual por aquela especificada por *color*. Ela devolve a cor de texto anterior.

Exemplo

Este programa mostra **Isto está em vermelho** em vermelho:

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

void main(void)
{
  _setvideomode(_VRES16COLOR);

  _settextcolor(4);
  _outtext("Isto está em vermelho\n");
  _getche();

  _setvideomode(_DEFAULTMODE);
}
```

Funções Relacionadas

_gettextcolor(), _setcolor()

#include <graph.h>
struct _rccoord __far _settextposition(short row, short col);

A função **_settextposition()** estabelece a posição de texto atual conforme especificado por *row* e *col*. A posição de texto atual especifica as coordenadas em que a próxima saída de texto começará. A posição de texto não está relacionada com a posição gráfica atual.

A função **_settextposition()** devolve a posição de texto anterior, nas coordenadas de linha e coluna, em uma estrutura do tipo **_rccoord**, definida pelo Microsoft em GRAPH.H como segue:

```
struct _rccoord {
  short row;
  short col;
};
```

Exemplo

Este fragmento estabelece a posição de texto atual para a linha 10, coluna 40:

```
_settextposition(10, 40);
```

Função Relacionada

_gettextposition()

#include <graph.h>
void __far _settextwindow(short row1, short col1, short row2, short col2);

A função **_settextwindow()** define uma janela de texto especificada por *row1*, *col1* e *row2*, *col2* — os cantos superior esquerdo e inferior direito da janela. O texto escrito na janela com **_outtext()** aparecerá relativamente à janela e não à tela. Isso significa que o texto rolará quando for escrito até o fundo da janela. Além disso, o texto passará para a próxima linha quando tentar ultrapassar a borda direita da janela. O resto da janela fica intacto.

As funções padrões de saída para o console ignoram janelas de texto.

Exemplo

Este programa cria uma pequena janela de texto e escreve sua saída nela. O texto mudará de linha e rolará dentro da janela, mas o resto da tela permanecerá intacto.

```
#include <graph.h>
#include <conio.h>

void main(void)
{
  int i;

  _settextwindow(1, 1, 5, 40);
```

```
    for(i=0; i<100; i++)
      _outtext("Isto é um teste.");
  }
```

Funções Relacionadas

_outtext(), _setviewport()

#include <graph.h>
short _ _far _setvideomode(short mode);

A função **_setvideomode()** ativa o modo de vídeo especificado por *mode*. Os modos válidos são mostrados a seguir. (As macros são definidas pelo Microsoft em GRAPH.H.)

_TEXTBW40	_HERCMONO	_VRES16COLOR
_TEXTC40	_TEXTMONO	_MRES256COLOR
_TEXTBW80	_MRES16COLOR	_DEFAULTMODE
_TEXTC80	_HRES16COLOR	_MAXRESMODE
_MRES4COLOR	_ERESNOCOLOR	_MAXCOLORMODE
_MRESNOCOLOR	_ERESCOLOR	
_HRESBW	_VRES2COLOR	

Você pode restaurar o modo de vídeo padrão chamando **_setvideomode()** com **_DEFAULTMODE.**

A função **_setvideomode()** devolve verdadeiro se a mudança de modo foi bem-sucedida. Se o modo especificado não for suportado pelo hardware do sistema, **_setvideomode()** devolverá zero.

Exemplo

O fragmento seguinte estabelece o modo de vídeo para texto colorido de 80 colunas:

```
_setvideomode(_TEXTOC80);
```

Função Relacionada

_getvideoconfig()

#include <graph.h>
void _ _far _setviewport(short x1, short y1, short x2, short y2);

A função **_setviewport()** cria uma janela gráfica, denominada *viewport*, cujo canto superior esquerdo é especificado por *x1,y1* e o canto inferior direito, por *x2,y2*. Uma vez que "viewport" tenha sido definido, as funções gráficas operam relativamente a ele em lugar da tela. A saída é automaticamente cortada (*clipped*) nas bordas do viewport.

Exemplo

Este programa cria uma janela gráfica e desenha uma linha dentro dela.

```
#include <graph.h>
#include <conio.h>

void main(void)
{
  _setvideomode(_VRES16COLOR);

  _setviewport(200, 200, 300, 300);
  _moveto(0, 0); _setcolor(3);
  _lineto(600, 400); /* esta linha será cortada */
  getche()

  _setvideomode(_DEFAULTMODE);
}
```

Função Relacionada

_settextwindow()

MAKRON Books

18

Funções Miscelâneas

Este capítulo discute todas as funções do padrão ANSI que não se encaixam facilmente em alguma outra categoria. Elas incluem diversas conversões, suporte para argumentos de tamanho variável, ordenação e outras funções.

Muitas dessas funções exigem o uso do cabeçalho STDLIB.H. Esse cabeçalho define dois tipos, **div_t** e **ldiv_t**, que são os tipos de valores devolvidos por **div()** e **ldiv()**, respectivamente. STDLIB.H também define os tipos **size_t**, que é o valor sem sinal devolvido por **sizeof**, e **wchar_t**, que é o tipo de dados dos caracteres largos (16 bits) usados por um conjunto estendido de caracteres. Além disso, o cabeçalho define estas macros:

NULL	Um ponteiro nulo.
RAND_MAX	O máximo valor que pode ser devolvido pela função **rand()**.
EXIT_FAILURE	O valor devolvido para o processo chamador com a terminação mal-sucedida de um programa.
EXIT_SUCCESS	O valor devolvido para o processo chamador caso a terminação do programa tenha sido bem-sucedida.
MB_CUR_MAX	O número máximo de bytes de um caractere multibyte.

Se alguma função requer um arquivo de cabeçalho diferente, ele será discutido na descrição da função.

#include <stdlib.h>
void abort(void);

A função **abort()** provoca a conclusão imediata e anormal do programa. Geralmente, nenhum arquivo é fechado. Em ambientes que a suportam, **abort()** devolve um valor definido pela implementação ao processo chamador (normalmente o sistema operacional), indicando falha.

Exemplo

Este programa termina se o usuário digitar um **A**:

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

void main(void)
{
  for(;;)
  if(getchar()=='A') abort();
}
```

Funções Relacionadas

atexit(), exit()

#include <stdlib.h>
int abs(int num);

A função **abs()** devolve o valor absoluto do inteiro *num*.

Exemplo

Esta função converte os números digitados pelo usuário em seus valores absolutos:

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

get_abs(void)
{
  char num[80];

  gets(num);
  return abs(atoi(num));
}
```

Função Relacionada

labs()

#include <assert.h>
void assert(int exp);

A macro **assert()**, definida no seu cabeçalho ASSERT.H, escreve informação de erro em **stderr** e, então, aborta a execução do programa se a expressão *exp* vale zero. Caso contrário, **assert()** não faz nada. Embora a saída exata seja definida pela implementação, muitos compiladores usam uma mensagem semelhante à seguinte:

Assertion failed: <*expressão*>, file <*arquivo*>, line <*numlinha*>

A macro **assert()** geralmente é usada para ajudar a verificar se um programa está operando corretamente. A expressão é planejada de forma que ela chegue a verdadeiro somente quando não ocorre nenhum erro.

Você não precisa remover as sentenças **assert()** do código-fonte assim que o programa tenha sido depurado, porque, se a macro **NDEBUG** estiver definida (com qualquer valor) antes de incluir ASSERT.H, as macros **assert()** são ignoradas.

Exemplo

Este fragmento de código testa se os dados lidos de uma porta serial estão em ASCII (isto é, se ela não usa o oitavo bit):

```
.
.
.
ch = read_port();
assert(!(ch & 128)); /* verifica bit 7 */
.
.
.
```

Função Relacionada

abort()

#include <stdlib.h>
int atexit(void (*func)(void));

A função **atexit()** registra a função apontada por *func* como uma função a ser utilizada na terminação normal do programa. Isto é, ao final da execução de um programa, a função especificada será chamada.

A função **atexit()** devolve zero se a função foi registrada, com sucesso, como uma função de terminação; caso contrário, ela devolve um valor diferente de zero.

O padrão C ANSI especifica que pelo menos 32 funções de terminação podem ser estabelecidas e que elas são chamadas na ordem inversa do seu estabelecimento. Em outras palavras, o processo de registro constrói uma pilha de funções.

Exemplo

Este programa escreve **alô aqui** na tela quando termina.

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

void done(void);

void main(void)
{

  if(atexit(done)) printf("Erro em atexit().");
}

void done(void)
{
  printf("alô aqui");
}
```

Funções Relacionadas

abort(), exit()

#include <stdlib.h>
double atof(const char *str);

A função **atof()** converte a string apontada por *str* em um valor **double** e retorna o resultado. A string deve conter um número em ponto flutuante válido. Caso contrário, o valor devolvido é indefinido.

O número pode ser terminado por qualquer caractere que não possa fazer parte de um número em ponto flutuante válido. Isso inclui espaço em branco, pontuação diferente do ponto e caracteres diferentes de "E" ou "e". Isso significa que, se **atof()** for chamada com 100.00HELLO", o valor 100.00 será devolvido e o resto da string será ignorado.

Exemplo

Este programa lê dois números em ponto flutuante e mostra sua soma:

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

void main(void)
{
  char num1[80], num2[80];

  printf("digite o primeiro: ");
  gets(num1);
  printf("digite o segundo: ");
  gets(num2);
  printf("A soma é: %lf.", atof(num1) + atof(num2));
}
```

Funções Relacionadas

atoi(), atol()

#include <stdlib.h>
int atoi(const char *str);

A função **atoi()** converte a string apontada por *str* em um valor inteiro e retorna o resultado. A string deve conter um inteiro válido. Caso contrário, o valor devolvido é indefinido; no entanto, a maioria das implementações retornam zero.

O número pode ser terminado por qualquer caractere que não pode fazer parte de um inteiro. Isso inclui espaço em branco, pontuação e outros caracteres que não sejam dígitos. Por exemplo, se **atoi()** for chamada com "123.23", o valor inteiro 123 é devolvido e 0.23 é ignorado.

Exemplo

Este programa lê dois inteiros e mostra sua soma.

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

void main(void)
{
  char num1[80], num2[80];

  printf("digite o primeiro: ");
  gets(num1);
  printf("digite o segundo: ");
  gets(num2);
  printf("A soma é: %d.", atoi(num1) + atoi(num2));
}
```

Funções Relacionadas

atof(), atol()

#include <stdlib.h>
int atol(const char *str);

A função **atol()** converte a string apontada por *str* em um valor **long int**. A string deve conter um inteiro longo válido. Caso contrário, o valor devolvido é indefinido; no entanto, a maioria das implementações retorna zero.

O número pode ser terminado por qualquer caractere que não pode fazer parte de um inteiro. Isso inclui espaço em branco, pontuação e outros caracteres que não sejam dígitos. Por exemplo, se **atol()** for chamada com "123.23", por exemplo, o valor inteiro 123 é devolvido e 0.23 é ignorado.

Exemplo

Este programa lê dois inteiros longos e mostra sua soma:

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

```
void main(void)
{
  char num1[80], num2[80];

  printf("digite o primeiro: ");
  gets(num1);
  printf("digite o segundo: ");
  gets(num2);
  printf("A soma é: %ld.", atol(num1) + atol(num2));
}
```

Funções Relacionadas

atof(), atoi()

#include <stdlib.h> void *bsearch(const void *key, const void *buf, size_t num, size_t size, int (*compare)(const void *, const void *));

A função **bsearch()** realiza uma pesquisa binária na matriz ordenada apontada por *buf*, devolvendo um ponteiro para o primeiro membro que coincide com a chave apontada por *key*. O número de elementos da matriz é especificado por *num* e o tamanho (em bytes) de cada elemento é descrito por *size*.

A função apontada por *compare* compara um elemento da matriz com a chave. A função deve ter o formato

int *compare* (const void **arg1*, const void **arg2*);

A função pode ter o nome que você quiser. No entanto, ela deve devolver os seguintes valores:

Se *arg1* é menor que *arg2*, devolve menor que zero.
Se *arg1* é igual a *arg2*, devolve zero.
Se *arg1* é maior que *arg2*, devolve maior que zero.

A matriz deve estar em ordem ascendente, com o endereço mais baixo contendo o menor elemento. Se a matriz não contém a chave, é devolvido um ponteiro nulo.

Exemplo

O programa seguinte lê caracteres digitados no teclado e determina se eles pertencem ao alfabeto.


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

char *alpha =  "abcdefghijklmnopqrstuvwxyz";

int comp(const void *ch, const void *s);

void main(void)
{
  char ch;
  char *p;

  do {
    printf("digite um caractere: ");
    ch = getche();
    ch = tolower(ch);
    p = (char *) bsearch(&ch, alpha, 26, 1, comp);
    if(p) printf("está no alfabeto\n");
    else printf("não está no alfabeto\n");
  } while(p);
}

/* Compara dois caracteres. */
comp(const char *ch, const char *s)
{
  return *(char *)ch -*(char *)s;
}
```

Função Relacionada

qsort()

#include <stdlib.h> div_t div(int numerator, int denominator);

A função **div()** devolve o quociente e o resto da operação *numerador/denominador* em uma estrutura do tipo **div_t**.

O tipo de estrutura **div_t** é definido em STDLIB.H e tem pelo menos dois campos:

int quot; /* o quociente */

int rem; /* o resto */

Exemplo

Este programa mostra o quociente e o resto de 10/3:

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

void main(void)
{
  div_t n;

  n = div(10, 3);

  printf("O quociente e resto: %d %d.\n", n.quot, n.rem);
}
```

Função Relacionada

ldiv()

#include <stdlib.h>
void exit(int exit_code);

A função **exit()** provoca a terminação normal imediata de um programa.

O valor de *exit_code* é passado ao processo chamador, normalmente o sistema operacional, se o ambiente suporta-o. Por convenção, se o valor de *exit_code* é zero, uma terminação normal do programa é assumida. Um valor diferente de zero pode indicar um erro definido pela implementação. Em C também são definidos dois valores que podem ser usados como parâmetros para **exit()**: **EXIT_SUCCESS** e **EXIT_FAILURE**. Estes valores indicarão encerramento bem-sucedido e mal-sucedido, respectivamente, em todos os ambientes de execução.

Exemplo

Esta função executa uma seleção por menu para um programa de lista postal. Se **S** for selecionado, o programa terminará.

```
menu(void)
{
  char choice;

  do {
    printf("Inserir nomes (I)\n");
    printf("Apagar nomes  (A)\n");
```



```
    printf("Imprimir  (P)\n");
    printf("Sair      (S)\n");
    choice = getche ();
  } while(!strchr("IAPS", toupper(choice)));

  if(choice=="S") exit(0);

  return choice;
}
```

Funções Relacionadas

abort(), atexit()

#include <stdlib.h>
char *getenv(const char *name);

A função **getenv()** devolve um ponteiro para informações do ambiente associadas à string apontada por *name* na tabela de informações do ambiente definida pela implementação. A string devolvida nunca deve ser alterada pelo programa.

O ambiente de um programa pode incluir coisas como nomes de caminho e dispositivos on-line. A natureza exata desse dado é definida pela implementação. Consulte seu manual do usuário para detalhes.

Se é feita uma chamada a **getenv()** com um argumento que não coincide com nenhum dado do ambiente, um ponteiro nulo é devolvido.

Exemplo

Assumindo que um compilador específico mantém informações do ambiente sobre os dispositivos conectados ao sistema, o fragmento de código seguinte devolve um ponteiro para a lista de dispositivos.

```
char *p
.
.
.
p = getenv("DEVICES");
.
.
.
```

Função Relacionada

system()

#include <stdlib.h>
char *itoa(int num, const char *str, int radix);

A função **itoa()** não é atualmente definida pelo padrão C ANSI, mas é encontrada em muitos compiladores.

A função **itoa()** converte o número inteiro *num* em sua string equivalente e coloca o resultado na string apontada por *str*. A base da string de saída é determinada por *radix*, que geralmente está entre 2 e 16.

A função **itoa()** devolve um ponteiro para *str*. Normalmente, não há nenhum valor de erro devolvido. Assegure-se de chamar **itoa()** com uma string de comprimento suficiente para conter o resultado convertido.

Exemplo

Este programa mostra o valor de 1423 em hexadecimal (58F):

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

void main(void)
{
  char p[20];

  itoa(1423, p, 16);

  printf(p);
}
```

Funções Relacionadas

atoi(), sscanf()

#include <stdlib.h>
long labs(long num);

A função **labs()** devolve o valor absoluto de *num*.

Exemplo

Esta função converte o número digitado no teclado em seu valor absoluto:

```
#include <stdlib.h>
```



```
long int get_labs()
{
  char num[80];

  gets(num);

  return labs(atol(num));
}
```

Função Relacionada

abs()

#include <stdlib.h>
ldiv_t ldiv(long numerator, long denominator);

A função **ldiv()** devolve o quociente e o resto da operação *numerador/denominador* em uma estrutura *ldiv_t*.

O tipo de estrutura **ldiv_t** é definido em STDLIB.H e tem pelo menos dois campos:

long quot; /* o quociente */

long rem; /* o resto */

Exemplo

Este programa mostra o quociente e o resto de 100000L/3L:

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

void main(void)
{
  ldiv_t n;
  n = ldiv(100000L, 3L);

  printf("quociente & resto: %ld %ld.\n", n.quot, n.rem);
}
```

Função Relacionada

div()

#include <setjmp.h>
void longjmp(jmp_buf envbuf, int status);

A função **longjmp()** faz com que a execução do programa continue no ponto da última chamada a **setjmp()**. Essas duas funções constituem a maneira de C fornecer um desvio entre funções. Observe que é exigido o cabeçalho SETJMP.H.

A função **longjmp()** repõe a pilha como descrito em *envbuf*, que deve ter sido atribuída por uma chamada anterior a **setjmp()**. Isso faz com que a execução do programa continue no comando seguinte à invocação de **setjmp()**. Ou seja, o computador é "levado" a pensar que nunca deixou a função que chamou de **setjmp()**. Com efeito, **longjmp()** "desvia-se", no tempo e no espaço (de memória), para um ponto anterior do programa sem precisar efetuar o processo normal de retorno de uma função.

O buffer *envbuf* é do tipo **jmp_buf**, que é definido no cabeçalho SETJMP.H. O buffer deve ter sido estabelecido por meio de uma chamada a **setjmp()** antes da chamada a **longjmp()**.

O valor de *status* torna-se o valor devolvido por **setjmp()** e pode ser examinado para determinar a origem do desvio. O único valor não permitido é zero.

A função **longjmp()** deve ser chamada antes que a função que chamou **setjmp()** retorne. Se isso não ocorrer, o resultado será tecnicamente indefinido. (Na verdade, é quase certo que ocorrerá um "crash".)

O uso mais comum de **longjmp()** é no retorno de um conjunto de rotinas profundamente aninhadas quando ocorre um erro.

Exemplo

Este programa escreve **1 2 3**:

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

jmp_buf ebuf;

void f2(void);

void main(void)
{
  char first=1;
  int i;
```



```
  printf("1 ");
  i = setjmp(ebuf);
  if(first) {
    first = ! first;
    f2();
    printf("Isto não será escrito.");
  }
  printf("%d", i);
}

void f2(void)
{
  printf("2 ");
  longjmp(ebuf, 3);
}
```

Função Relacionada

setjmp()

#include <stdlib.h>
char *ltoa(long num, const char *str, int radix);

A função **ltoa()** não é atualmente definida pelo padrão C ANSI, mas é encontrada em muitos compiladores.

A função **ltoa()** converte o número inteiro longo *num* em sua string equivalente e coloca o resultado na string apontada por *str*. A base da string de saída é determinada por *radix*, que geralmente está entre as faixas 2 e 16.

A função **ltoa()** devolve um ponteiro para *str*. Normalmente, não há nenhum valor de erro devolvido. Assegure-se de chamar **ltoa()** com uma string de comprimento suficiente para conter o resultado convertido.

Exemplo

Este programa mostra o valor de 1423 em hexadecimal (58F):

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

void main(void)
{
  char p[20];
```

```
    itoa(1423L, p, 16);

    printf(p);
}
```

Função Relacionada

itoa()

#include <stdlib.h> int mblen(const char *str, size_t size);

A função **mblen()** retorna o comprimento (em bytes) de um caractere multibyte apontado por *str*. Somente os primeiros *size* caracteres serão examinados. Ela retorna -1 em caso de erro.

Se *str* for nula, então **mblen()** retornará um valor diferente de zero se os caracteres multibyte dependem da distinção entre maiúsculas e minúsculas. Se este não for o caso, ela retornará zero.

Exemplo

Este comando exibe o comprimento do caractere multibyte apontado por **mb**.

```
printf("%d", mblen(mb, 2));
```

Funções Relacionadas

mbtowc(), wctomb()

#include <stdlib.h> size_t mbstowcs(wchar_t *out, const char *in, size_t size);

A função **mbstowcs()** converte a string multibyte apontada por *in* em uma string de caracteres largos e coloca o resultado na matriz apontada por *out*. O tipo **wchar_t** é definido em STDLIB.H. Somente *size* bytes serão armazenados em *out*.

A função **mbstowcs()** retorna o número de caracteres multibytes que foram convertidos. Se ocorrer um erro, a função retornará -1.

Exemplo

Este comando converte os 4 primeiros caracteres na string multibyte apontada por **mb** e coloca o resultado em **str**.

```
mbstowcs(str, mb, 4);
```


Funções Relacionadas

wcstombs(), mbtowc()

#include <stdlib.h> int mbtowc(wchar_t *out, const char *in, size_t size);

A função **mbtowc()** converte o caractere multibyte apontado por *in* em seu equivalente de caracteres largos e coloca o resultado na matriz apontada por *out*. O tipo **wchar_t** é definido em STDLIB.H. Somente *size* caracteres serão analisados.

Esta função retorna o número de bytes que são inseridos em *out*. Se ocorrer um erro, a função retornará -1. Se *in* for nulo, então **mbtowc()** retorna um valor diferente de zero se os caracteres multibyte dependem da distinção entre maiúsculas e minúsculas. Se este não for o caso, ela retornará zero.

Exemplo

Este comando converte o caractere multibyte em **mbstr** em seu caractere largo equivalente e coloca o resultado na matriz apontada por **widenorm**. (Somente os 2 primeiros caracteres de **mbstr** são examinados.)

```
mbtowc(widenorm, mbstr, 2);
```

Funções Relacionadas

mblen(), wctomb()

#include <stdlib.h> void qsort(void *buf, size_t num, size_t size, int (*compare)(const void *, const void *));

A função **qsort()** ordena a matriz apontada por *buf*, usando quicksort. O quicksort geralmente é considerado o melhor algoritmo de ordenação de uso geral. (Veja o Capítulo 19 para uma discussão completa de ordenação e busca em C.) Ao terminar, a matriz está ordenada. O número de elementos na matriz é especificado por *num* e o tamanho (em bytes) de cada elemento é descrito em *size*.

A função apontada por *compare* é usada para comparar um elemento da matriz com a chave. A forma de *compare* deve ser

int *compare* (const void **arg1*, const void *arg2*);

A função pode ter o nome que você quiser. No entanto, ela deve devolver os seguintes valores:

Se *arg1* é menor que *arg2*, devolve menor que zero.
Se *arg1* é igual a *arg2*, devolve zero.
Se *arg1* é maior que *arg2*, devolve maior que zero.

A matriz é classificada em ordem crescente, com o endereço mais baixo contendo o menor elemento.

Exemplo

Este programa ordena uma lista de inteiros e mostra o resultado:

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

int num[10] = {
  1, 3, 6, 5, 8, 7, 9, 6, 2, 0
};

int comp(const void *, const void *);

void main(void)
{
  int i;

  printf("matriz original: ");
  for(i=0; i<10; i++) printf("%d ", num[i]);

  qsort(num, 10, sizeof(int), comp);

  printf("matriz ordenada: ");
  for(i=0; i<10; i++) printf("%d ", num[i]);
}

/* compara os inteiros */
comp(const int *i, const int *j)
{
  return *(int *)i - *(int *)j;
}
```

Função Relacionada

bsearch()

#include <signal.h>
int raise(int signal);

A função **raise()** envia o sinal especificado por *signal* para o programa executor. Ela devolve zero caso seja bem-sucedida; caso contrário, devolve um valor diferente de zero. Ela usa o arquivo de cabeçalho SIGNAL.H. Os seguintes sinais padrões são definidos pela padrão C ANSI. (No entanto, uma implementação de C pode suportar sinais adicionais.)

Sinal	Significado
SIGABRT	Encerramento anormal do programa
SIGFPE	Erro matemático
SIGILL	Instrução ilegal
SIGINT	Exigência de interação
SIGSEGV	Acesso ilegal à memória
SIGTERM	Pedido de encerramento do programa

Exemplo

A função **raise()** é, de certa forma, específica da implementação. Consulte o manual do usuário do seu compilador para detalhes e exemplos.

Função Relacionada

signal()

#include <stdlib.h>
int rand(void);

A função **rand()** gera uma seqüência de números pseudo-randômicos. Cada vez que é chamada, é devolvido um inteiro entre zero e **RAND_MAX**.

Exemplo

O programa seguinte mostra dez números pseudo-randômicos.

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

void main(void)
{
  int i;
```

```
  for(i=0; i<10; i++)
    printf("%d ", rand());
}
```

Função Relacionada

srand()

#include <setjmp.h>
int setjmp(jmp_buf envbuf);

A função **setjmp()** salva o conteúdo da pilha do sistema, no buffer *envbuf*, para uso posterior por **longjmp()**. Ela utiliza o arquivo de cabeçalho SETJMP.H, que define o tipo **jmp-buf**.

A função **setjmp()** devolve zero quando invocada. Porém, um **longjmp()** passa um argumento para **setjmp()** quando é executada; esse valor (sempre diferente de zero) torna-se o valor de **setjmp()** após uma chamada a **longjmp()**. Veja **longjmp()** para informações adicionais.

Exemplo

Este programa escreve **1 2 3**:

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

jmp_buf ebuf;
void f2(void);

void main(void)
{
  char first=1;
  int i;

  printf("1 ");
  i = setjmp(ebuf);
  if(first) {
    first = !first;
    f2();
    printf("Isto não será escrito.");
  }
  printf("%d", i);
}

void f2(void)
{
  printf("2 ");
  longjmp(ebuf, 3);
}
```


Função Relacionada

longjmp()

#include <signal.h>
void (*signal(int sig, void (*func)(int))) (int);

A função **signal()** definirá a função *func*, para ser executada se o sinal especificado *signal* for recebido. Essa função é, de certa forma, específica da implementação. O valor para *func* deve ser uma das seguintes macros, definidas em SIGNAL.H, ou o endereço de uma função:

Macro	Significado
SIG_DFL	Usa o tratador do sinal padrão
SIG_IGN	Ignora o sinal

Se um endereço de uma função é usado, a função especificada é executada. Se uma função de tratamento de sinal não pode processar um sinal, ela deve retornar **SIG_ERR** (que é uma macro definida em SIGNAL.H).

signal() é usada freqüentemente para configurar tratadores de erros críticos e de control C.

Exemplo

Veja o manual do usuário de seu compilador para detalhes e exemplos relativos ao seu sistema.

Função Relacionada

raise()

#include <stdlib.h>
void srand(unsigned seed);

A função **srand()** estabelece um ponto de partida para a seqüência gerada por **rand()**, que devolve números pseudo-randômicos.

srand() é geralmente utilizada para permitir que programas usem seqüências diferentes de números pseudo-randômicos a cada execução, especificando diferentes pontos de partida. No entanto, você pode gerar a mesma seqüência pseudo-randômica repetidamente, chamando **srand()**, com a mesma semente, antes de iniciar a seqüência.

Exemplo

Este programa usa a hora do sistema para inicializar randomicamente a função **rand()** usando **srand()**:

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

/* Retira a semente para rand da hora do sistema e mostra os
   primeiros 10 números.
*/
void main(void)
{
  int i, stime;
  long ltime;

  /* obtém a hora de calendário atual */
  ltime =time(NULL);
  stime = (unsigned) ltime/2;
  srand(stime);

  for(i=0; i<10; i++) printf("%d ", rand());
}
```

Função Relacionada

rand()

#include <stdlib.h>
double strtod(const char *start, char **end);

A função **strtod()** converte em um **double** a representação em string de um número armazenado na string apontada por *start* e devolve o resultado.

A função **strtod()** opera da seguinte forma. Primeiro, qualquer espaço em branco na string apontada por *start* é eliminado. Em seguida, qualquer caractere que constitua o número é lido. Qualquer caractere que não possa fazer parte de um número em ponto flutuante provoca a parada do processo. Isso inclui espaços em branco, pontuação (diferente do ponto) e caracteres diferentes de "E" ou "e". Finalmente, *end* passa a apontar o resto, se houver, da string original. Isso significa que, se **strtod()** for chamada com "100.00 Alicates", o valor 100.00 é devolvido e *end* aponta para o espaço que precede a palavra "Alicates".

Se ocorre um erro de conversão, **strtod()** devolve **HUGE_VAL** para estouro ou **-HUGE_VAL** para estouro negativo. Se nenhuma conversão pôde ocorrer, será devolvido zero. Em qualquer caso, a variável global *errno* recebe ERANGE, indicando um erro de escala.

Exemplo

Este programa lê números em ponto flutuante de uma matriz de caracteres:

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

void main(void)
{
  char *end, *start = "100.00 alicates 200.00 martelos";

  end = start;
  while(*start) {
    printf("%f, ", strtod(start, &end));
    printf("restante: %s\n", end);
    start = end;
    /* pula os não-dígitos */
    while(!isdigit(*start) && *start) start++;
  }
}
```

A saída é

100.000000, restante: alicates 200.00 martelos

200.000000, restante: martelos

Função Relacionada

atof()

#include <stdlib.h>
long strtol(const char *start, char **end, int radix);

A função **strtol()** converte em um **long** a representação em string de um número armazenado na string apontada por *start* em um **long** e devolve o resultado. A base do número é determinada por *radix*. Se *radix* é zero, a base é determinada por regras que governam a especificação de constantes. Se *radix* é um valor diferente de zero, então ele deve estar entre 2 e 36.

A função **strtol()** opera da seguinte forma. Primeiro, qualquer espaço em branco na string apontada por *start* é eliminado. Em seguida, cada caractere que constitui o número é lido. Qualquer caractere que não possa fazer parte de um número inteiro longo fará com que o processo pare. Isto inclui espaços em branco, pontuação e caracteres. Finalmente, *end* passa a apontar o resto, se houver algum, da string original. Isso significa que, se **strtol()** for chamada com "100 Alicates", o valor **100L** será devolvido e *end* apontará para o espaço que precede a palavra "Alicates".

Se ocorre um erro de conversão, **strtol()** devolve **LONG_MAX** para estouro ou **LONG_MIN** para estouro negativo. A variável global **errno** também recebe **ERANGE**, indicando um erro de escala. Se não pôde ocorrer nenhuma conversão, zero será devolvido.

Exemplo

Esta função lê números na base 10 da entrada padrão e devolve os seus equivalentes **long**.

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

long read_long(void)
{
  char start[80], *end;

  printf("Digite um número: ");
  gets(start);
  return strtol(start, &end, 10);
}
```

Função Relacionada

atol()

#include <stdlib.h>
unsigned long strtoul(const char *start, char **end, int radix);

A função **strtoul()** converte em um **unsigned long** a representação em string de um número armazenado na string apontada por *start* e devolve o resultado. A base do número é determinada por *radix*. Se *radix* é zero, a base é determinada por regras que governam a especificação de constantes. Se *radix* é um valor diferente de zero, então ele deve estar entre 2 e 36.

A função **strtoul()** opera da seguinte forma. Primeiro, qualquer espaço em branco na string apontada por *start* é eliminado. Em seguida, cada caractere que constitui o número é lido. Qualquer caractere que não possa fazer parte de um número inteiro longo sem sinal fará com que o processo pare. Isso inclui espaços em branco, pontuação e caracteres. Finalmente, *end* passa a apontar o resto, se houver algum, da string original. Isso significa que, se **strtoul()** for chamada com "100 Alicates", o valor **100L** será devolvido e *end* apontará para o espaço que precede a palavra "Alicates".

Se ocorre um erro de conversão, **strtoul()** devolve **ULONG_MAX** e a variável global **errno** também recebe **ERANGE**, indicando um erro de escala. Se não pôde ocorrer nenhuma conversão, será devolvido zero.

Exemplo

Esta função lê números na base 16 (hexadecimal) da entrada padrão e devolve seus **unsigned long** equivalentes.

```
#include <stdlib.h>

unsigned long read_unsigned_long(void)
{
  char start[80], *end;

  printf("digite um número: ");
  gets(start);
  return strtoul(start, &end, 16);
}
```

Função Relacionada

strtol()

#include <stdlib.h>
int system(const char *str);

A função **system()** passa a string apontada por *str* como um comando para o processador de comandos do sistema operacional.

Quando **system()** é chamada com um ponteiro nulo, ela devolve um valor diferente de zero se um processador de comandos está presente; caso contrário, devolve zero. (Lembre-se de que alguns códigos em C são executados em sistemas dedicados que não têm sistemas operacionais e processadores de comando.) O valor devolvido por **system()**, quando esta é chamada com um pon-

teiro para uma string de comando, é definido pela implementação. No entanto, geralmente devolve zero se o comando foi executado com sucesso, e um valor diferente de zero, caso contrário.

Exemplo

Utilizando o sistema operacional DOS, este programa mostra o conteúdo do diretório de trabalho atual.

```
#include <stdlib.h>

void main(void)
{
  system("dir");
}
```

Função Relacionada

exit()

#include <stdarg.h>
void va_arg(va_list argptr, type);
type va_start(va_list argptr, last_parm);
void va_end(va_list argptr);

As macros **va_arg()**, **va_start()** e **va_end()** trabalham em conjunto para permitir que um número variável de argumentos seja passado para uma função. O exemplo mais comum de uma função que recebe um número variável de argumentos é **printf()**. O tipo **va_list** é definido em STDARG.H.

O procedimento geral para criar uma função que pode receber um número variável de argumentos é o seguinte: a função deve ter pelo menos um parâmetro conhecido, podendo, porém, ter mais anteriormente à lista variável de parâmetros. O parâmetro conhecido, mais à direita, é *last_parm*. Antes que qualquer dos parâmetros de comprimento variável possa ser acessado, o argumento ponteiro *argptr* deve ser inicializado por meio de uma chamada a **va_start()**. Em seguida, parâmetros são devolvidos via chamadas a **va_arg()**, com *type* sendo o tipo do próximo parâmetro. Finalmente, após todos os parâmetros terem sido lidos e antes de retornar da função, deve ser feita uma chamada a **va_end()** para garantir que a pilha seja corretamente restaurada. Se **va_end()** não for chamada, será muito provável que ocorra um "crash" do programa.

Exemplo

Este programa utiliza **sum_series()** para devolver a soma de uma série de números. O primeiro argumento contém o número de argumentos que se sucedem. Neste exemplo, o programa soma os cinco primeiros elementos da série:

$$\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \dots + \frac{1}{2^N}$$

O resultado mostrado é **0.968750**.

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

double sum_series(int num, ...);

/* Exemplo de argumentos de comprimento variável
   soma de uma série. */
void main(void)
{
  double d;

  d = sum_series(5, 0.5, 0.25, 0.125, 0.0625, 0.03125);

  printf("A soma da série: %f.\n", d);
}

double sum_series(int num, ...)
{
  double sum=0.0, t;
  va_list argptr;

  /* inicializa argptr */
  va_start(argptr, num);

  /* soma a série */
  for( ; num; num--) {
    t = va_arg(argptr, double); /*obtém próximo argumento */
    sum += t;
  }

  /* finaliza a lista de argumentos */
  va_end(argptr);
  return sum;
}
```