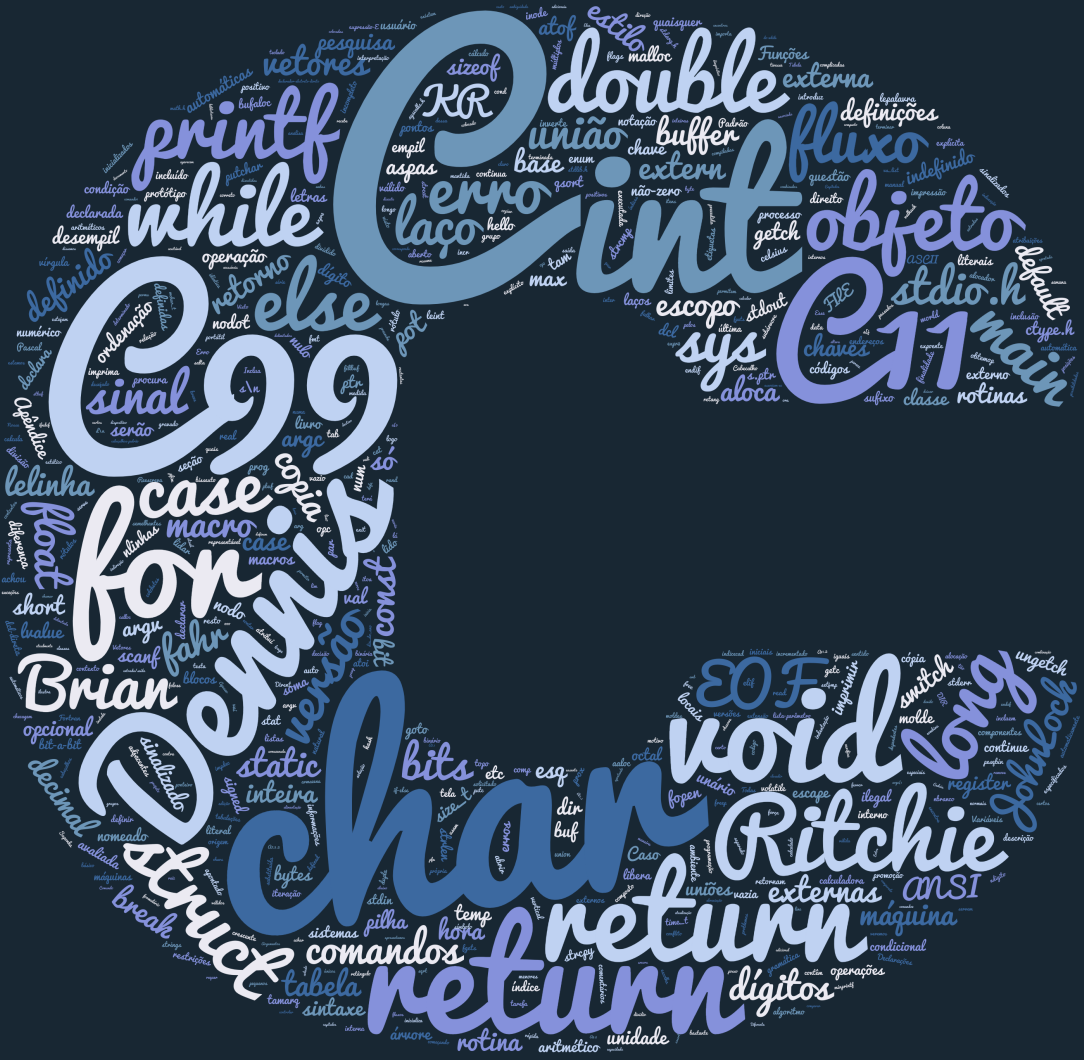
# A Linguagem de Programação C

## Padrão C11

## Kernighan & Ritchie



*First printing, March 2019*

# Conteúdo

# 1. Uma Introdução por Meio de Exemplos

Vamos começar com uma rápida introdução a **C**. Nosso objetivo é mostrar os elementos essenciais da linguagem em programas reais, mas sem esmiuçar detalhes, regras formais e exceções. Neste ponto não estamos tentando ser completos ou mesmo precisos (salvo nos exemplos, que são corretos). Queremos levá-lo, tão rápido quanto possível, ao ponto no qual você possa escrever programas úteis; devemos, portanto, nos concentrar nos elementos básicos: variáveis e constantes, aritmética, fluxo de controle, funções e noções elementares de entrada e saída. Intencionalmente, deixamos de fora deste capítulo facilidades de **C** que são de vital importância para escrever programas maiores. Isto inclui ponteiros, estruturas, a maior parte do rico conjunto de operadores C, vários comandos de fluxo de controle e a biblioteca-padrão.

Este enfoque tem suas desvantagens. O mais notável é que o tratamento completo de algum aspecto da linguagem não é encontrado num único local, e o capítulo, por ser breve, pode também confundir. E já que os exemplos não podem usar o poder total de **C**, eles não são tão concisos e elegantes quanto poderiam ser. Tentamos minimizar esses efeitos, mas esteja prevenido. Uma outra desvantagem é que os próximos capítulos necessariamente repetirão alguma parte deste. Esperamos que a repetição ajude mais do que atrapalhe.

Em todo caso, programadores experientes devem ser capazes de extrapolar o material deste capítulo para suas próprias necessidades de programação. Iniciantes deveriam suplementá-lo, tentando pequenos programas, similares aos deste capítulo. Ambos os grupos podem usá-lo como infraestrutura na qual se acoplam as descrições mais detalhadas que iniciamos no Capítulo 2.

## 1.1 Início

A única forma de aprender uma nova linguagem de programação é escrever programas nesta linguagem. O primeiro programa a escrever é o mesmo em todas as linguagens:

```
Imprima as palavras
     hello, world
```

Isto é um grande obstáculo: para saltá-lo você deve ser capaz de criar o texto do programa em

algum lugar, compilá-lo com sucesso, carregá-lo, executá-lo e ver para onde foi sua resposta. Uma vez aprendidos estes detalhes mecânicos, tudo o mais é relativamente fácil.

Em C, o programa para imprimir "Hello World" é:

Programa 1.1: Hello World.

```c
#include <stdio.h>

int main(void)
{
    printf("hello, world\n");
    return 0;
}
```

O modo de executar este programa depende do sistema que você está usando. Como um exemplo específico, no sistema operacional **UNIX/LINUX** você deve criar o programa-fonte num arquivo cujo nome termine com "*.c*", tal como *hello.c* e então compilá-lo com o comando

```
cc hello.c
```

Se você não errar nada, tal como omitir um caractere ou escrever algo errado, a compilação transcorrerá silenciosamente, e produzirá um arquivo executável chamado *a.out*. Executando-o com o comando

```
./a.out
```

produzirá como saída:

```
hello, world
```

Em outros sistemas, as regras são diferentes; verifique com um especialista.

**Adendo 1.1.1 — Como instalar o compilador C.** Em sistemas Linux baseados em **Debian**, a instalação do compilador C é bem fácil, basta abrir um terminal de comandos e digitar:

```
sudo apt update
```

para atualizar seus pacotes e depois:

```
sudo apt install build-essential
```

para instalar o pacote padrão de programação C para GNU (gcc, g++ e make). Em sistemas *Windows* você deve usar o *Windows Subsystem for Linux (WSL)*.

Para instalar o pacote de manuais, você deve digitar:

```
sudo apt-get install manpages-dev
```

Para confirmar que o compilador GCC foi corretamente instalado, use o comando `gcc -version` que imprimirá a versão do GCC:

```
gcc --version
```

**Adendo 1.1.2 — Como compilar o programa C.** Antes de tudo, você deve editar seu programa em um editor de texto puro, isto é, sem formatação. Em sistemas Linux, pode ser o editor **nano**, **emacs** ou **vi**. Uma vez salvo o programa como hello.c, você compilá-lo simplesmente digitando

| | |
|---|---|
| `#include <stdio.h>` | *inclui funções da biblioteca padrão* |
| `int main(void)` | *define uma função chamada* ***main*** *que não recebe argumentos.*<br>*Comandos de* ***main*** *são delimitados por chaves.* ***int*** *define um valor de retorno para o sistema operacional. A palavra* ***void*** *significa que a função não recebe parâmetros* |
| `{`<br>`printf("hello, world\n");` | main *chama a função de biblioteca* ***printf*** *para imprimir esta sequência de caracteres;* ***\n*** *representa o caractere de nova linha.* |
| `return 0;`<br>`}` | ***return 0*** *retorna ao sistema operacional o valor* zero*, significando que o programa terminou normalmente.* |

Tabela 1.1: O primeiro programa em C

```
gcc hello.c
```

ou, se quiser escolher o nome do programa executável:

```
gcc hello.c -o hello
```

isto vai gerar um programa chamado *hello*.

Agora, algumas explicações sobre o programa. Um programa em **C**, independentemente de seu tamanho, consiste em funções e variáveis. Uma função contém comandos que especificam as operações a serem feitas e as variáveis armazenam valores usados durante a computação. Funções em C são similares a funções e sub-rotinas de um programa **Fortran**, ou às procedures de **Pascal**. No nosso exemplo, *main* é uma função. Normalmente, você pode escolher o nome que quiser, mas "*main*" é um nome especial — seu programa começa a ser executado no início de *main*. Isso significa que todo programa deve ter um *main* em algum lugar.

*main* normalmente invocará outras funções para realizar seu trabalho, algumas delas você mesmo escreveu, outras vêm de bibliotecas de funções previamente escritas. A primeira linha do programa,

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

diz ao compilador para incluir informação sobre a biblioteca-padrão de entrada/saída; esta linha aparece no início de muitos arquivos-fonte em **C**. A biblioteca-padrão é descrita no Capítulo 7 e no Apêndice B.

Um método para se comunicar dados entre funções é fazer com que a função de chamada forneça uma lista de valores, chamados argumentos, à função que ela chama. Os parênteses após o nome da função delimitam a lista de argumentos. Neste exemplo, *main* é definida como uma função que não espera argumentos, o que é indicado pela palavra *void*.

Os comandos de uma função são delimitados por chaves **{}**. A função *main* contém o comando.

```
    printf("hello, world\n");
```

Uma função é chamada indicando-se seu nome, seguido por uma lista de argumentos entre parênteses, de modo que esse comando chama a função *printf* com o argumento "hello, world\n ". *printf* é uma função de biblioteca que imprime uma saída, neste caso, a cadeia de caracteres entre aspas.

Uma sequência de qualquer número de caracteres entre aspas, como "hello, world\n ", é chamada cadeia de caracteres ou constante do tipo string. Por ora, nosso único uso de cadeias de caracteres será como argumento para *printf* e outras funções.

A sequência **\n** na string é a notação **C** para o caractere de nova-linha, que, quando impresso, provoca o avanço do cursor do terminal para o início da próxima linha. Se você omitir o **\n** (vale a pena tentar), você encontrara sua saída não terminada por uma alimentação de linha. A única maneira de se obter um caractere de nova-linha no argumento de *printf* é com o uso de **\n** ; se você tentar algo como

```
printf("hello, world
");
```

o compilador **C** imprimirá um diagnóstico não amigável sobre a falta de aspas.

*printf* nunca fornece uma nova-linha automaticamente, de forma que múltiplas ativações podem ser usadas para construir uma linha de saída, passo a passo. Nosso primeiro programa poderia ter sido escrito como

Programa 1.2: "Segunda versão de Hello World."

```
#include <stdio.h>

int main(void)
{
    printf("hello, ");
    printf("world");
    printf("\n");
    return 0;
}
```

para produzir uma saída idêntica.

Observe que \n representa somente um único caractere. Uma sequência de escape tal como \n provê um mecanismo geral e extensível para a representação de caracteres difíceis de obter ou invisíveis. Entre os outros que **C** provê estão \t para tabulação, \b para retrocesso, \" para a aspas, e \\ para a contrabarra propriamente dita. Há uma lista completa na Seção 2.3.

**Exercício 1.1** Execute este programa em seu sistema. Experimente deixar de fora partes do programa para ver qual mensagem de erro você obtém. ■

**Exercício 1.2** Verifique o que acontece quando a string argumento de *printf* contém \c, onde *c* é algum caractere não listado acima. ■

## 1.2 Variáveis e Expressões Aritméticas

O próximo programa imprime a seguinte tabela de graus *Fahrenheit* e seus correspondentes graus centígrados ou *Celsius*, usando a fórmula $^{\circ}C = (5/9)(^{\circ}F - 32)$.

```
0     -17
20    -6
40    4
60    15
80    26
100   37
120   48
140   60
160   71
180   82
200   93
220   104
240   115
260   126
280   137
300   148
```

O programa em si ainda consiste na definição de uma única função chamada *main*. Ela é maior do que aquela que imprimiu "*hello, world*", mas não é complicada. Ela introduz diversas novas ideias, incluindo comentários, declarações, variáveis, expressões aritméticas, laços e saída formatada.

Programa 1.3: Conversão Fahrenheit - Celsius.

```c
#include <stdio.h>

/* imprime a tabela de conversão Fahrenheit-Celsius
   para fahr = 0, 20,..., 300 */
int main(void)
{
    int fahr, celsius;
    int inicio, fim, incr;
    inicio = 0; // limite inferior da tabela
    fim = 300;  // limite superior
    incr = 20;  // incremento
    fahr = inicio;

    while (fahr <= fim) {
        celsius = 5 * (fahr - 32) / 9;
        printf ("%d\t%d\n", fahr, celsius);
        fahr = fahr + incr;
    }

    return 0;
}
```

As duas linhas

```c
/* imprime a tabela de conversão Fahrenheit-Celsius
   para fahr = 0, 20,..., 300 */
```

são um comentário que neste caso explica brevemente o que o programa faz. Quaisquer caracteres entre /* e */ são ignorados pelo compilador; eles podem ser usados livremente para tornar o programa mais fácil de ser entendido. Comentários podem aparecer em qualquer lugar onde possa aparecer um espaço em branco, uma marca de tabulação ou uma nova linha.

C99 Na versão C99 foram introduzidos os comentários com duas barras (//). Este tipo de comentário termina automaticamente ao final da linha, como em:

```
inicio = 0; // limite inferior da tabela
fim = 300;  // limite superior
incr = 20;  // incremento
```

Em **C**, todas as variáveis devem ser declaradas antes de usadas, normalmente, no início da função, antes de qualquer comando executável. Uma declaração anuncia as propriedades das variáveis; ela consiste de um nome de tipo e uma lista de variáveis, como

```
int fahr, celsius;
int inicio, fim, incr;
```

C99 Em C99 as variáveis podem ser declaradas mesmo após os comandos, desde que o sejam antes de serem usadas pelo programa. Uma boa prática é declarar a variável o mais perto possível de onde ela será usada.

O tipo *int* implica que as variáveis listadas são inteiras; *float* define ponto flutuante, isto é, números que podem ter uma parte fracionária. A faixa de **int** e **float** depende da máquina que você está usando; **int**s de **16 bits**, que se encontram na faixa entre $-32768$ e $+32767$, são comuns, assim como **int**s de **32 bits**. Um número **float** é normalmente uma quantidade de **32 bits**, com pelo menos seis dígitos significativos e magnitude geralmente entre $10^{-38}$ e $10^{+38}$.

C provê vários outros tipos de dados básicos além de *int* e *float*, incluindo:

**char** caractere - um único byte
**short** inteiro curto
**long** inteiro longo
**double** ponto flutuante em dupla precisão

C99 Em C99 temos também os tipos:
**long long** inteiro mais longo
**long double** ponto flutuante longo em dupla precisão

Os tamanhos desses objetos são também dependentes da máquina; detalhes estão no Capítulo 2. Há também vetores, estruturas e uniões desses tipos básicos, ponteiros para eles, e funções que os retornam; todos serão vistos ao longo do livro.

O cálculo atual no programa de conversão de temperatura começa com as atribuições

```
inicio = 0; // limite inferior da tabela
fim = 300;  // limite superior
incr = 20;  // incremento
fahr = inicio;
```

que atribuem às variáveis seus valores iniciais. Comandos individuais são terminados por *ponto-e-vírgula*.

Cada linha da tabela é computada da mesma forma, e usamos, portanto, um laço que se repete uma vez por linha; essa é a finalidade do laço **while**.

```
while (fahr <= fim) {
  ...
}
```

O laço **while** opera da seguinte forma: a condição entre parênteses é testada. Se ela é verdadeira (*fahr* é menor ou igual a *fim*), o corpo do laço (todos os comandos entre {e }) é executado. Então a condição é retestada, e se verdadeira, o corpo é executado novamente. Quando o teste se tornar *falso* (*fahr* exceder *fim*) o laço terminará, e a execução continuará no comando que segue o laço. Não há mais comandos neste programa e ele termina.

O corpo de um **while** pode ter um ou mais comandos entre chaves, como no conversor de temperatura, ou um único comando sem chaves como em

```
while (i < j)
  i = 2 * i;
```

Em ambos os casos, os comandos controlados pelo **while** estão indentados por uma posição de tabulação (que mostramos como dois espaços) de forma que você possa ver rapidamente que comandos estão dentro do laço. A indentação enfatiza a estrutura lógica do programa. Embora **C** seja muito liberal quanto ao posicionamento de comandos, a indentação correta e o uso de espaços são críticos na construção de programas fáceis de serem entendidos pelas pessoas. Recomendamos escrever somente um comando por linha, e (normalmente) deixar espaços em volta dos operadores. A posição das chaves é menos importante; escolhemos um entre vários estilos populares. Escolha um estilo que lhe agrade, e use-o sempre.

A maior parte do trabalho é feita no corpo do laço. A temperatura **Celsius** é computada e atribuída a *celsius* pelo comando

```
        celsius = 5 * (fahr - 32) / 9;
```

A razão para multiplicarmos por cinco e depois dividirmos por nove ao invés da construção mais simples $5/9$ é que, em **C**, como na maioria das outras linguagens, a divisão inteira trunca o resultado, de forma que qualquer parte fracionária é perdida. Como 5 e 9 são inteiros, $5/9$ seria truncado para **zero**, e assim todas as temperaturas em Celsius seriam relatadas como zero.

Esse exemplo também mostra um pouco mais sobre a operação de *printf*. *printf* é, na realidade, uma função geral de conversão de formatos, que descreveremos por completo no Capítulo 7. Seu argumento é uma string de caracteres a ser impressa, onde cada sinal % indica onde um dos outros (segundo, terceiro,...) argumentos deve ser substituído, e sob que formato será impresso. Por exemplo, %d especifica um argumento inteiro, de modo que o comando

```
  printf ("%d\t%d\n", fahr, celsius);
```

faz com que valores dos dois inteiros *fahr* e *celsius* sejam impressos, com uma tabulação (\t) entre eles. Cada construção do tipo % no primeiro argumento de *printf* casa-se com o segundo argumento, terceiro argumento, etc.; eles devem, correspondentemente, alinhar-se corretamente por número e tipo, caso contrário você obterá respostas sem significado.

A propósito, *printf* não faz parte da linguagem **C** ; não há entrada ou saída definida na linguagem. *printf* é apenas uma função útil que faz parte da biblioteca-padrão de funções normalmente acessíveis aos programas em **C**. No entanto, o comportamento de *printf* é definido no padrão **ANSI**, de modo que suas propriedades devem ser idênticas com qualquer compilador e biblioteca que esteja de acordo com o padrão.

Para nos concentrarmos somente na linguagem **C**, não vamos falar muito sobre entrada e saída até o Capítulo 7. Em particular, vamos deixar o assunto de entrada formatada até lá. Se você tem de ler números, leia sobre a função *scanf* na Seção 7.4. *scanf* é muito parecida com *printf*, exceto que ela lê a entrada ao invés de imprimir na saída.

Há alguns problemas com o programa de conversão de temperatura. O mais simples é que a saída não é muito bonita, pois os números não são justificados à direita. Isso é fácil de ser consertado; se incluirmos em cada %d do comando *printf* uma largura, os números impressos serão justificados à direita dentro dos seus campos. Por exemplo, poderíamos escrever

```
printf ("%3d %6d\n", fahr, celsius);
```

para imprimir o primeiro número de cada linha em um campo de três dígitos de largura, e o segundo em um campo de seis dígitos, assim:

```
  0    -17
 20     -6
 40      4
 60     15
 80     26
100     37
 ...
```

O problema mais sério vem do fato de, por termos usado a aritmética para inteiros, as temperaturas em *Celsius* não são muito precisas; por exemplo, $0^oF$ corresponde a cerca de $-17.8^oC$, e não $-17$. Para obtermos respostas mais precisas, temos que usar a aritmética de ponto flutuante no lugar dos inteiros. Isto requer algumas mudanças no programa. Aqui está uma segunda versão.

Programa 1.4: Conversão Fahrenheit- Celsius com ponto flutuante.

```
#include <stdio.h>

/* imprime a tabela de conversão Fahrenheit-Celsius
    para fahr = 0, 20,   300; versão com ponto flutuante */
int main(void)
{
    float fahr, celsius;
    int inicio, fim, incr;
    inicio = 0; // limite inferior da tabela
    fim = 300;  // limite superior
    incr = 20;  // incremento
    fahr = inicio;

    while (fahr <= fim) {
        celsius = (5.0 / 9.0) * (fahr - 32.0);
        printf ("%3.0f %6.1f\n", fahr, celsius);
        fahr = fahr  + incr;
    }

    return 0;
}
```

Isto é muito semelhante ao anterior, exceto que *fahr* e *celsius* são declarados como **float**, e a fórmula para conversão é escrita em uma forma mais natural. Não pudemos usar $5/9$ na versão anterior porque a divisão de inteiros truncaria o resultado para zero. Um ponto decimal em uma constante, no entanto, indica que ela é ponto flutuante, de modo que $5.0/9.0$ não é truncado porque é a razão de dois valores de ponto flutuante.

Se um operador aritmético tiver operandos inteiros, uma operação inteira é executada. Se um operador aritmético tiver um operando de ponto flutuante e um operando inteiro, no entanto, o inteiro será convertido para ponto flutuante antes que a operação seja realizada. Se tivéssemos escrito $(fahr - 32)$, o 32 seria automaticamente convertido para ponto flutuante. Apesar disso, escrever constantes de ponto flutuante com pontos decimais explícitos, mesmo quando possuem valores integrais, enfatiza sua natureza de ponto flutuante para os leitores humanos.

As regras detalhadas para quando os inteiros são convertidos para ponto flutuante estão no Capítulo 2. Por ora, observe que a atribuição

```
fahr = inicio;
```

e o teste

```
while (fahr <= fim)
```

também funcionam de forma natural — o **int** é convertido para *float* antes que a operação seja feita.

A especificação de conversão de *printf* %3.0f diz que um número de ponto flutuante (aqui *fahr*) deve ser impresso com pelo menos três caracteres de largura, sem ponto decimal e sem dígitos fracionários. %6.1f descreve um outro número (*celsius*) que deve ser impresso com pelo menos seis caracteres de largura, com 1 dígito após o ponto decimal. A saída ficará assim:

```
 0     -17.8
20      -6.7
40       4.4
   ...
```

A largura e precisão podem ser omitidos da especificação: %6f diz que o número deve ter pelo menos seis caracteres de largura; %.2f especifica dois caracteres após o ponto decimal, mas a largura não tem restrição; e %f simplesmente diz para imprimir o número como ponto flutuante.

| | |
|---|---|
| %d | imprime como inteiro decimal |
| %6d | imprime como inteiro decimal, pelo menos 6 caracteres de largura |
| %f | imprime como ponto flutuante |
| %6f | imprime como ponto flutuante, pelo menos 6 caracteres de largura |
| %.2f | imprime como ponto flutuante, 2 caracteres após o ponto decimal |
| %6.2f | imprime como ponto flutuante, pelo menos 6 caracteres de largura e 2 após ponto decimal |

Entre outros, printf também reconhece %o para octal, %x para hexadecimal, %c para caractere, %s para cadeia (string) de caracteres e %% para o próprio %.

**Exercício 1.3** Modifique o programa de conversão de temperatura para imprimir um cabeçalho acima da tabela. ■

**Exercício 1.4** Escreva um programa para imprimir a tabela correspondente de Celsius a Fahrenheit. ■

## 1.3 O Comando For

Há uma infinidade de maneiras de se escrever um programa para uma tarefa em particular. Vamos tentar uma variante do conversor de temperatura.

Programa 1.5: Faherenheit para Celsius com for.

```
#include <stdio.h>

/* imprime tabela Fahrenheit-Celsius */
int main(void)
{
    for (int fahr = 0; fahr <= 300; fahr = fahr + 20) {
        printf("%3d %6.1f\n", fahr, (5.0 / 9.0) * (fahr - 32));
    }

    return 0;
}
```

Ele produz as mesmas respostas, mas certamente parece diferente. Uma mudança principal é a eliminação da maioria das variáveis; somente *fahr* permaneceu, como **int**. Os limites *inicial* e *final* e o tamanho do *incremento* aparecem somente como constantes no comando **for**, que é uma nova construção, e a expressão que calcula a temperatura **Celsius** agora aparece como terceiro argumento de *printf* ao invés de ser um comando de atribuição separado.

Esta última mudança é um exemplo de uma regra geral de **C** – em qualquer contexto onde é permissível usar o valor de uma variável de algum tipo, você pode usar uma expressão mais complicada desse tipo. Como o terceiro argumento de *printf* deve ser um valor de ponto flutuante para combinar com o %6.1f, qualquer expressão em ponto flutuante pode aparecer aí.

O comando *for* é um *loop*, uma generalização do **while**. Se você compará-lo com o **while** anterior, sua operação deverá ser clara. Dentro dos parênteses existem três partes, separadas por ponto-e-vírgula. A primeira parte

```
int fahr = 0
```

é executada uma vez, antes do laço ser iniciado.

C99 **C ANSI** não permitia o uso de declarações de variáveis dentro de comandos. Em **C99** o **for** pode conter a declaração da variável *fahr*, como mostrado no programa:

```
for (int fahr = 0; fahr <= 300; fahr = fahr + 20)
```

Fique atento, porém, ao escopo da variável definida dentro do laço **for**, no caso, *fahr* existe apenas dentro do laço e seria um erro tentar usar seu valor fora dele.

Talvez seja necessário configurar o compilador para aceitar a sintaxe de *C99*. No compilador **gcc** isso pode ser feito passando o parâmetro *-std=c99*.

```
gcc -std=c99 prog.c -o prog
```

ou, em sistemas **Linux**, simplesmente

```
c99 prog.c -o prog
```

A segunda parte é o teste ou condição que controla o laço:

```
fahr <= 300
```

Esta condição é avaliada; se verdadeira, o corpo do laço (um único comando *printf*) é executado. finalmente, o passo de incremento

```
fahr = fahr + 20
```

é feito, e a condição reavaliada. O laço termina quando a condição se torna falsa. Como no **while**, o corpo do laço pode ser um único comando ou um grupo de comandos entre chaves. A inicialização, condição e incremento podem ser qualquer expressão.

A escolha entre **while** e **for** é arbitrária, baseada no que parece mais claro. O **for** é normalmente apropriado para laços onde a inicialização e incremento são comandos simples e logicamente relacionados, pois é mais compacto que o **while** e mantém os comandos de controle do laço juntos no mesmo local.

**Exercício 1.5** Modifique o programa de conversão de temperatura para imprimir a tabela em ordem inversa, isto é, de 300 graus até 0 grau. ■

## 1.4 Constantes Simbólicas

Uma última observação antes de deixarmos para sempre o conversor de temperatura. Não é boa prática de programação utilizar "*números mágicos*" como 300 e 20 num programa; eles trazem pouca informação para alguém que tenha de ler o programa posteriormente, e são difíceis de serem alterados de forma sistemática. Uma forma de lidar com os números mágicos é dar a eles nomes significativos. Uma linha **#define** define um nome simbólico ou constante simbólica como sendo uma string de caracteres em particular:

```
#define nome texto_substituto
```

Depois disso, qualquer ocorrência do nome (não entre aspas ou parte de outro nome) será substituída pelo texto substituto correspondente. O nome tem a mesma forma de um nome de variável: uma sequência de letras e dígitos que começam com uma letra. O texto substituto pode ser qualquer sequência de caracteres; ele não é limitado a números.

Programa 1.6: Faherenheit para Celsius com constantes simbólicas.

```
#include <stdio.h>

#define INICIO 0 // limite inferior da tabela
#define FIM 300  // limite superior
#define INCR 20  // incremento

/* imprime tabela Fahrenheit-Celsius */
int main(void)
{
    for (int fahr = INICIO; fahr <= FIM; fahr = fahr + INCR) {
        printf ("%3d %6.1f\n", fahr, (5.0 / 9.0) * (fahr - 32));
    }

    return 0;
}
```

As quantidades `INICIO`, `FIM` e `INCR` são constantes simbólicas, e não variáveis, de modo que não aparecem nas declarações. Os nomes das constantes simbólicas, por convenção, são escritos com letras maiúsculas para que sejam facilmente distintos dos nomes de variáveis em minúsculas. Observe que não existe um ponto-e-vírgula ao final de uma linha #define.

**Adendo 1.4.1 — const.** Uma melhor opção é usar a palavra reservada **const** que diz que o valor declarado a seguir será constante durante a execução do bloco de código. Sua vantagem em relação ao define é que ele possui um tipo e assim pode ser avaliado pelo compilador.

Programa 1.7: Faherenheit para Celsius com const.

```
#include <stdio.h>

/* imprime tabela Fahrenheit-Celsius */
int main(void)
{
  const int INICIO = 0;
  const int FIM = 300;
  const int INCR = 20;

  for (int fahr = INICIO; fahr <= FIM; fahr = fahr + INCR){
    printf ("%3d %6.1f\n", fahr, (5.0 / 9.0) * (fahr - 32));
```

```
  }

  return 0;
}
```

## 1.5 Entrada e Saída de Caracteres

Vamos agora considerar uma família de programas relacionados para o processamento de dados em forma de caracteres. Você verá que muitos programas são apenas versões expandidas dos protótipos discutidos aqui.

O modelo de entrada e saída suportado pela biblioteca-padrão é muito simples. O texto de entrada ou saída, não importa de onde ele vem ou para onde vai, é manipulado como um fluxo de caracteres. Um fluxo de texto é uma sequência de caracteres dividida em linhas; cada linha consiste em zero ou mais caracteres seguidos por um caractere de nova linha. A responsabilidade por fazer com que cada fluxo de entrada ou saída esteja de acordo com este modelo é da biblioteca; o programador **C** usando a biblioteca não precisa se preocupar sobre como as linhas são representadas fora do programa.

A biblioteca-padrão provê diversas funções para leitura e escrita de um caractere por vez, das quais *getchar* e *putchar* são as mais simples. Toda vez que for chamada, *getchar* lê o próximo caractere de entrada em um fluxo de texto e o retorna como seu valor. Isto é, após

```
c = getchar()
```

a variável *c* contém o próximo caractere de entrada. Os caracteres normalmente vêm do teclado; a entrada de arquivos é discutida no Capítulo 7.

A função *putchar* imprime um caractere toda vez que for chamada:

```
putchar(c)
```

imprime, normalmente, na tela, o conteúdo da variável inteira *c* sob a forma de um caractere. As chamadas a *putchar* e *printf* podem ser intercaladas; a saída aparecerá na ordem em que as chamadas são feitas.

### 1.5.1 Cópia de Arquivos

Com *getchar* e *putchar*, você pode escrever uma grande quantidade de código útil sem saber nada mais sobre entrada e saída. O exemplo mais simples é um programa que copia sua entrada para sua saída com um caractere de cada vez:

```
lê um caractere
while (caractere não é indicador de fim-de-arquivo)
  sai com o caractere lido
  lê um caractere
```

Convertendo isto para **C** :

Programa 1.8: Cópia de Arquivos.

```
#include <stdio.h>

/* copia entrada para saída; 1a. versão */
int main(void)
{
    int c;
    c = getchar();
```

```
    while (c != EOF) {
        putchar(c);
        c = getchar();
    }

    return 0;
}
```

O operador relacional `!=` significa "*não igual a*".

Naturalmente, o que aparece como um caractere no teclado ou tela, como em tudo mais, é armazenado internamente apenas como um padrão de bits. O tipo **char** foi feito especificamente para armazenar esses dados de caractere, mas qualquer tipo inteiro pode ser usado. Usamos **int** por um motivo sutil, mas importante.

O problema é distinguir o fim da entrada dos dados válidos. A solução é que *getchar* retorna um valor distinto quando não há mais entrada, um valor que não pode ser confundido com qualquer caractere real. Este valor é chamado *EOF*, o "end of file" (fim de arquivo). Devemos declarar *c* por meio de um tipo grande o suficiente para conter qualquer valor que *getchar* retorne. Não podemos usar **char** porque *c* deve conter, além de qualquer caractere possível, o valor do *EOF*. Portanto, usamos **int**.

*EOF* é um inteiro definido em <*stdio.h*>, mas o valor numérico específico não importa, desde que não seja o mesmo que qualquer valor de **char**. Usando a constante simbólica, temos certeza de que nada no programa depende do valor numérico específico.

O programa para copiar seria escrito mais resumidamente por programadores **C** experientes. Em **C**, qualquer atribuição, como

```
c = getchar()
```

é uma expressão e tem um valor, que é o valor do lado esquerdo após a atribuição. Isso significa que uma atribuição pode aparecer como parte de uma expressão maior. Se a atribuição de um caractere a *c* for colocada dentro da parte de teste de um laço **while**, o programa de cópia pode ser escrito da seguinte forma:

Programa 1.9: copia, segunda versão.

```
#include <stdio.h>

/* copia entrada para saída; 2a. versão */
int main(void)
{
    int c;

    while ((c = getchar()) != EOF) {
        putchar (c);
    }

    return 0;
}
```

O **while** obtém um caractere, atribuindo-o a *c*, e depois testa se o caractere foi o sinal de fim-de-arquivo. Se não foi, o corpo do *while* é executado, imprimindo o caractere. O **while** então é repetido. Quando o fim da entrada for finalmente atingido, o **while** termina assim como *main*.

Esta versão centraliza a entrada — há agora somente um referência a *getchar* — e encurta o programa. O programa resultante é mais compacto e, uma vez dominada a linguagem, mais fácil de

se ler. Você verá este estilo muitas vezes. (Ele também possibilita a criação de código impenetrável, tendência essa que tentaremos coibir.)

Os parênteses em torno da atribuição dentro da condição são necessários. A precedência de `!=` é maior que a de `=` , o que significa que na ausência de parênteses o teste relacional `!=` seria feito antes da atribuição `=`. Assim, o comando

```
c = getchar() != EOF
```

é equivalente a

```
c = (getchar() != EOF)
```

Isso teria o efeito indesejado de atribuir a *c* o valor 0 ou 1, dependendo se a chamada a *getchar* encontrou ou não o fim de arquivo. (Mais sobre isso no Capítulo 2.)

**Exercício 1.6** Verifique se a expressão `getchar() !=EOF` é 0 ou 1. ■

**Exercício 1.7** Escreva um programa para imprimir o valor de *EOF*. ■

**Adendo 1.5.1 — Como enviar EOF pelo teclado?.** Quando estiver executando este programa, você vai precisar terminar a entrada de dados enviando um caractere de EOF pelo teclado. Em sistemas **UNIX** (Linux) isso é feito pressionando simultaneamente as teclas *CONTROL* e *D*. Em sistemas **Windows**, deve-se pressionar *CONTROL* e *Z* e depois pressionar *ENTER*.

Também fique atento que este programa não vai repetir logo em seguida cada letra que for digitada. Ele repete cada linha, ou seja, lê todas as letras do texto digitado apenas após você pressionar a tecla *ENTER*.

### 1.5.2 Contagem de Caracteres

O próximo programa conta caracteres; ele é semelhante ao programa de cópia.

```
#include <stdio.h>

/* conta caracteres da entrada; 1a. versão */
int main(void)
{
    long nc = 0;

    while (getchar() != EOF) {
        ++nc;
    }

    printf ("%ld\n", nc);
    return 0;
}
```

O comando

```
++nc;
```

mostra um novo operador, `++` , que significa incremento de um. Você poderia escrever `nc = nc + 1`, mas `++nc` é mais conciso e frequentemente mais eficiente. Há um operador correspondente `--` para decrementar de 1. Os operadores `++` e `--` podem vir antes (`++nc`) ou depois (`nc++`) do nome; essas duas formas possuem valores diferentes em expressões, como será mostrado no Capítulo 2, mas as duas formas `++nc` e `nc++` incrementam `nc`. Por ora, usaremos o incremento prefixado.

O programa de contagem de caracteres acumula seu contador em uma variável do tipo **long** ao invés de **int**. Os inteiros **long** ocupam pelo menos 32 bits. Embora em algumas máquinas **int** e **long** sejam do mesmo tamanho, em outras um **int** ocupa 16 bits, com um valor máximo de 32767, e isso precisaria de uma entrada relativamente pequena para estourar um contador **int**. A especificação de conversão *%ld* diz a *printf* que o argumento correspondente é um inteiro **long**.

Para enfrentar números ainda maiores, você pode usar um **double** (ponto flutuante de "dupla precisão") ou um **long long**(C99). Usaremos também um comando **for** ao invés de *while*, para ilustrar uma forma alternativa de escrever o laço.

```
#include <stdio.h>

/* conta caracteres da entrada; 2a. versão */
int main(void)
{
  double nc = 0;

  for (; getchar () != EOF; ++nc)
    ;
  printf ("%.0f\n", nc);

  return 0;
}
```

*printf* usa $\%f$ tanto para **float** quanto para **double**; $\%.0f$ suprime a impressão do ponto decimal e da parte fracionária, que é zero.

O corpo deste laço **for** é **vazio**, pois todo o trabalho é feito nas partes de teste e incremento. Mas as regras gramaticais de **C** exigem que um comando **for** tenha um corpo. O ponto-e-vírgula isolado, chamado **comando nulo**, está aí para satisfazer esse requisito. Nós o colocamos em uma linha separada para torná-lo visível.

Antes de deixarmos o programa de contagem de caracteres, observe que se a entrada não tiver caracteres, o teste **while** ou **for** falham na primeira chamada a **getchar** e o programa produz zero, a resposta correta. Isso é importante. Uma das coisas mais agradáveis do **while** e **for** é que eles testam no início do laço, antes de prosseguir no corpo. Se não há nada para fazer, nada é feito, mesmo que o corpo do laço nunca seja executado. Os programas devem agir inteligentemente quando manipulam entradas vazias. Os comandos **while** e **for** ajudam a garantir que coisas razoáveis sejam feitas pelo programa em condições limites.

### 1.5.3 Contagem de Linhas

O próximo programa conta linhas da entrada. Como mencionamos acima, a **biblioteca-padrão** assegura que o fluxo de texto da entrada apareça como uma sequência de linhas, cada uma terminando com um caractere de *nova-linha*. Daí, contar linhas significa contar *novas-linhas*:

Programa 1.10: Conta linhas da entrada.

```
#include <stdio.h>

/* conta linhas na entrada */
int main(void)
{
    int c, nl = 0;

    while ((c = getchar ()) != EOF) {
        if (c == '\n') {
            ++nl;
```

```
        }
    }

    printf ("%d\n", nl);
    return 0;
}
```

O corpo do **while** consiste agora em um **if**, que por sua vez controla o incremento `++nl`. O comando **if** testa a condição entre parênteses, e se a condição for verdadeira, executa o comando (ou grupo de comandos entre colchetes) que segue. Usamos a indentação novamente para mostrar "*o que*" é controlado "*por quem*".

O duplo sinal de igual == é a notação de **C** para "*é igual a*" (como o = simples do Pascal ou o *.EQ.* do **Fortran**). Este símbolo é usado para diferenciar o teste de igualdade do = simples que **C** usa para a atribuição. Uma palavra de cautela: os iniciantes em **C** ocasionalmente escrevem = quando deveriam usar ==. Como veremos no Capítulo 2, o resultado é geralmente uma expressão válida, de modo que não haverá qualquer aviso.

Um caractere escrito entre aspas simples representa um valor inteiro igual ao valor numérico do caractere no conjunto de caracteres da máquina. Esta é chamada *constante de caractere*, embora seja apenas uma outra forma de escrever um inteiro pequeno. Assim, por exemplo, `'A'` é uma constante de caractere; no conjunto de caracteres **ASCII** seu valor é 65, a representação interna do caractere **A**. Naturalmente `'A'` é preferido no lugar de 65: seu significado é óbvio, e é independente de um conjunto de caracteres particular.

As sequências de escape usadas em constantes de string de caracteres são legais também em constantes de caracteres, de modo que `'\n'` representa o valor de um caractere de nova-linha, que é 10 em **ASCII**. Você deve observar com cuidado que `'\n'` é **um único caractere**, e em expressões é apenas um inteiro; por outro lado, `"\n"` é uma **constante de string** que contém apenas um caractere. O tópico de cadeias versus caracteres é discutido mais amiúde no Capítulo 2.

**Exercício 1.8** Escreva um programa que conte espaços, caracteres de tabulação e de nova-linha. ■

**Exercício 1.9** Escreva um programa que copie sua entrada na saída, trocando cada cadeia de dois ou mais espaços por um único espaço. ■

**Exercício 1.10** Escreva um programa para copiar sua entrada na saída, trocando cada tabulação por \t, cada retrocesso por \b e cada contrabarra por \\. Isso torna as marcas de tabulação e retrocessos visíveis de forma não ambígua. ■

### 1.5.4 Contagem de Palavras

O quarto da nossa série de programas úteis conta linhas, palavras e caracteres, com a definição vaga de que uma palavra é qualquer sequência de caracteres que não contém um *espaço*, *tabulação* ou *nova-linha*. Esta é uma versão simplória do programa *wc* do **UNIX**.

Programa 1.11: Conta linhas, palavras e caracteres.

```
#include <stdio.h>

#define DENTRO 1 // dentro de uma palavra
#define FORA 0   // fora de uma palavra

/* conta linhas, palavras e caracteres na entrada */
```

```
int main(void)
{
    int c, nl, np, nc;
    nl = np = nc = 0;
    int estado = FORA;

    while ((c = getchar()) != EOF) {
        ++nc;

        if (c == '\n') {
            ++nl;
        }

        if(c == ' ' || c == '\n' || c == '\t') {
            estado = FORA;
        } else if (estado == FORA) {
            estado = DENTRO;
            ++np;
        }
    }

    printf ("%d %d %d\n", nl, np, nc);
    return 0;
}
```

Toda vez que o programa encontra o primeiro caractere de uma palavra, ele conta mais uma palavra. A variável *estado* registra se o programa está atualmente em uma palavra ou não; inicialmente ele "*não está em uma palavra*", e a variável estado recebe o valor *FORA*. Preferimos as constantes simbólicas *DENTRO* e *FORA* aos valores literais 1 e 0 porque tornam o programa mais legível. Em um programa curto como este, isso faz pouca diferença mas, em programas maiores, o aumento na clareza compensa o esforço extra para escrevê-los inicialmente dessa forma. Você verá também que é mais fácil fazer mudanças em programas onde números mágicos apareçam somente como constantes simbólicas.

A linha

```
nl = np = nc = 0;
```

atribui às três variáveis o valor 0. Isso não é um caso especial, mas a consequência do fato de que uma atribuição tem um valor e é associativa da direita para a esquerda. Poderíamos ter escrito

```
nl = (np = (nc = 0));
```

O operador `||` significa *OU*, de forma que a linha

```
if(c==' ' || c=='\n' || c=='\t')
```

significa "*se c é um espaço ou c é uma nova-linha ou c é uma tabulação...*". (Lembre-se de que a sequência de escape \t é uma representação visível do caractere de tabulação.) Há um operador correspondente `&&` para *E*; sua precedência vem logo acima de `||`. As expressões conectadas por `&&` ou `||` são avaliadas da esquerda para a direita, e é garantido que a avaliação terminará assim que a veracidade ou falsidade for conhecida. Se *c* for um espaço, então não há necessidade de se testar se ele indica uma *nova-linha* ou *tabulação*, de forma que esses testes não são feitos. Isso não é muito importante aqui, mas sim em situações mais complicadas, como veremos em breve.

O exemplo também mostra o comando **else**, que especifica uma ação se a parte de condição de um comando **if** for falsa. O formato geral é

```
if (expressão)
```

```
  comando1
else
  comando2
```

Um e somente um dos dois comandos associados com o **if-else** é executado. Se a expressão é verdadeira, *comando1* é executado; se não, *comando2* é executado. Cada comando pode, de fato, ser um único comando ou diversos entre chaves. No programa de contagem de palavras, o único comando após o **else** é um **if** que controla dois comandos entre chaves.

**Exercício 1.11** Como você poderia testar o programa de contagem de palavras? Que tipos de entrada desvendariam mais facilmente os erros, se houvessem? ■

**Exercício 1.12** Escreva um programa que imprima sua entrada com uma palavra por linha. ■

## 1.6 Vetores

Vamos escrever um programa que conte o número de ocorrências de cada *dígito*, de *caracteres de espaço em branco* (espaço, tabulação, nova-linha), e de todos os outros caracteres. Isso é artificial, é claro, mas permite ilustrar vários aspectos de **C** num único programa.

Há doze categorias de entrada, de modo que é conveniente usar um vetor para conter o número de ocorrências de cada dígito, ao invés de dez variáveis individuais. Aqui está uma versão do programa;

Programa 1.12: Conta dígitos e espaços.

```
#include <stdio.h>

/* conta dígitos, espaços, outros */
int main(void)
{
    int ndigito [10];

    for (int i = 0; i < 10; ++i) {
        ndigito [i] = 0;
    }

    int c, nbranco, noutro;
    nbranco = noutro = 0;

    while ((c = getchar()) != EOF)
        if (c >= '0' && c <= '9') {
            ++ndigito[c - '0'];
        } else if (c == ' ' || c == '\n' || c == '\t') {
            ++nbranco;
        } else {
            ++noutro;
        }

    printf ("dígitos =");

    for (int i = 0; i < 10; ++i) {
        printf (" %d", ndigito[i]);
    }
```

```
    printf (", espaço branco=%d, outros=%d\n", nbranco, noutro);
    return 0;
}
```

A saída deste programa sobre si mesmo é

```
dígitos = 10 3 0 0 0 0 0 0 0 1, espaço branco=249, outros=387
```

**Adendo 1.6.1 — Redirecionamento de entrada e saída.** Para executar um programa sobre si mesmo utilize o redirecionamento do ambiente, com os sinais <, para entrada e > para saída. Assim, o comando

```
./prog < prog.c
```

vai usar o programa prog.c como entrada de prog. Se você quiser redirecionar a saída, use

```
./prog > saida.txt
```

E para redirecionar entrada e saida, combine os dois

```
./prog < prog.c > saida.txt
```

Você também pode usar >>:

```
./prog < prog.c >> saida.txt
```

e vai acumular a saída do programa em *saida.txt*, isto é, cada saída do programa será guardada em *saida.txt* sem apagar o conteúdo anterior.

A declaração

```
int ndigito[10];
```

declara *ndigito* como sendo um vetor de 10 inteiros. Os subscritos (índices) dos vetores sempre começam de 0 em **C**, de modo que os elementos são $ndigito[0], ndigito[1], \ldots, ndigito[9]$. Isso se reflete nos laços **for** que inicializam e imprimem o vetor.

Um subscrito pode ser qualquer expressão inteira, incluindo variáveis inteiras como *i*, e constantes inteiras.

Este programa em particular utiliza propriedades da representação em caractere dos dígitos. Por exemplo, o teste

```
if (c >= '0' && c <= '9') ...
```

determina se o caractere em *c* é um dígito. Se for, o valor numérico do mesmo é

```
c -'0'
```

Isso funciona somente se `'0'`, `'1'`, ..., `'9'` são positivos, em ordem crescente e consecutivos. Felizmente, isso acontece em todos os conjuntos de caracteres.

Por definição, *chars* são apenas inteiros pequenos, de modo que as variáveis e constantes do tipo **char** são idênticas a *ints* em expressões. Isso é natural e conveniente; por exemplo, `c - '0'` é uma expressão inteira com um valor entre 0 e 9 correspondente ao caractere de `'0'` a `'9'` armazenado em *c*, e é assim um subscrito válido para o vetor *ndigito*.

**Adendo 1.6.2 — O que significa a expressão c-'0'?.** Os caracteres em **C** são representados por inteiros pequenos que correspondem ao conjunto de caracteres usados na sua máquina. No

conjunto de caracteres ASCII, o caractere `'A'` possui código numérico 65 enquanto o '0' (zero) possui código 48. O caractere `'9'` possui código 57. Desta forma, quando *C* faz `c-'0'`, e o caractere lido foi '9', a conta feita é $57 - 48$ que é igual ao número 9. Utilizar expressão do tipo `c-'0'` garante que o código é independente da máquina em que o programa está sendo compilado, pois mesmo que ela não utilize o conjunto **ASCII**, a diferença entre os caracteres '9' e '0' vai ser sempre igual a 9.

A decisão que verifica se um caractere é um dígito, espaço em branco ou algo mais é feita com a sequência:

```
if (c >= '0' && c <= '9'){
   ++ndigito[c-'0'];
} else if (c == ' ' || c == '\n' || c == '\t'){
   ++nbranco;
} else
   ++ noutro;
```

O padrão

```
 if (condição_1)
   comando_1
 else if (condição_2)
   comando_2
   -
   -
 else
   comando_n
```

ocorre frequentemente em programas como uma forma de expressar decisões múltiplas. As condições são avaliadas na ordem a partir do topo até que alguma condição seja satisfeita; nesse ponto, a parte de comando correspondente é executada, e a construção inteira é acabada. (Qualquer comando pode ser diversos comandos delimitados por chaves.) Se nenhuma das condições for satisfeita, o comando após o **else** final é executado, se estiver presente. Se o **else** e o comando finais forem omitidos, como no programa de contagem de palavras, nenhuma ação é tomada. Pode haver um número qualquer de grupos de

```
else if (condição)
     comando
```

entre o **if** inicial e o **else** final.

Por questão de estilo, aconselha-se a formatação desta construção conforme mostramos; se cada **if** fosse indentado após o **else** anterior, uma longa sequência de decisões certamente esbarraria na margem direita da página.

O comando **switch**, que será discutido no Capítulo 3, provê uma outra forma de escrever um desvio múltiplo, particularmente apropriada quando a condição a ser testada é a de ver se alguma expressão do tipo inteiro ou caractere casa com um elemento de um conjunto de valores constantes. Para fins de comparação, apresentamos uma versão **switch** deste programa na Seção 3.4.

**Exercício 1.13** Escreva um programa para imprimir um histograma do tamanho das palavras da entrada. É mais fácil desenhar o histograma com barras horizontais; uma representação vertical é mais desafiadora. ■

**Exercício 1.14** Escreva um programa que imprima um histograma das frequências de diferentes caracteres na sua entrada. ■

## 1.7 Funções

Em **C**, uma função é equivalente a uma sub-rotina ou função em **Python** ou **Fortran**, ou a uma *procedure* ou função em **Pascal**. Uma função fornece um meio conveniente de encapsular alguma computação, que pode ser depois utilizada sem a preocupação com detalhes de implementação. Com funções projetadas adequadamente, é possível ignorar *como* uma tarefa é feita; saber *o que é* feito é suficiente. **C** torna o uso de funções fácil, conveniente e eficiente; você vai ver, frequentemente, uma função com poucas linhas e ativada uma vez, só porque ela esclarece alguma parte do código.

Até agora, nós usamos funções como *printf*, *getchar* e *putchar* que nos foram fornecidas; agora é hora de escrevermos algumas nós mesmos. Como **C** não possui operador de exponenciação como o $**$ do Fortran, vamos ilustrar o mecanismo de definição de função, escrevendo uma função *pot(m,n)* que eleva um inteiro *m* à potência positiva inteira *n*. Isto é, o valor de *pot(2,5)* é 32. Esta função, certamente, não é uma rotina prática para exponenciação, já que só manipula potências positivas de pequenos inteiros, mas serve bem para fins de ilustração. (A biblioteca-padrão contém uma função *pow(x,y)*, que calcula $x$ elevado a $y$). Aqui está a função *pot* e um programa principal para testá-la, de modo que você possa ver a estrutura inteira de uma vez.

Programa 1.13: Exponenciação.

```
#include <stdio.h>

int pot (int m, int n);

/* testa função pot */
int main(void)
{
    for (int i = 0; i < 10; ++i) {
        printf ("%d %d %d\n", i, pot (2, i), pot (-3, i));
    }

    return 0;
}

/* pot: eleva base a enésima potência; n >= 0 */
int pot (int base, int n)
{
    int p = 1;

    for (int i = 1; i <= n; ++i) {
        p = p * base;
    }

    return p;
}
```

Cada definição de função tem esta forma:

```
tipo_retorno nome_função(declarações de parâmetros, se algum)
{
  declarações
  comandos
}
```

As declarações de função podem aparecer em qualquer ordem, e em um arquivo-fonte ou diversos, embora nenhuma função possa estar dividida em dois arquivos. Se o programa-fonte

aparecer em diversos arquivos, você pode ter que dar mais informações para compilá-lo e carregá-lo do que se aparecesse somente em um, mas isso diz respeito ao sistema operacional, e não é um atributo da linguagem. Por ora, presumimos que as duas funções estão no mesmo arquivo, de modo que tudo o que foi aprendido sobre executar programas em **C** ainda é válido.

A função *pot* é chamada duas vezes por *main*, na linha

```
        printf ("%d %d %d\n", i, pot (2, i), pot (-3, i));
```

Cada chamada passa dois argumentos a *pot*, que a cada vez retorna um inteiro a ser formatado e impresso. Em uma expressão, *pot(2,i)* é um inteiro assim como 2 e *i* o são. (Nem toda função produz um valor inteiro; veremos isso no Capítulo 4.)

A primeira linha da função *pot*,

```
int pot (int base, int n)
```

declara os tipos e nomes dos parâmetros, e o tipo do resultado que a função retorna. Os nomes usados por *pot* para seus parâmetros são locais a *pot*, não sendo visíveis a qualquer outra função: outras rotinas podem usar os mesmos nomes sem conflito. Isso também é verdadeiro para as variáveis *i* e *p*: o *i* em *pot* não está relacionado ao *i* em *main*.

Geralmente usaremos *parâmetro* para uma variável nomeada na lista entre parênteses numa definição de função, e *argumento* para o valor usado em uma chamada da função. Os termos *argumento formal* e *argumento real* são às vezes usados para esta mesma distinção.

O valor que *pot* calcula é retornado a *main* por meio de um comando *return*. Qualquer expressão pode vir após o *return*:

```
  return expressão;
```

Uma função não precisa retornar um valor; um comando **return** sem expressão faz com que o controle, mas não um valor útil, seja retornado a quem o chama, como se houvesse encontrado o fim da função no fecha-chave final. Uma função de cálculo pode ignorar um valor retornado por uma função.

Você pode ter notado que há um comando **return** ao final de *main*. Como *main* é uma função como outra qualquer, ela pode retornar um valor a quem a chama, que é o próprio ambiente em que o programa foi executado. Normalmente, um valor zero no retorno indica término normal; valores *não-zero* sinalizam condições de término incomuns ou erros. Por questão de simplicidade, poderíamos ter omitido os comandos **return** de nossas funções *main* até este ponto, mas as incluímos, como aviso de que os programas devem retornar um estado ao seu ambiente.

A declaração

```
int pot (int m, int n);
```

logo antes de *main* diz que *pot* é uma função que espera dois argumentos **int** e retorna um **int**. Esta declaração, que é chamada **protótipo de função**, deve combinar com a definição e usos de *pot*. Será considerado erro se a definição de uma função ou qualquer local de uso não combinar com seu protótipo.

Os nomes de parâmetros não precisam combinar. Na verdade, os nomes são opcionais em um protótipo de função, de modo que, para o protótipo, poderíamos ter escrito

```
int pot (int, int);
```

No entanto, nomes bem escolhidos são uma boa documentação, de modo que normalmente iremos utilizá-los.

Uma breve nota de história: A maior mudança entre o **ANSI C** e as versões anteriores é a forma com que as funções são declaradas e definidas. Na definição original de **C**, a função *pot* teria sido escrita assim:

Programa 1.14: Declaração de função, método antigo.

```
/* pot: eleva base a enésima potência: n >= 0 */
/* (versão antiga) */
pot (base, n)
int base, n;
{
  int i, p;

  p = 1;
  for (i = 1; i <= n; ++i)
      p = p * base;
  return p;
}
```

Os parâmetros são nomeados entre parênteses, e seus tipos são declarados antes do abre-chave; os parâmetros não declarados são tomados como *int*. (O corpo da função é o mesmo que antes.)

A declaração de *pot* no início do programa teria sido da seguinte forma:

```
int pot();
```

Nenhuma lista de parâmetro era permitida, de modo que o compilador não poderia checar prontamente se *pot* estava sendo chamado corretamente. Na verdade, como por padrão *pot* presumidamente retornaria um **int**, a declaração inteira poderia também ser omitida.

A nova sintaxe para protótipos de função facilita bastante a detecção de erros pelo compilador quanto ao número de argumentos ou seus tipos. O velho estilo de declaração e definição ainda funciona no **ANSI C**, pelo menos por um período de transição, mas recomendamos que você use a nova forma quando tiver um compilador que a suporte.

**Exercício 1.15** Reescreva o programa de conversão de temperatura da Seção 1.2, de modo que use uma função para a conversão. ■

## 1.8 Argumentos – chamada por Valor

Um dos aspectos das funções em **C** pode não ser familiar a programadores que usem outras linguagens, particularmente **Fortran**. Em **C**, todos os argumentos são passados "*por valor*". Isso significa que, à função chamada, é dada uma **cópia** dos valores de seus argumentos em variáveis temporárias ao invés de seus originais. Isso leva a algumas propriedades diferentes das vistas em linguagens que trabalham com "chamadas por referência", como **Fortran**, ou com parâmetros *var* em **Pascal**, onde a rotina chamada tem acesso ao argumento original, e não a uma cópia local.

A diferença principal é que, em **C**, a função chamada não pode alterar o valor de uma variável da função que chama; ela só pode alterar sua cópia temporária, particular.

A chamada por valor, entretanto, é uma vantagem, e não uma desvantagem. Leva normalmente a programas mais compactos com poucas variáveis adicionais, porque os parâmetros podem ser tratados como variáveis locais convenientemente inicializadas na rotina chamada. Por exemplo, segue uma versão de *pot* que faz uso dessa propriedade.

Programa 1.15: Exponenciação, versão 2.

```
/*pot: eleva a enésima potência; n>=0; versão 2 */
int pot (int base, int n)
{
    int p;

    for (p = 1; n > 0; --n) {
```

```
        p = p * base;
    }

    return p;
}
```

O parâmetro *n* é usado como variável temporária, sendo decrementada (um laço **for** que trabalha ao contrário) até se tornar zero; não há mais necessidade da variável *i*. Tudo o que for feito com *n* dentro de *pot* não tem efeito no argumento com que *pot* foi originalmente chamado. Quando necessário, é possível permitir a uma função modificar uma variável na rotina que chama. Quem chama deve fornecer o endereço da variável a ser alterada (tecnicamente, um ponteiro para a variável), e a função chamada deve declarar que o parâmetro é um ponteiro e deve referenciar indiretamente o valor atual da variável. Veremos os ponteiros em detalhes no Capítulo 5.

A coisa é diferente para vetores. Quando o nome de um vetor é usado como argumento, o valor passado à função é o local ou endereço do início do vetor — não há cópia de elementos do vetor. Pelo uso deste valor com subscritos, a função pode acessar e alterar qualquer elemento do vetor. Este é o assunto da próxima seção.

## 1.9 Vetores de Caracteres

Provavelmente, o tipo mais comum de vetor em C é o de caracteres. Para ilustrar o uso de vetores de caracteres, e funções que os manipulam, vamos escrever um programa que lê um conjunto de linhas e imprime a maior delas. O algoritmo básico é simples:

```
while (há outra linha),
  if (ela é maior que a anterior)
     salve-a
     salve seu tamanho
imprime a maior linha
```

Este algoritmo deixa claro que o programa divide-se naturalmente em partes. Uma parte contém uma nova linha, outra testa-a, outra salva-a e o resto controla o processo.

Visto que as coisas dividem-se tão facilmente, seria bom escrevê-las desta forma também. Portanto, vamos primeiro escrever uma função *lelinha* separada para obter a próxima linha da entrada. Tentaremos tornar a função útil em outros contextos. No mínimo *lelinha* deve retornar um sinal sobre um possível *fim de arquivo*; um projeto mais útil seria retornar o tamanho da linha, ou zero se fosse encontrado o fim de arquivo. **Zero** é um retorno de fim de arquivo aceitável porque nunca é um tamanho de linha válido. Toda linha tem pelo menos um caractere — o de nova-linha.

Quando encontrarmos uma linha que é maior que a maior linha anterior, ela deve ser guardada em algum lugar. Isso sugere uma segunda função, *copia*, para copiar a nova linha para um lugar seguro; finalmente, precisamos de um programa principal para controlar *lelinha* e *copia*. E aqui está o resultado.

Programa 1.16: Imprime a maior linha.

```
#include <stdio.h>
#define MAXLINHA 1000 //tamanho máximo da linha entrada

int lelinha (char linha[], int maxlinha);
void copia (char para[], char de[]);

/* imprime maior linha entrada */
int main(void)
```

```
{
    int tam;               // tamanho atual da linha
    int max = 0;           // tamanho máximo visto até agora
    char linha [MAXLINHA]; // linha atual
    char maior [MAXLINHA]; // maior linha guardada

    while ((tam = lelinha (linha, MAXLINHA)) > 0) {
        if (tam > max) {
            max = tam;
            copia (maior, linha);
        }
    }

    if (max > 0) { // entrada tinha uma linha
        printf ("%s", maior);
    }

    return 0;
}

/* lelinha: lê uma linha em s, retorna tamanho */
int lelinha (char s[], int lim)
{
    int c, i;

    for(i = 0; i < lim - 1 && (c = getchar()) != '\n' && c != EOF; ++i) {
        s[i] = c;
    }

    if (c == '\n') {
        s[i] = c;
        ++i;
    }

    s[i] = '\0';
    return i;
}

/* copia: copia 'de' -> 'para';
   presume que 'para' é grande o suficiente */
void copia (char para[], char de[])
{
    int i = 0;

    while ((para[i] = de[i]) != '\0') {
        ++i;
    }
}
```

As funções *lelinha* e *copia* são declaradas no início do programa, que supostamente está contido em um único arquivo.

*main* e *lelinha* comunicam-se por meio de um par de argumentos e um valor retornado. Em *lelinha*, os argumentos são declarados pela linha

```
int lelinha (char s[], int lim)
```

especificando que o primeiro parâmetro, *s*, é um vetor, e o segundo, *lim*, é um inteiro. A finalidade de fornecer o tamanho de um vetor em uma declaração é separar memória. O tamanho do vetor *s* não é necessário em *lelinha*, pois seu tamanho é definido em *main*. *lelinha* usa **return** para enviar um valor de volta a quem o chama, assim como na função *pot*. A linha também declara que *lelinha* retorna um valor **int**; como **int** é o tipo de retorno **default**, poderia ser omitido.

Algumas funções retornam um valor útil; outras, como *copia*, são usadas somente pelo seu efeito e não retornam um valor. O tipo de retorno de copia é **void**, dizendo explicitamente que **nenhum** valor é retornado.

*lelinha* coloca o caractere `'\0'` (o caractere nulo, cujo valor é zero) ao final do vetor, para marcar o fim da cadeia de caracteres. Esta convenção é também usada pelo compilador **C** : quando uma constante do tipo string tal como

```
"hello\n"
```

aparece em um programa **C**, o compilador cria um vetor de caracteres contendo os caracteres de string, terminando com um `'\0'` para marcar o fim.

| h | e | l | l | o | \n | \0 |
|---|---|---|---|---|---|---|

A especificação de formato %s em *printf* espera que o argumento correspondente seja uma **string** representada dessa forma. *copia* também se baseia no fato de que seu argumento de entrada é terminado por `'\0'`, e copia este caractere para o argumento de saída. (Tudo isso significa que `'\0'` não faz parte do texto normal.)

Vale a pena mencionarmos que, mesmo um programa tão pequeno como este, apresenta alguns problemas de projeto. Por exemplo, o que *main* faria se encontrasse uma linha maior do que seu limite? *lelinha* funciona de forma segura, pois ela interrompe a coleta de caracteres quando o vetor está cheio, mesmo que não tenha sido visto um caractere de nova-linha. Testando o tamanho e o último caractere retornado, *main* pode determinar se a linha foi muito grande, e então prosseguir da forma desejada. Para abreviar o programa, ignoramos este caso.

Um usuário de *lelinha* não tem como saber, antecipadamente, o tamanho que uma linha de entrada pode ter; portanto *lelinha* verifica o estouro. Por outro lado, o usuário de *copia* já conhece (ou pode conhecer) o tamanho das cadeias, de modo que escolhemos não incluir a verificação de erro na função.

**Exercício 1.16** Revise a rotina *lelinha* do programa que imprime a maior linha para que imprima corretamente o tamanho de linhas de entrada arbitrariamente longas, e a maior quantidade de texto possível da mesma. ■

**Exercício 1.17** Escreva um programa que imprima todas as linhas de entrada com mais de **80 caracteres**. ■

**Exercício 1.18** Escreva um programa que remova espaços em branco e caracteres de tabulação finais de cada linha de entrada e que delete as linhas inteiramente em branco. ■

**Exercício 1.19** Escreva uma função *inverte(s)* que inverta a ordem dos caracteres da string *s*. Use-a para escrever um programa que inverta sua entrada, linha a linha. ■

## 1.10 Variáveis Externas e Escopo

As variáveis em *main*, como *linha*, *maior*, etc., são privativas ou locais a *main*. Visto que elas são declaradas em *main*, nenhuma outra função tem acesso direto a elas. O mesmo é verdadeiro

para variáveis de outras funções; por exemplo, a variável *i* em *lelinha* não tem relação alguma com *i* em *copia*. Cada variável local numa rotina passa a existir somente quando a função é chamada, e deixa de existir somente quando a função termina. Por essa razão, tais variáveis são conhecidas normalmente como **variáveis automáticas**, seguindo a terminologia de outras linguagens. Usaremos o termo automática daqui para a frente para nos referirmos a essas **variáveis locais dinâmicas**. (O Capítulo 2 discute a classe de armazenamento **static**, onde variáveis locais retêm seus valores entre ativações da função.)

Devido às variáveis automáticas virem e irem com a chamada da função, elas não retêm seus valores de uma chamada para outra, e devem ser explicitamente inicializadas a cada chamada. Se elas não forem iniciadas, conterão lixo.

Como uma alternativa para as variáveis automáticas, é possível definir variáveis que sejam **externas** a todas as funções, isto é, variáveis globais que possam ser acessadas pelo nome por qualquer função que assim o desejar. (Este mecanismo é semelhante ao *COMMON* do **Fortran** ou às variáveis do **Pascal** declaradas no bloco mais externo.) Visto que as variáveis externas sejam globalmente acessíveis, podem ser usadas, ao invés de listas de argumentos, para comunicar dados entre funções. Além do mais, devido às variáveis externas existirem permanentemente, ao invés de aparecerem com a ativação e desativação de funções, elas retêm seus valores mesmo quando as funções que as acessam retornam.

Uma **variável externa** deve ser definida, exatamente uma vez, fora de qualquer função; isso aloca armazenamento para ela. A variável deve também ser declarada em cada função que desejar acessá-la; isso pode ser feito por uma declaração explícita **extern** ou implicitamente pelo contexto. Para tornarmos a discussão concreta, vamos reescrever o programa que imprima a maior linha com *linha*, *maior* e *max* sendo variáveis externas. Isso requer mudanças nas chamadas, declarações e corpos de todas as três funções.

Programa 1.17: Imprime a maior linha com variáveis externas.

```
#include <stdio.h>

#define MAXLINHA 1000  // tamanho máximo da linha

int max;               // tamanho visto até agora
char linha [MAXLINHA]; // linha de entrada
char maior [MAXLINHA]; // linha mais longa

int lelinha (void);
void copia (void) ;

/* imprime maior linha entrada; versão especializada */
int main(void)
{
    int tam;
    extern int max;
    extern char maior [];
    max = 0;

    while ((tam = lelinha ()) > 0) {
        if (tam > max) {
            max = tam;
            copia ();
        }
    }

    if (max > 0) { // entrada tinha uma linha
```

```
        printf ("%s", maior);
    }

    return 0;
}

/* lelinha: versão especializada */
int lelinha (void)
{
    int c, i = 0;
    extern char linha [];

    for (; i < MAXLINHA - 1 && (c = getchar())!='\n' && c!=EOF; ++i) {
        linha [i] = c;
    }

    if (c == '\n') {
        linha [i] = c;
        ++i;
    }

    linha [i] = '\0';
    return i;
}

/* copia: versão especializada */
void copia (void)
{
    int i = 0;
    extern char linha [], maior[];

    while ((maior [i] = linha [i]) != '\0') {
        ++i;
    }
}
```

As variáveis externas em *main*, *lelinha* e *copia* são definidas pelas primeiras linhas do exemplo acima, que definem seu tipo e alocam área de armazenamento para as mesmas. Sintaticamente, definições externas são iguais às declarações das variáveis locais, mas como ocorrem fora de funções, as variáveis são **externas**. Antes que uma função possa usar uma variável externa, o nome da variável deve tornar-se conhecido pela função. Uma maneira de fazer isso é escrever uma declaração **extern** na função; a declaração é igual à anterior, exceto que é precedida da palavra-chave **extern**.

Em certas circunstâncias, a declaração **extern** pode ser omitida. Se a definição de uma variável **extern** ocorrer no arquivo-fonte antes do seu uso em uma função em particular, então não há necessidade de uma declaração **extern** na função. As declarações **extern** em *main*, *lelinha* e *copia* são, portanto, redundantes. De fato, a prática comum é colocar a definição de todas as variáveis externas no início do arquivo-fonte e, então, omitir todas as declarações **extern**.

Se o programa está em vários arquivos-fonte, e uma variável é definida em *arquivo1* e usada em *arquivo2* e *arquivo3*, então as declarações **extern** são necessárias em *arquivo2* e *arquivo3* para conectar as ocorrências da variável. A prática normal é coletar declarações **extern** de variáveis e funções em um arquivo separado, historicamente conhecido como *header* (cabeçalho), que é incluído por um **#include** no início de cada arquivo-fonte. Convencionou-se o sufixo *.h* para os arquivos de cabeçalho. As funções da biblioteca-padrão, por exemplo, são declaradas em cabeçalhos

como **<stdio.h>**. Esse tópico é discutido mais detalhadamente no Capítulo 4, e a própria biblioteca no Capítulo 7 e Apêndice B.

Visto que as versões especializadas de *lelinha* e *copia* não possuem parâmetros, a lógica sugeriria que seus protótipos no início do arquivo fossem $lelinha()$ e $copia()$. Mas, por compatibilidade com programas **C** mais antigos, o padrão toma uma lista vazia como uma declaração ao estilo antigo, e desativa toda a checagem da lista de argumentos; a palavra **void** deve ser usada para uma lista explicitamente vazia. Falaremos mais sobre isso no Capítulo 4.

Você deve observar que estamos usando as palavras **definição** e **declaração** cuidadosamente quando nos referimos a variáveis externas nesta seção. "**Definição**" refere-se ao lugar onde a variável é declarada ou à qual é atribuída uma área de armazenamento; "**declaração**" refere-se ao local onde a natureza da variável é dada, sem alocação de área de armazenamento.

A propósito, há uma tendência de se fazer com que tudo em vista seja uma variável **extern** porque simplifica aparentemente a comunicação – listas de argumentos são curtas e as variáveis estão sempre onde você as quer. Porém variáveis externas estão sempre lá, mesmo quando você não as quer. Basear-se muito em variáveis externas é perigoso, pois leva a programas cujas conexões de dados não são óbvias — variáveis podem ser alteradas de formas não esperadas e inadvertidamente, e o programa é difícil de modificar se for necessário. A segunda versão do programa da maior linha é inferior à primeira, em parte por essas razões, e em parte porque elimina a generalidade de duas funções bastante úteis, por tê-las amarrado às variáveis que elas manipulam.

Neste ponto já cobrimos o que poderia ser chamado o núcleo convencional de **C**. Com este punhado de ferramentas, é possível escrever programas úteis de tamanho considerável, e provavelmente seria uma boa ideia se você parasse um tempo para fazê-lo. Estes exercícios sugerem programas um tanto mais complexos que aqueles apresentados neste capítulo.

**Exercício 1.20** Escreva um programa *destab* que troque caracteres de tabulação da entrada pelo número apropriado de espaços para atingir o próximo ponto de tabulação. Assuma um conjunto fixo de pontos de tabulação, digamos a cada $n$ colunas. Será que $n$ deve ser uma variável ou um parâmetro simbólico? ■

**Exercício 1.21** Escreva um programa *tab* que troque cadeias de brancos pelo número mínimo de caracteres de tabulação e de brancos, de forma a obter o mesmo espaçamento. Use os mesmos pontos de tabulação do programa *destab*. Quando uma tabulação ou um espaço em branco for suficiente para se atingir uma marca, qual deve receber a preferência? ■

**Exercício 1.22** Escreva um programa para "*dobrar*" linhas longas após o último caractere não branco que ocorra antes da *n-ésima* coluna de entrada, onde $n$ é um parâmetro. Faça com que seu programa funcione inteligentemente com linhas muito longas e no caso de não haver espaços ou caracteres de tabulação antes da coluna especificada. ■

**Exercício 1.23** Escreva um programa para remover todos os comentários de um programa **C**. Não se esqueça de manipular cadeias e constantes do tipo caractere entre aspas de forma correta. Os comentários em **C** não são indentados. ■

**Exercício 1.24** Escreva um programa para verificar num programa **C** erros rudimentares de sintaxe, tais como parênteses, colchetes e chaves não balanceados. Não se esqueça de aspas, apóstrofos e comentários. (Este programa é difícil, se for feito em toda generalidade.) ■

# 2. Tipos, Operadores e Expressões

Variáveis e constantes são os objetos de dados básicos manipulados em um programa. As declarações listam as variáveis a serem usadas, e definem os tipos que as mesmas devem ter e talvez seus valores iniciais. Os operadores especificam o que deve ser feito com as variáveis. As expressões combinam variáveis e constantes para produzir novos valores. Essas ferramentas são os tópicos deste capítulo.

O padrão **ANSI** fez muitas pequenas mudanças e acréscimos aos tipos e expressões básicos. Existem agora formas sinalizadas e não sinalizadas de todos os tipos inteiros, e notações para constantes não sinalizadas e constantes de caracteres hexadecimais. As operações com ponto flutuante podem ser feitas em precisão simples; há também um tipo *long double* para uma maior precisão. As constantes de cadeia de caracteres podem ser concatenadas em tempo de compilação. As enumerações tornaram-se parte da linguagem, formalizando uma característica há muito existente. Objetos podem ser declarados como *const*, evitando que sejam alterados. As regras para coerções automáticas entre tipos automáticos foram aumentadas de forma a lidar com o conjunto mais rico de tipos.

## 2.1 Nomes de Variáveis

Embora não tenhamos dito no Capítulo 1, há algumas restrições aos nomes das variáveis e constantes simbólicas. Os nomes são compostos de letras e dígitos; o primeiro caractere deve ser uma letra. O caractere de sublinha "_" é contado como se fosse uma letra; ele é útil para aumentar a clareza de nomes longos de variável. Entretanto não inicie nomes de variáveis com este caractere, pois as rotinas da biblioteca normalmente usam nomes assim. As letras maiúsculas e minúsculas são distintas, de modo que **x** e **X** são dois nomes diferentes. A prática tradicional de **C** é usar minúsculas para nomes de variáveis, e maiúsculas para constantes simbólicas.

Pelo menos os **31 primeiros caracteres** de um nome interno são significativos. Para nomes de função e variáveis externas, o número máximo pode ser menor que 31, pois os nomes externos podem ser usados por *assemblers* e carregadores sob os quais a linguagem não tem controle. Para nomes externos, o padrão garante singularidade somente para **6 caracteres** e um único tipo de letra (maiúscula ou minúscula), ou seja, não é sensível ao *case* da letra. As palavras-chave como $if$, *else*,

*int*, *float*, etc., são reservadas: você não pode usá-las como nomes de variável. Elas devem estar em minúsculas.

C99 Em C99 os primeiros **63 caracteres** dos identificadores internos são significativos. Para nomes externos, os primeiros **31 caracteres** são significativos e letras maiúsculas são diferentes das minúsculas (*case sensitive*).

Aconselha-se a escolha de nomes de variáveis que estejam relacionados à finalidade da variável, e que provavelmente não serão misturadas tipograficamente. Costumamos usar nomes curtos para variáveis locais, especialmente índices de laço, e nomes maiores para variáveis externas.

## 2.2 Tipos de Dados e Tamanhos

Há poucos tipos básicos de dados em **C** :
**char** um único byte, capaz de conter um caractere no conjunto de caracteres local.
**int** um inteiro, normalmente refletindo o tamanho natural dos inteiros da máquina hospedeira;
**float** ponto flutuante em precisão simples.
**double** ponto flutuante em dupla precisão.

Além do mais, há uma série de *qua1ificadores* que podem ser aplicados a estes tipos básicos. *short* e *long* referem-se a tamanhos de inteiros:

```
short int x;
long int contador;
```

A palavra *int* pode omitida nessas declarações, e normalmente o é.

A intenção é que *short* e *long* devam prover tamanhos diferentes de inteiros onde isso for prático; *int* normalmente será o tamanho para uma determinada máquina. *short* ocupa normalmente **16 bits**, *long* **32 bits**, e *int*, **16** ou **32 bits**. Cada compilador é livre para escolher tamanhos adequados para o seu próprio hardware, com a única restrição de que *shorts* e *ints* devem ocupar pelo menos **16 bits**, *longs* pelo menos **32 bits**, e *short* não pode ser maior que *int*, que não é maior do que *long*.

C99 C99 inclui o tipo *long long int* com um tamanho mínimo de **8 bytes**.

O qualificador *signed* ou *unsigned* pode ser aplicado a *char* ou a qualquer inteiro. Números *unsigned* são sempre positivos ou **0**, e obedecem às leis aritméticas de **2 elevado a n**, onde **n** é o número de bits no tipo. Assim, por exemplo, se *chars* ocuparem 8 bits, as variáveis do tipo *unsigned char* possuem valores entre 0 e 255, enquanto as variáveis *signed char*, entre -128 e 127 (em uma máquina que use o complemento a dois). Se os *chars* simples são sinalizados ou não, depende da máquina, mas os caracteres que podem ser impressos são sempre positivos.

O tipo *long double* especifica ponto flutuante com precisão estendida. Assim como os inteiros, os tamanhos de objetos em ponto flutuante são definidos pela implementação; *float*, *double* e *long double* poderiam representar um, dois ou três tamanhos distintos.

Os cabeçalhos-padrão **<limits.h>** e **<float.h>** contêm constantes simbólicas para todos esses tamanhos, juntamente com outras da máquina e compilador. Estas são discutidas no Apêndice B.

**Exercício 2.1** Escreva um programa para determinar as faixas de variáveis *char*, *short*, *int*, *long* e *long long*, tanto *signed* quanto *unsigned*, imprimindo os valores apropriados dos cabeçalhos-padrão e por cálculo direto. Mais difícil se você os calcula: determine as faixas dos vários tipos de ponto-flutuante. ■

## 2.3 Constantes

Uma constante inteira como 1234 é um *int*. Uma constante *long* é escrita com um **l** (letra éle) ou **L** ao final, como em 123456789*L*; um inteiro muito grande para caber em um *int* também será considerado como *long*. Constantes não sinalizadas são escritas com uma letra **u** ou **U** ao final, e o sufixo **ul** ou **UL** indica *unsigned long* (longo não sinalizado).

As constantes de ponto flutuante contêm um ponto decimal (**123.4**) ou um expoente (**1e-2**) ou ambos; seu tipo é *double*, a menos que tenham um sufixo. Os sufixos **f** ou **F** indicam uma constante *float*; **l** ou **L** indicam um *long double*.

O valor de um inteiro pode ser especificado em **octal** ou **hexadecimal** ao invés de decimal. Um **0** (zero) inicial em uma constante inteira significa **octal**; um **0x** ou **0X** inicial significa **hexadecimal**. Por exemplo, o decimal **31** pode ser escrito como **037** em octal e **0x1f** ou **0X1F** em hexa. As constantes **octais** e **hexadecimais** podem ser também seguidas por **L** para torná-las *long* e **U** para torná-las *unsigned*: **0XFUL** é uma constante longa não sinalizada com o valor 15 decimal.

Uma constante de caractere é um inteiro, escrito como um caractere entre aspas simples, como em **'x'**. O valor de uma constante de caractere é o valor numérico do caractere no conjunto de caracteres da máquina. Por exemplo, no conjunto de caracteres **ASCII**, a constante de caractere **'0'** tem o valor 48, que não está relacionado ao valor numérico 0. Se escrevermos **'0'** no lugar de um valor numérico como 48, que depende do conjunto de caractere, o programa se toma independente do valor em particular e mais fácil de ser lido. As constantes de caractere participam em operações numéricas assim como quaisquer outros inteiros, embora sejam mais comumente usadas em comparações com outros caracteres.

Certos caracteres podem ser representados em constantes de caractere e em cadeias de caracteres por sequências de escape como \n (nova-linha); essas sequências se parecem com dois caracteres, mas representam apenas um. Além do mais, um padrão de bit arbitrário, no tamanho de um byte, pode ser especificado por

```
'\ooo'
```

onde ooo é de um a três dígitos octais (0...7) ou por

```
'\xhh''
```

onde *hh* é um ou mais dígitos hexadecimais (0...9, *a*...*f*, *A*...*F*). Assim, poderíamos escrever,

```
#define VTAB '\013' // tabulação vertical em ASCII/
#define BELL '\007' // caractere de alerta em ASCII
```

ou, em hexadecimal,

```
#define VTAB '\xb' // tabulação vertical em ASCII
#define BELL '\x7' // caractere de alerta em ASCII
```

O conjunto completo de sequências de escape é

| | | | |
|---|---|---|---|
| \a | caractere de alerta (bell) | \\ | contrabarra |
| \b | retrocesso | \? | Ponto de interrogação |
| \f | alimentação de formulário | \' | apóstrofo |
| \n | nova-linha | \" | aspas |
| \r | retorno de carro | \ooo | número octal |
| \t | tabulação horizontal | \xhh | número hexadecimal |
| \v | tabulação vertical | | |

A constante de caractere **'\0'** representa o caractere com o valor zero, o caractere nulo. **'\0'** é normalmente escrito no lugar de zero para enfatizar a natureza de caractere de alguma expressão, mas os valor numérico é apenas 0.

Uma expressão constante é uma expressão que envolve somente constantes. Tais expressões são avaliadas em tempo de compilação, e não em tempo de execução, e podem ser usadas onde uma constante possa aparecer, tal como em

```
#define MAXLINHA 1000
char linha [MAXLINHA + 1];
```

ou

```
#define BISSEXTO 1 // em anos bissextos
int dias[31+28+BISSEXTO+31+30+31+30+31+31+30+31+30+31];
```

Uma constante do tipo **string**, ou literal do tipo string, é uma sequência de zero ou mais caracteres delimitados por aspas, tal como

```
"Eu sou uma cadeia de caracteres"
```

ou

```
"" // uma string vazia ou nula
```

As aspas não fazem parte da string, mas servem apenas para delimitá-la. As mesmas sequências de escape usadas em constantes aplicam-se às cadeias; \" representa o caractere de aspas. As constantes do tipo string podem ser concatenadas em tempo de compilação:

```
"primeiro " "programa"
```

é equivalente a

```
"primeiro programa"
```

Isso é útil para se dividir longas cadeias em diversas linhas de código.

Tecnicamente, uma string é um vetor cujos elementos são caracteres. A representação interna de uma string possui um caractere nulo **'\0'** ao final, de modo que a área física de armazenamento necessária é um a mais que o número de caracteres escritos entre aspas. Esta representação significa que não há limite ao tamanho de uma string, mas os programas devem analisar completamente a string para determinar seu tamanho. A função da biblioteca-padrão $strlen(s)$ retorna o tamanho do seu argumento de string de caracteres $s$, excluindo o **'\0'** terminal. Aqui está nossa versão:

Programa 2.1: Função strlen()

```
/* strlen: retorna tamanho de s */
int strlen (char s[])
{
    int i;
    i = 0;

    while (s[i] != '\0') {
        ++i;
    }

    return i;
}
```

*strlen* e outras funções de string são declaradas no arquivo de cabeçalho $< string.h >$.

Tenha cuidado ao distinguir entre uma constante de caractere e uma string que contém um único caractere: **'x'** não é o mesmo que **"x"**. O primeiro é um inteiro, usado para produzir o valor numérico da letra **x** no conjunto de caracteres da máquina. O segundo é um vetor de caracteres que contém apenas um caractere (a letra **x**) e um **'\0'**.

Há um outro tipo de constante, a constante de enumeração. Uma enumeração é uma lista de valores inteiros constantes, como em

```
enum boolean {NAO, SIM};
```

O primeiro nome em uma *enum* tem o valor 0, o seguinte 1, e assim por diante, a não ser que valores explícitos sejam especificados. Se nem todos os valores forem especificados, aqueles que não foram continuam a progressão a partir do último valor especificado, como no segundo desses exemplos:

```
enum escapes{BELL='\a', RETROCESSO='\b', TAB='\t',
             NOVALINHA='\n', TABV='\v', RETORNO='\r' };

enum meses {JAN=1, FEV, MAR, ABR, MAI, JUN,
            JUL, AGO, SET, OUT, NOV, DEZ};
                /* FEV = 2, MAR = 3, etc.*/
```

Os nomes em diferentes enumerações devem ser distintos. Os valores não precisam ser distintos na mesma enumeração.

As enumerações fornecem uma forma conveniente de associar valores constantes a nomes, uma alternativa a *#define* com a vantagem dos valores serem gerados para você. Embora variáveis de tipos *enum* possam ser declaradas, os compiladores não precisam checar se o que você armazena em uma dessas variáveis é um valor válido para a enumeração. Apesar disso, as variáveis de enumeração oferecem a chance de checagem, e sendo assim, são normalmente melhores do que *define*s. Além disso, um depurador pode ser capaz de imprimir valores de variáveis de enumeração em seu formato simbólico.

## 2.4 Declarações

Todas as variáveis devem ser declaradas antes de usadas, embora certas declarações possam ser feitas implicitamente pelo contexto. Uma declaração especifica um tipo, e é seguida por uma lista de uma ou mais variáveis daquele tipo, tal como no seguinte exemplo:

```
int inicio, fim, incr;
char c, linha [1000];
```

As variáveis podem ser distribuídas entre declarações livremente; as listas acima poderiam ser igualmente escritas como

```
int inicio;
int fim;
int incr;
char c;
char linha [1000];
```

Esta última forma ocupa mais espaço, mas é mais conveniente para acrescentar um comentário a cada declaração ou para modificações subsequentes. Uma variável também pode ser inicializada na sua declaração. Se o nome for seguido por um sinal de igual e uma expressão, a expressão serve como um inicializador, como em

```
char esc = '\\';
int i = 0;
int limite = MAXLINHA + 1;
float eps = 1.0e-5
```

Se a variável em questão não for **automática**, a inicialização é feita somente uma vez, conceitualmente antes do programa começar, e o inicializador deve ser uma expressão constante. Uma variável automática inicializada explicitamente tem seu valor inicializado toda vez que a função ou bloco em que está for entrado; o inicializador pode ser qualquer expressão. As variáveis externa e estáticas são inicializadas em **zero** por *default*. As variáveis automáticas sem inicialização explícita têm valores iniciais indefinidos (i.e. , lixo).

O qualificador *const* pode ser aplicado à declaração de qualquer variável para especificar que seu valor não deve ser mudado. Para um vetor, o qualificador *const* diz que os elementos não serão alterados.

```
const  double e = 2.71828182845905;
const char msg [] = "aviso: ";
```

A declaração const também pode ser usada com argumentos de vetor, indicando que a função não altera esse vetor

```
int strlen (const char [ ]);
```

O resultado é definido pela implementação do compilador se for feita uma tentativa de alteração em uma variável *const*.

## 2.5 Operadores

Os operadores aritméticos binários são $+, -, *,$ e o operador de módulo %. A divisão inteira trunca qualquer parte fracionária. A expressão

```
x % y
```

produz o resto quando x é dividido por y, e é igual a 0 se y dividir x exatamente. Por exemplo, um ano é um ano bissexto se for divisível por 4 mas não por 100, exceto pelos anos divisíveis por 400, que são bissextos. Portanto

```
if ((ano % 4 == 0 && ano % 100 != 0) || ano % 400 == 0) {
    printf ("%d é um ano bissexto\n", ano);
} else {
    printf ("%d não é um ano bissexto\n", ano);
}
```

O operador **%** não pode ser aplicado a *float* ou *double*. A direção do truncamento para **/** e o sinal do resultado para % dependem da máquina para operandos negativos, assim como a ação tomada em caso de estouro (*overflow*) ou estouro negativo (*underflow*).

C99 O resultado do uso das operações de divisão inteira e resto (/ e %) com números negativos é confusa em C ANSI. Assim, o resultado da divisão tanto pode ser arredondado para cima ou para baixo. O padrão não especifica. Exemplo, o resultado de $-9/7$ tanto pode ser $-1$ quanto $-2$ e o resultado de $-9\%7$ tanto pode ser $-2$ quanto 5. Em C99 o resultado da divisão inteira sempre é truncado em direção a zero ($-9/7 \rightarrow -1$) e o resto sempre usa o sinal do primeiro operando ($-9\%7 \rightarrow -2$).

Os operadores binários $+$ e $-$ possuem a mesma precedência, que é menor que a precedência de $+$, $/$ e %, que por sua vez é menor que a do $+$ e $-$ unário. Os operadores aritméticos **se associam da esquerda para a direita**. A Tabela 2-1 ao final deste capítulo resume a precedência e associatividade para todos os operadores.

## 2.6 Operadores Relacionais e Lógicos

Os operadores relacionais são

```
>  >=  <  <=
```

Todos eles têm a mesma precedência. Logo acima deles em precedência estão os operadores de igualdade:

```
==   !=
```

Os operadores relacionais possuem menor precedência que os operadores aritméticos, de forma que expressões tais como $i < lim - 1$ são avaliadas como $i < (lim - 1)$, como seria esperado.

Mais interessantes são os operadores lógicos && e ||. Expressões conectadas por && e || são avaliadas da esquerda para a direita, e a avaliação é interrompida assim que a veracidade ou falsidade do resultado for conhecida. A maioria dos programas em **C** baseia-se nessas propriedades. Por exemplo, aqui está um laço da função *lelinha* que nós escrevemos no Capítulo 1:

```
for(i = 0; i < lim - 1 && (c = getchar()) != '\n' && c != EOF; ++i)
```

Antes de lermos um novo caractere, é necessário verificar se há espaço para armazená-lo no vetor *s*, de modo que o teste $i < lim - 1$ deve ser feito. Além do mais, se este teste falhar, não podemos prosseguir e ler outro caractere.

De forma semelhante, seria desastroso se *c* fosse testado contra *EOF* antes de *getchar* ser chamada; a chamada e a atribuição devem ocorrer antes que o caractere *c* seja testado.

A precedência de && é maior que a de ||, e ambas têm menor precedência que os operandos relacionais e de igualdade, de forma que expressões tais como:

```
i<lim-1 && (c = getchar())!='\n' && c!=EOF
```

não necessitam de parênteses extras. Mas desde que a precedência **!=** é maior que a da atribuição, parênteses são necessários em:

```
(c = getchar ()) != '\n'
```

para obter o resultado desejado da atribuição a *c* e depois a comparação com `'\n'`. Por definição, o valor numérico de uma expressão relacional ou lógica é 1 se a relação for verdadeira, e 0 se a relação for falsa. O operador unário de negação ! converte um operando diferente de zero ou verdadeiro no valor **0**, e um operando **0** ou **falso** no valor **1**. Uma utilidade comum de ! está em construções do tipo

```
if (!valido)
```

ao invés de

```
if (valido == 0)
```

É difícil generalizar sobre qual forma é melhor. Construções tais como !*valido* são mais agradáveis de se ler ("se não válido"), mas construções mais complicadas podem se tomar muito difíceis de se entender.

**Exercício 2.2** Escreva um laço equivalente ao laço for acima sem usar && ou ||. ■

## 2.7 Conversões de Tipo

Quando um operador possui operandos de tipos diferentes, eles são convertidos para um tipo comum de acordo com um pequeno número de regras. Em geral, as únicas conversões automáticas são

aquelas que convertem um operador "mais estreito" em um "mais largo" sem perder informações, como na conversão de um inteiro para ponto flutuante em uma expressão do tipo $f + i$. Expressões que não fazem sentido, como usar um $float$ como um subscrito, não são permitidas. Expressões que poderiam perder informação, como atribuir um tipo inteiro mais longo a um mais curto, ou um tipo de ponto flutuante a um inteiro, podem gerar um aviso, mas continuam sendo válidas.

Um tipo *char* é somente um pequeno inteiro, de modo que *char*s podem ser livremente usados em expressões aritméticas. Isso permite uma grande flexibilidade em certos tipos de transformações de caracteres. Uma delas é exemplificada por esta simples implementação da função *atoi*, que converte uma *string* de dígitos no seu valor numérico correspondente.

Programa 2.2: Função atoi()

```
/* atoi: converte s em um inteiro */
int atoi (char s [])
{
    int n = 0;

    for (int i = 0; s[i] >= '0' && s[i] <= '9'; ++i) {
        n = 10 * n + (s[i] - '0');
    }

    return n;
}
```

Como falamos no Capítulo 1, a expressão

```
  s[i]-'0'
```

dá o valor numérico do caractere armazenado em $s[i]$, pois os valores de '0', '1', etc. formam uma sequência continuamente crescente.

Um outro exemplo de conversão de *char* para *int* é a função *lower*, que mapeia um caractere para minúsculo considerando-se apenas o conjunto de caracteres **ASCII**. Se o caractere não é maiúsculo, *lower* o retorna inalterado.

Programa 2.3: Função lower()

```
/* lower: converte c para minúsculo; só para ASCII */
int lower (int c)
{
    if (c >= 'A' && c <= 'Z') {
        return c + 'a' - 'A';
    }

    return c;
}
```

Isso funciona para **ASCII** porque as letras maiúsculas e minúsculas correspondentes estão a uma distância numérica fixa, e cada alfabeto é contíguo — não há nada exceto letras entre **A** e **Z**. Esta última observação, entretanto, não é verdadeira para o conjunto de caracteres **EBCDIC**, de forma que este código converteria mais do que apenas letras em **EBCDIC**.

O cabeçalho-padrão $< ctype.h >$, descrito no Apêndice B, define uma família de funções que fazem testes e conversões independentes do conjunto de caracteres. Por exemplo, a função $tolower(c)$ retorna o valor minúsculo de $c$ se $c$ for maiúsculo, de forma que $tolower$ é um substituto portátil para a função minúsculo mostrada acima. Da mesma forma, o teste

```
  c >= '0' && c <= '9'
```

pode ser substituído por

```
isdigit(c)
```

Usaremos as funções de $< ctype.h >$ daqui para frente.

Há um ponto delicado sobre a conversão de caracteres para inteiros. A linguagem não especifica se variáveis do tipo *char* são valores com ou sem sinal. Quando um *char* é convertido para um *int*, pode-se produzir um inteiro negativo? A resposta varia de acordo com a máquina, refletindo as diferenças na arquitetura. Em algumas máquinas um *char* cujo bit da esquerda é 1 será convertido para um inteiro negativo ( "extensão de sinal"). Em outras, um *char* é promovido para um *int* incluindo-se zeros no extremo esquerdo, e é então sempre positivo.

A definição de **C** garante que qualquer caractere no conjunto de caracteres-padrão da máquina nunca será negativo, de modo que esses caracteres podem ser usados livremente em expressões como quantidades positivas. Porém padrões arbitrários de bits armazenados em variáveis do tipo caractere podem aparecer como negativos em algumas máquinas, ainda que positivos em outras. Por questões de portabilidade, especifique *signed* ou *unsigned* se dados não-caractere puderem ser armazenados em variáveis do tipo *char*.

As expressões relacionais como $i > j$ e expressões lógicas conectadas por && e || são definidas de forma que tenham valor 1 se verdadeiras, e 0 se falsas. Assim, a atribuição

```
d = c >= '0' && c <= '9'
```

atribui 1 a $d$ se $c$ for um dígito, e 0 se não. Contudo, funções como *isdigit* podem retornar qualquer valor não-zero indicando um resultado verdadeiro. Na parte de teste de $if$, *while*, $for$, etc., "verdadeiro" significa "**não-zero**", de modo que isso não fará diferença.

As conversões aritméticas implícitas funcionam conforme o esperado. Em geral, se um operador como $+$ ou $*$ que usa dois operandos (operador binário) tiver operandos de tipos diferentes, o tipo "menor" é promovido para o tipo "maior" antes que a operação prossiga. O resultado é do tipo maior. A Seção A.6 informa com exatidão as regras de conversão. Se não houver operandos *unsigned*, no entanto, o seguinte conjunto informal de regras será suficiente:

- *Se um dos operandos for long double, converte o outro para long double.*
- *Se não, se um dos operandos for double, converte o outro para double.*
- *Se não, se um dos operandos for float, converte o outro para float.*
- *Se não, converte char e short para int.*
- *Então, se um dos operandos for long, converte o outro para long.*

Observe que $floats$ em uma expressão não são automaticamente convertidos para *double*; esta é uma mudança da definição original. Em geral, as funções matemáticas como aquelas em $< math.h >$ usarão a precisão dupla. O principal motivo para o uso de $float$ é economizar memória em grandes vetores, ou, com frequência menor, ganhar tempo em máquinas onde a aritmética de dupla precisão é particularmente cara.

As regras de conversão são mais complicadas quando estão envolvidos operandos *unsigned*. O problema é que as comparações entre valores sinalizados e não sinalizados dependem da máquina, pois dependem dos tamanhos dos vários tipos de inteiros. Por exemplo, suponha que *int* ocupe 16 bits e *long*, 32. Então $-1L < 1U$, pois $1U$, que é um *unsigned int*, é promovido para *signed long*. Mas $-1L > 1UL$, porque $-1L$ é promovido para *unsigned long*, e sendo assim, parece ser um número positivo maior.

As conversões ocorrem em atribuições; o valor do lado direito é convertido para o tipo do lado esquerdo, o qual é o tipo do resultado. Um caractere é convertido para inteiro, por extensão de sinal ou não, como descrito acima.

Os inteiros mais longos são convertidos para mais curtos ou para *chars* descartando-se os bits excedentes mais significativos. Então em

```
int i;
char c;

i=c;
c=i;
```

o valor de *c* não é alterado. Isso é verdadeiro com ou sem extensão de sinal.

Se *x* é *float* e *i* é *int*, então $x = i$ e $i = x$ causam conversão; *float* para *int* causa truncamento da parte fracionária. Quando *double* é convertido para *float*, se o valor será arredondado ou truncado depende da implementação da linguagem.

Visto que o argumento de uma função é uma expressão, a conversão de tipos ocorrerá quando os argumentos forem passados para a função. Na ausência de um protótipo de função, *char* e *short* tornam-se *int*, e *float* torna-se *double*. É por isso que declaramos os argumentos da função como *int* e *double* mesmo quando a função é chamada com *char* e *float*.

Finalmente, a conversão explícita de tipos pode ser forçada ("coagida"), em qualquer expressão, com um operador unário chamado molde. Na construção:

```
(nome-de-tipo) expressão
```

a expressão é convertida para o tipo especificado pelas regras de conversão acima. O significado preciso de um molde é de fato como se a expressão fosse atribuída a uma variável do tipo especificado, que é então usada no lugar da construção. Por exemplo, a rotina de biblioteca *sqrt* espera um *double* como argumento, e dará um resultado sem sentido se outro argumento lhe for passado inadvertidamente. Assim, se *n* é um inteiro, podemos usar

```
sqrt ((double) n)
```

para converter o valor de n antes de passá-lo a *sqrt*. Observe que o molde produz o valor de *n* no tipo apropriado; o conteúdo de *n* não é alterado. O operador de molde tem a mesma precedência que qualquer outro operador unário, como está resumido na tabela ao final deste capítulo. Se forem declarados argumentos por um protótipo de função, como normalmente devem ser, a declaração causa uma moldagem automática de quaisquer argumentos quando a função é chamada. Assim, com o protótipo de função para *sqrt*:

```
double sqrt(double);
```

a chamada

```
raiz2 = sqrt(2);
```

força o inteiro 2 para o valor *double* 2.0 sem qualquer necessidade de um molde.

A **biblioteca-padrão** inclui uma implementação portátil de um gerador de número pseudo-randômico e uma função para inicializar a semente; a primeira ilustra o uso de um molde:

Programa 2.4: Gerador de números pseudo-aleatórios

```
unsigned long int prox = 1;

/*rand:retorna inteiro pseudo-randômico entre 0..32767 */
int rand (void)
{
    prox = prox * 1103515245 + 12345;
    return (unsigned int) (prox / 65536) % 32768;
}

/* srand: define semente para rand () */
```

```
void srand (unsigned int semente)
{
    prox = semente;
}
```

**Exercício 2.3** Escreva a função $htoi(s)$, que converte uma *string* de dígitos hexadecimais (incluindo um $0x$ ou $0X$ opcional) no seu valor inteiro equivalente. Os dígitos permitidos são de 0 a 9, de $a$ a $f$, e de $A$ a $F$. ■

## 2.8 Operadores de Incremento e Decremento

**C** fornece dois operadores não usuais para incrementar e decrementar variáveis. O operador de incremento $++$ soma 1 ao seu operando, enquanto o operador de decremento $--$ subtrai 1. Usamos frequentemente o operador $++$ para incrementar variáveis, como no caso de

```
if (c == '\n'){
  ++nl;
}
```

O aspecto incomum é que $++$ e $--$ podem ser usados tanto como operadores prefixados (antes da variável, como em $++n$) ou pós-fixados (após a variável, como em $n++$). Em ambos os casos, a variável $n$ é incrementada. Mas a expressão $++n$ incrementa $n$ antes que seu valor seja usado, enquanto $n++$ incrementa n depois que seu valor foi usado. Isso significa que em um contexto onde o valor está sendo usado, e não apenas o efeito, $++n$ e $n++$ são diferentes. Se $n$ é 5, então

```
x =  n++;
```

atribui 5 a $x$, mas

```
x = ++n;
```

atribui 6 a $x$. Em ambos os casos, $n$ torna-se 6. Os operadores de incremento e decremento só podem ser aplicados a variáveis; uma expressão como $(i+j)++$ é ilegal.

Num contexto no qual o valor não é usado e queremos apenas o efeito de incrementar, como em

```
if (c=='\n')
  ++nl;
```

a escolha de prefixação ou posfixação não faz diferença. Mas há situações onde um ou outro uso é mais apropriado. Por exemplo, considere a função $comprime(s,c)$, que remove todas as ocorrências do caractere $c$ da string $s$.

Programa 2.5: Função comprime()

```
/* comprime: deleta todos os c de s */
void comprime (char s[], int c)
{
    int i,j;

    for(i = j = 0; s[i] != '\0'; i++){
        if (s[i] != c){
            s[j++] = s[i];
        }
    }
    s[j] = '\0';
}
```

Cada vez que um caractere diferente de $c$ ocorre em $s$, ele é copiado para a *j-ésima* posição, e somente então $j$ é incrementado para ficar pronto para o próximo caractere. Isso é exatamente equivalente a

```
  if (s[i] != c) {
    s[j] = s[i];
    j++;
  }
```

Um outro exemplo de uma construção semelhante vem da função *lelinha* que escrevemos no Capítulo 1, onde podemos substituir

```
if (c == '\n') {
  s[i] = c;
  ++i;
}
```

pela forma mais compacta

```
if (c == '\n'){
  s[i++] = c;
}
```

Como terceiro exemplo, considere a função $strcat(s,t)$, que concatena a string $t$ ao fim da string $s$. *strcat* assume que há espaço suficiente em $s$ para conter a combinação. Conforme está escrito, *strcat* não retorna um valor; a versão da biblioteca-padrão retorna um apontador para a string resultante.

Programa 2.6: Função strcat()

```
/* strcat: concatena t ao final de s;
 * s deve ter tamanho suficiente */
void strcat(char s[], char t[])
{
    int i, j;
    i = j = 0;

    while (s [i] != '\0') { // acha fim de s
        i++;
    }

    while((s[i++] = t[j++]) != '\0') // copia t
        ;
}
```

A cada caractere copiado de $t$ para $s$, o $++$ posfixado é aplicado a $i$ e $j$ para garantir que eles estejam em posição para a próxima passada do laço.

**Exercício 2.4** Escreva uma versão alternativa de *comprime (s1, s2)* que remova cada caractere de $s1$ que se case a algum caractere presente na string $s2$. ■

**Exercício 2.5** Escreva uma função qualquer *(s1, s2)* que retorne a primeira posição na string $s1$ onde qualquer caractere presente na string $s2$ ocorra, ou $-1$ se $s1$ não contiver caracteres de $s2$. (A função da biblioteca-padrão *strpbrk* faz a mesma coisa, mas retorna um apontador para o local.). ■

## 2.9 Operadores Lógicos Bit-a-Bit

**C** provê seis operadores para manipulação de bit; eles somente podem ser aplicados a operandos inteiros, isto é, *char*, *short*, *int* e *long*, sinalizados ou não.

| | |
|---|---|
| & | E bit-a-bit |
| \| | OU inclusivo bit-a-bit |
| ∧ | OU exclusivo bit-a-bit |
| << | deslocamento a esquerda |
| >> | deslocamento a direita |
| ∼ | complemento de um (unário) |

O operador E bit-a-bit & é normalmente usado para mascarar algum conjunto de bits; por exemplo,

```
n = n & 0177;
```

zera todos os bits de *n*, exceto os 7 menos significativos.

O operador OU bit-a-bit | é usado para ativar bits:

```
x = x | ATIVO;
```

torna um em *x* os bits que são um em ATIVO.

O operador OU exclusivo bit-a-bit ∧ torna um em cada posição de bit onde seus operandos possuem bits diferentes, e zero onde forem iguais.

Deve-se distinguir os operadores bit-a-bit & e | dos operadores lógicos && e ||, que implicam uma avaliação da esquerda para a direita de um valor booleano. Por exemplo, se *x* é 1 e y é 2, então *x&y* é zero, enquanto *x&&y* é um.

Os operadores de deslocamento << e >> executam deslocamentos à esquerda e direita do seu operando da esquerda pelo número de posições de bit dadas pelo operando da direita, que deve ser não negativo. Assim, *x* << 2 desloca o valor de *x* à esquerda de duas posições, preenchendo os bits vagos com zero; isso é equivalente a multiplicar por 4. Um deslocamento à direita de uma quantidade sem sinal preenche os bits vagos com 0. Um deslocamento à direita de uma quantidade com sinal preenche os bits vagos como bit de sinal ("deslocamento aritmético") em algumas máquinas, e com bits 0 ("deslocamento lógico") em outras.

O operador unário ∼ gera o complemento de um inteiro, isto é, ele converte cada bit 1 em 0 e vice-versa. Por exemplo,

```
x = x & ~077
```

zera os últimos seis bits de *x*. Observe que *x* & ∼ 077 é independente de tamanho de palavra, e é preferível a, por exemplo, *x* & 0177700, que supõe que *s* tenha 16 bits. A forma transportável não envolve custo extra, desde que ∼ 077 é uma expressão constante que pode ser avaliada em tempo de compilação.

Para ilustrar o uso de alguns operadores bit-a-bit, considere a função *obtembits(x, p, n)* que retorna (ajustado à direita) o campo de n-bits de *x* que começa na *p-ésima* posição. Nós assumimos que a posição 0 é a mais à direita e que *n* e *p* são valores positivos com sentido. Por exemplo, *obtembits (x, 4. 3)* retorna os três bits nas posições de bit, 4, 3 e 2, ajustados à direita.

Programa 2.7: Função obtembits()

```
/* obtembits: obtém n bits da posição p */
unsigned obtembits (unsigned x, int p, int n)
{
    return (x >> (p + 1 - n)) & ~(~0 << n);
}
```

A expressão $x >> (p+1-n)$ move o campo desejado para a direita da palavra. $\sim 0$ significa todos os bits l; deslocá-lo para a esquerda $n$ posições de bit com $\sim 0 << n$ coloca zeros nos $n$ bits mais à direita; a complementação com $\sim$ faz com que a máscara tenha uns nos $n$ bits mais à direita.

**Exercício 2.6** Escreva uma função $setabits(x,p,n,y)$ que retorne $x$ com os $n$ bits que começam na posição $p$ setados com os $n$ bits mais à direita de $y$, deixando os outros bits inalterados. ■

**Exercício 2.7** Escreva uma função $inverte(x,p,n)$ que retorne $x$ com os $n$ bits que começam na posição $p$ invertidos (i.e., 1 passado para 0 e vice-versa), deixando os demais inalterados. ■

**Exercício 2.8** Escreva uma função $giradir(x,n)$ que retorne o valor do inteiro $x$ girado à direita de $n$ posições de bit. ■

## 2.10 Operadores e Expressões de Atribuição

Expressões do tipo

```
i=i+2
```

em que a variável do lado esquerdo é repetida imediatamente à direita, podem ser escritas de forma mais compacta

```
i+=2
```

O operador $+=$ é chamado *operador de atribuição*.

A maioria dos operadores binários (operadores como $+$, que possuem um operando à esquerda e outro à direita) tem um operador de atribuição correspondente $op=$, onde $op$ pode ser um dos seguintes:

$+$ $-$ $*$ $/$ $\%$ $<<$ $>>$ $\&$ $\wedge$ $|$ Se $exp1$ e $exp2$ são expressões, então

```
exp1 op= exp2
```

é equivalente a

```
exp1 = (exp1) op (exp2)
```

exceto que $exp1$ só é avaliada uma vez. Observe os parênteses em $exp2$:

```
x *= y + 1
```

significa

```
x = x * (y + 1)
```

e não

```
x = x * y + 1
```

Como exemplo, a função *contabits* conta o número de bits 1 no seu argumento inteiro.

Programa 2.8: Função contabits()

```
/* contabits: conta bits 1 em x */
int contabits (unsigned x)
{
    int b;
```

```
    for (b = 0; x != 0; x >>= 1) {
        if (x & 01) {
            b++;
        }
    }

    return b;
}
```

Declarar o argumento *x* como *unsigned* assegura que, ao ser deslocado à direita, os bits vagos serão preenchidos com zeros, e não com bits de sinal, não importa a máquina em que o programa está sendo rodado.

Concisão à parte, os operadores de atribuição têm a vantagem de corresponder melhor ao modo de pensar das pessoas. Nós dizemos "*soma 2 a i*" ou "*incremente i de 2*", e não "*toma i, soma 2 e coloca o resultado em i*". Assim, a expressão `i += 2` é preferível a `i = i + 2`. Além disso, para uma expressão complicada do tipo

```
yyval[yypv[p3+p4] + yypv[p1+p2]] += 2
```

o operador de atribuição toma o código mais fácil de entender, já que o leitor não tem de analisar a escrita correta de duas expressões longas para verificar se são iguais ou porque não o são. E um operador de atribuição pode até ajudar o compilador a produzir um código mais eficiente.

Já vimos que o comando de atribuição tem um valor, e pode ocorrer em expressões, o exemplo mais comum é

```
while ((c = getchar ()) != EOF)
```

Os outros operadores de atribuição (`+=`, `-=`, etc.) também podem ocorrer em expressões, embora seja menos comum.

Em todas essas expressões, o tipo de uma expressão de atribuição é o tipo do seu operando à esquerda, e o valor é o valor após a atribuição.

**Exercício 2.9** Em um sistema de numeração usando complemento de 2, `x &= (x - 1)` apaga o bit 1 que está mais à direita em *x*. Por que? Use essa observação para escrever uma versão mais rápida de *contabits*. ■

## 2.11 Expressões Condicionais

Os comandos

```
if (a > b)
  z = a;
else
  z = b;
```

calculam em *z* o valor máximo entre *a* e *b*. A *expressão condicional*, escrita com o operador ternário "? :", fornece uma forma alternativa de escrever isso e outras construções semelhantes. Na expressão

```
expr1 ? expr2 : expr3
```

a expressão $expr_1$ é avaliada primeiro. Se for diferente de zero (verdadeira), então a expressão $expr_2$ é avaliada, e esse é o valor da expressão condicional. Caso contrário, $expr_3$ é avaliada, e esse é o valor. Somente uma dentre as expressões $expr_2$ e $expr_3$ é avaliada. Assim, para que *z* receba o maior valor entre *a* e *b*,

```
  z = (a > b) ? a : b; // z = max(a, b)
```

Deve-se notar que uma expressão condicional é também uma expressão, e pode ser usada como qualquer outra expressão. Se expr2 e expr3 são de tipos diferentes, o tipo do resultado é determinado pelas regras de conversão discutidas inicialmente neste capítulo. Por exemplo, se $f$ for um $float$, e $n$ for um $int$, então a expressão

```
(n > 0) ? f : n
```

é do tipo $float$, não importa se $n$ é positivo.

Parênteses não são necessários ao redor da primeira expressão de uma expressão condicional, pois a precedência de ? : é muito baixa, imediatamente acima da atribuição. Os parênteses são aconselháveis de qualquer forma, entretanto, uma vez que eles tornam a parte condicional da expressão mais visível.

A expressão condicional leva frequentemente a um código compacto. Por exemplo, o laço a seguir imprime $n$ elementos de um vetor, 10 por linha, com cada coluna separada por um espaço, e com cada linha (inclusive a última) terminada por exatamente um caractere de nova-linha.

```
for (i = 0; i < n; i++){
  printf("%6d%c", a[i], (i % 10 == 9 || i == n - 1) ? '\n':' ');
}
```

Um caractere de nova-linha é impresso após cada conjunto de dez elementos e após o *n-ésimo* elemento. Todos os outros elementos são seguidos por um espaço. Embora isso possa parecer complicado, é instrutivo tentar escrevê-lo sem expressão condicional.

**Exercício 2.10** Reescreva a função minúsculo a qual converte letras maiúsculas em minúsculas, com uma expressão condicional no lugar de $if-else$. ■

## 2.12 Precedência e Ordem de Avaliação

A Tabela 2.1 resume as regras para precedência e associatividade de todos os operadores, incluindo aqueles de que ainda não falamos. Os operadores na mesma linha possuem a mesma precedência; as linhas estão em ordem decrescente de precedência, de modo que, por exemplo, $*$, $/$ e $\%$ têm a mesma precedência, que é maior que a do binário $+$ e $-$. O "operador" $()$ refere-se a uma chamada de função. Os operadores `->` e `.` são usados para referenciar membros de estruturas; eles serão vistos no Capítulo 6, juntamente com $sizeof$ (tamanho de um objeto). O Capítulo 5 discute sobre $*$ (indireção por meio de um apontador) e & (endereço de um objeto), e o Capítulo 3 discute o operador vírgula.

Observe que a precedência dos operadores bit-a-bit &, ^ e | fica abaixo de `==` e `!=`. Isto implica que expressões de teste de bit tais como

```
if ((x & MASCARA) == 0)...
```

devem ter parênteses incluídos para que gere resultados corretos. **C** , como a maioria das outras linguagens, não especifica a ordem em que os operandos de um operador são avaliados. (As exceções são &&, ||, ? : e ','.) Por exemplo, em um comando como

```
x = f() + g();
```

$f$ pode ser avaliado antes de $g$ ou vice-versa; assim, se $f$ ou $g$ alterar uma variável sobre a qual o outro depende, o valor de $x$ pode depender da ordem de avaliação. Os resultados intermediários podem ser armazenados em variáveis temporárias para assegurarem uma sequência particular.

De forma semelhante, a ordem em que os argumentos de função são avaliados não é especificada, de modo que o comando

```
printf("%d %d\n", ++n, pot(2, n));  // ERRADO
```

pode produzir resultados diferentes com compiladores diferentes, dependendo de *n* ser ou não incrementado antes da chamada a *pot*. A solução, é claro, seria escrever

```
++n;
printf("%d %d\n", n, pot(2, n));
```

| Operadores | Associatividade |
|---|---|
| () [] −> . * | esquerda para direita |
| ! ~ ++ −− + − * (type) sizeof | direita para esquerda |
| * / % | esquerda para direita |
| + − | esquerda para direita |
| << >> | esquerda para direita |
| < <= > >= | esquerda para direita |
| == != | esquerda para direita |
| & | esquerda para direita |
| ∧ | esquerda para direita |
| \| | esquerda para direita |
| && | esquerda para direita |
| \|\| | esquerda para direita |
| ?: | direita para esquerda |
| = += −= *= /= %= &= ∧= \|= <<= >>= | direita para esquerda |
| , | esquerda para direita |

Unários &, +, − e * tem maior precedência que as formas binárias.

Tabela 2.1: Precedência e Associatividade de Operadores

As chamadas de função, comandos de atribuição indentados e operadores de incremento e decremento causam "efeitos colaterais" – alguma variável é mudada como subproduto da avaliação de uma expressão. Em qualquer expressão envolvendo efeitos colaterais, pode haver dependências sutis na ordem em que as que tomam parte na expressão são atualizadas. Uma situação infeliz é exemplificada pelo comando

```
a[i] = i++;
```

A questão é: o subscrito é o valor antigo de *i* ou o novo? Os compiladores podem interpretar isto de formas diferentes, gerando diferentes respostas dependendo de sua interpretação. O padrão deixa intencionalmente a maioria dessas questões indeterminadas. Quando os efeitos colaterais (atribuição a variáveis) ocorrerão dentro de uma expressão é deixado por conta do compilador, pois a melhor ordem depende muito da arquitetura da máquina. (O padrão especifica que todos os efeitos colaterais nos argumentos ocorrem antes que uma função seja chamada, mas isso não ajudaria na chamada a *printf* acima.)

A moral é que a escrita de código com dependência na ordem de avaliação de expressões não é boa prática de programação em qualquer linguagem. Naturalmente, é necessário conhecer o que evitar, mas se você não conhece como certas coisas são feitas em várias máquinas, esta inocência pode ajudar a protegê-lo da tentação de tirar proveito de uma implementação particular.

# 3. Fluxo de Controle

Os comandos de fluxo de controle de uma linguagem especificam a ordem em que a computação é feita. Nós já vimos as construções mais comuns de fluxo de controle nos exemplos anteriores; aqui completaremos o conjunto e seremos mais precisos na discussão das construções já vistas.

## 3.1 Comandos e Blocos

Uma expressão tal como $x = 0$ ou $i++$ ou $printf(...)$ torna-se um comando quando seguida por um ponto-e-vírgula, como em:

```
x = 0;
i++;
printf (...);
```

Em **C** , o ponto-e-virgula é um terminador de comandos e não um separador como em linguagens do tipo Pascal. As chaves { e } são usadas para agruparem declarações e comandos num comando composto ou bloco de modo que são sintaticamente equivalentes a um único comando. As chaves que cercam os comandos de uma função são um exemplo óbvio; as chaves que cercam múltiplos comandos após um *if*, *else*, *while* ou *for* são outro. (Variáveis podem ser declaradas dentro de qualquer bloco; falaremos sobre isto no Capítulo 4.) Não há um ponto-e-vírgula após a chave direita que termina um bloco.

## 3.2 If-else

O comando $if-else$ é usado para expressar decisões. Formalmente, a sintaxe é

```
if (expressão)
  comando1
else
  comando2
```

onde a parte do *else* é opcional. A expressão é avaliada; se for verdadeira (isto é, se expressão tiver um valor diferente de zero), `comando`$_1$, é executado. Se for falsa (expressão é zero) e se houver

uma parte *else*, $\texttt{comando}_2$ é executado. Como um *if* simplesmente testa o valor numérico de uma expressão, certas abreviações na codificação são possíveis. A mais óbvia é escrever

```
if (expressão)
```

no lugar de

```
if (expressão != 0)
```

Algumas vezes isso é natural e claro; outras vezes, pode ser totalmente obscuro. Devido à parte *else* do *if* – *else* ser opcional, há uma ambiguidade quando um *else* é omitido em uma sequência de comandos *if* aninhados. Isso é resolvido do modo usual – o *else* é sempre associado com o mais recente *if* sem *else*. Por exemplo, em

```
if (n > 0)
  if(a>b)
    z = a;
  else
    z = b;
```

o *else* corresponde ao *if* interno, como mostrado pela indentação. Caso isto não seja o que você quer, chaves devem ser usadas para forçar a associação apropriada:

```
if (n>0) {
   if (a>b)
     z = a;
}
else
  z = b;
```

A ambiguidade é especialmente perniciosa em situações como:

```
if (n >= 0)
  for (i = 0; i < n; ++i)
    if (s [i] > 0) {
      printf ("...");
      return i;
    }
else /*ERRADO */
  printf ("erro - n é negativo\n");
```

A indentação mostra inequivocamente o que você quer, mas o compilador não entende e associa o *else* ao *if* interno. Esse tipo de problema pode ser muito difícil de achar.

Observe que há um ponto-e-vírgula após $z = a$ em

```
if(a > b)
  z = a;
else
  z = b;
```

Isso ocorre porque, gramaticamente, um comando segue o *if*, e um comando de expressão como "`z = a;`" é sempre terminado por um ponto-e-vírgula.

## 3.3 else-if

A construção

```
if (expressão)
  comando
else if (expressão)
  comando
else if (expressão)
  comando
else if (expressão)
  comando
else
  comando
```

ocorre tantas vezes que merece uma discussão em separado. Esta sequência de comandos *if* é a forma mais geral de escrever uma decisão múltipla. As expressões são avaliadas em ordem; se qualquer expressão for verdadeira, o comando associado a ela é executado, e toda a string é terminada. Como sempre, o código para cada comando pode ser um único comando ou um grupo entre chaves.

A última parte *else* trata do caso "nenhuma das anteriores" ou caso *default*, quando nenhuma das condições é satisfeita. Algumas vezes não há ação explicita para o *default*; neste caso o

```
else
  comando
```

final pode ser omitido, ou pode ser usado para a verificação de erro, capturando assim uma condição "impossível".

Para ilustrar uma decisão tripla, segue uma função de pesquisa binária que decide se um valor particular de $x$ ocorre em um vetor ordenado $v$. Os elementos de $v$ devem estar em ordem crescente. A função retorna a posição (um número entre 0 e $n-1$) se $x$ pertence a $v$, e $-1$ caso contrário.

A pesquisa binária primeiro compara o valor de entrada $x$ com o elemento do meio do vetor $v$. Se $x$ for menor que o valor do meio, a pesquisa abrange a metade inferior da tabela, caso contrário, a metade superior. De qualquer modo, o próximo passo é comparar $x$ ao elemento do meio da metade selecionada. Este processo de dividir a faixa ao meio continua até que o valor seja encontrado ou a faixa esteja vazia.

Programa 3.1: Função para pesquisa binária.

```
/* pesqbin: acha x em v[0] <= v[1]... <= v[n - 1] */
int pesqbin (int x, int v[], int n)
{
    int inicio, fim, meio;
    inicio = 0;
    fim = n - 1;

    while (inicio <= fim) {
        meio = (inicio + fim) / 2;

        if (x <  v [meio]) {
            fim = meio - 1;
        } else if (x > v [meio]) {
            inicio = meio + 1;
        } else { // encontrou
            return meio;
        }
    }

    return -1; //não encontrou
```

```
}
```

A decisão fundamental é se *x* é menor, maior ou igual ao elemento do meio *v[meio]* em cada passo; isso é uma construção apropriada para um *else-if*.

**Exercício 3.1** Nossa pesquisa binária faz dois testes dentro do laço, quando somente um seria suficiente (desde que houvessem mais testes fora dele). Escreva uma versão com somente um teste dentro do laço e meça a diferença em tempo de execução. ■

## 3.4 Switch

O comando *switch* é uma estrutura de decisão múltipla que testa se uma expressão casa um de vários valores inteiros constantes, e desvia de acordo com o resultado.

```
switch (expressão) {
  case expr-constante: comandos
  case expr-constante: comandos
  default: comandos
}
```

Cada caso é rotulado por uma ou mais constantes de valor inteiro ou expressões constantes. Se um caso combina com o valor da expressão, a execução inicia nesse caso. Todas as expressões de caso devem ser diferentes umas das outras. O caso intitulado *default* é executado se nenhum outro for satisfeito. Um *default* é opcional se não estiver presente e nenhum dos casos combinar com a expressão, nenhuma ação é tomada. Os casos e a cláusula *default* podem ocorrer em qualquer ordem.

No Capítulo 1 nós escrevemos um programa para contar o número de cada dígito, espaço em branco, e outros caracteres usando uma sequência *if ... else if... else*. Aqui está o mesmo programa com um *switch*:

Programa 3.2: Programa que conta dígitos, espaços e caracteres usando switch.

```
#include <stdio.h>
/* conta dígitos, espaço branco, outros */
int main ()
{
    int c, i, nbranco, noutro, ndigito [10];
    nbranco = noutro = 0;

    for (i = 0; i < 10; i++) {
        ndigito [i] = 0;
    }

    while (( c = getchar ()) != EOF) {
        switch(c) {
        case '0': case '1': case '2': case '3': case '4':
        case '5': case '6': case '7': case '8': case '9':
            ndigito [c - '0']++;
            break;

        case ' ':
        case '\n':
        case '\t':
            nbranco++;
            break;
```

```
        default:
            noutro++;
            break;
        }
    }

    printf ("dígitos =");

    for (i = 0; i < 10; i++) {
        printf (" %d", ndigito[i]);
    }

    printf (", espaço branco= %d, outros= %d\n", nbranco, noutro);
    return 0;
}
```

O comando *break* causa uma saída imediata do *switch*. Como os casos servem apenas como rótulos, após ser terminado o código para um caso, a execução prossegue para o próximo caso, a não ser que você tome uma ação alternativa de escape, *break* e *return* são as formas mais comuns de deixar um *switch*. Um comando *break* também pode ser usado para forçar uma saída imediata de laços *while*, *for* e *do*, como veremos mais adiante neste capítulo.

Prosseguir de um caso para outro é uma faca de dois gumes. Do lado positivo, ela permite múltiplos casos para uma única ação, como nos casos *branco*, *caractere de tabulação* ou *nova-linha* neste exemplo. Mas implica também que, normalmente, cada caso, deva ser encerrado com um *break* para evitar o prosseguimento de fluxo para o próximo caso. Prosseguir de um caso para outro não é robusto, propiciando a desintegração do programa quando o mesmo é modificado. Com a exceção de múltiplos rótulos para uma única computação, o prosseguimento de um caso para outro deve ser usado raramente, e comentado.

Por questão de bom estilo de programação, coloque um *break* após o último caso (o *default* aqui) mesmo que logicamente não seja necessário. Algum dia quando outros casos forem acrescentados, essa pequena precaução vai auxiliá-lo.

**Exercício 3.2** Escreva uma função *escape(s , t)* que converta caracteres como *nova-linha* e *tabulação* em sequências de escape visíveis como $\backslash n$ e $\backslash t$, respectivamente, enquanto copia a string *t* em *s*. Use um *switch*. Escreva uma função também para a outra direção, convertendo sequências de escape em caracteres reais. ■

## 3.5 Laços - While e For

Nós já vimos os laços *while* e *for*. Em

```
while (expressão)
   comando
```

a *expressão* é avaliada. Se for diferente de zero, *comando* é executado e *expressão* é reavaliada. Este ciclo continua até *expressão* se tornar zero, continuando então a execução após comando.

O comando *for*

```
for (expr1; expr2; expr3)
   comando
```

é equivalente a

```
expr1;
while (expr2) {
  comando
  expr3;
}
```

exceto pelo comportamento de *continue*, que é descrito na Seção 3.7.

Gramaticamente, os três componentes de um laço *for* são expressões. Normalmente, $expr_1$ e $expr_3$ são atribuições ou chamadas de função e $expr_2$ é uma expressão relacional. Qualquer uma das três partes pode ser omitida, embora os *ponto-e-vírgula* devam permanecer. Se $expr_1$ ou $expr_3$ forem omitidas, ela é simplesmente desconsiderada. Se o teste, $expr_2$, não está presente, é considerada permanentemente verdadeira, de forma que

```
for(;;) {
...
}
```

é um laço "infinito", e presumivelmente será encerrado por outros meios, tais como um *break* ou *return*.

Quando usar *while* ou *for* é uma questão de gosto. Por exemplo, em

```
while ((c = getchar()) == ' '  || c == '\n' || c == '\t')
    ;   // pula espaços em branco
```

não há inicialização ou reinicialização, de modo que o *while* parece mais natural.

O *for* é claramente superior quando há uma inicialização e reinicialização simples, já que ele guarda os comandos de controle do laço juntos e visíveis no topo do laço. Isso é muito óbvio em

```
for (i = 0; i < n; i++)
```

que é o idioma **C** para processar os primeiros n elementos de um vetor, análogo ao laço *DO* do *Fortran* ou ao *for* do *Pascal*. A analogia não é perfeita, entretanto, desde que o índice e limite de um laço for em **C** podem ser alterados de dentro do laço, e a variável de índice *i* retém seu valor quando o laço termina por algum motivo. Como os componentes do *for* são expressões arbitrárias, os laços *for* não são restritos a progressões aritméticas, não é um bom estilo forçar cálculos não relacionados na inicialização e incremento de um *for*, que são mais bem empregados para operações de controle de laços.

Como um exemplo maior, aqui está uma outra versão de *atoi* para converter uma *string* ao seu equivalente numérico. Este é um pouco mais geral do que o do Capítulo 2; ele copia com espaços iniciais opcionais e um sinal $+$ ou $-$ opcional. (O Capítulo 4 mostra a função *atof*, que faz a conversão para números de ponto flutuante.)

A estrutura do programa reflete a forma da entrada:

```
salte espaços em branco, se houver
obtenha o sinal, se houver
obtenha a parte inteira, converta-a
```

Cada passo faz sua parte e deixa as coisas arrumadas para o próximo passo. O processo inteiro termina no primeiro caractere que não puder fazer parte de um número.

Programa 3.3: Função atoi() aprimorada.

```
/* atoi: converte s para inteiro; versão 2 */
int atoi (char s[])
{
```

```
    int i, n, sinal;

    for (i = 0; isspace(s[i]); i++) // salta espaço
        ;

    sinal = (s [i] == '-') ? -1 : 1;

    if (s [i] == '+'  || s[i] == '-') { // salta sinal
        i++;
    }

    for (n = 0; isdigit(s[i]); i++) {
        n = 10 * n + (s[i] - '0');
    }

    return sinal * n;
}
```

A biblioteca-padrão fornece uma função mais elaborada *strtol* para a conversão de cadeias para inteiros longos; veja a Seção B.5.

As vantagens de manter o controle do laço centralizado são mais óbvias quando há vários laços aninhados. A função seguinte é uma ordenação **Shell** para um vetor de inteiros. A ideia básica deste algoritmo de ordenação, que foi inventada em 1959 por **D.L. Shell**, é em estágios iniciais, elementos distantes são comparados, ao invés de elementos adjacentes, como em ordenações simples de trocas. Isto tende a eliminar a grande quantidade de desordem rapidamente, de modo que estágios posteriores tenham menos trabalho a fazer. O intervalo entre elementos comparados é gradualmente decrementado até um, em cujo ponto a ordenação se torne efetivamente um método de troca adjacente.

Programa 3.4: Função de ordenação shell().

```
/* shell:ordena v [0]..v [n - 1] em ordem crescente */
void shell (int v [], int n)
{
    int inter, i, j, temp;

    for (inter = n / 2; inter > 0; inter /= 2)
        for (i = inter; i < n; i++)
            for(j = i - inter; j >= 0 && v[j] > v[j + inter]; j -= inter) {
                temp = v[j];
                v[j] = v[j + inter];
                v[j + inter] = temp;
            }
}
```

Há três laços aninhados. O mais externo controla o intervalo entre os elementos comparados partindo de $n/2$ e dividindo por 2 a cada passo até se tornar zero. O laço do meio percorre os elementos. O laço mais interno compara cada par de elementos separados por *inter* e inverte qualquer um que esteja fora de ordem. Como *inter* eventualmente é reduzido para um, todos os elementos ficam eventualmente ordenados corretamente. Observe como a generalidade do *for* torna o laço externo igual em formato aos outros laços, muito embora ele não seja uma progressão aritmética.

Nosso último operador em **C** é a vírgula "," , mais frequentemente usado no comando *for*. Um par de expressões separadas por uma vírgula é avaliado da esquerda para a direita, e o tipo e valor do resultado são o tipo e o valor do operando direito. Então, num comando *for*, é possível colocar

múltiplas expressões em várias partes, por exemplo para processar dois índices em paralelo. Isto é ilustrado na função *inverte(s)*, que inverte a string *s*.

Programa 3.5: Função inverte().

```
#include <string.h>
/* inverte: inverte string s no lugar */
void inverte (char s[])
{
    int c;
    for (int i = 0, j = strlen(s)-1; i < j; i++, j--) {
        c = s[i];
        s[i] = s[j];
        s[j] = c;
    }
}
```

As vírgulas que separam argumentos de funções, variáveis em declarações etc. não são operadores e não garantem a avaliação da esquerda para a direita.

Os operadores de vírgula devem ser usados com cautela. Os usos adequados são em construções fortemente relacionadas com outras, como no laço *for* em *inverte*, e em macros onde um cálculo múltiplo deva ser uma única expressão. Uma expressão com vírgula também poderia ser adequada para a troca de elementos em inverte, onde a troca seria feita numa única operação:

```
for(i = 0, j = strlen(s) - 1; i < j; i++, j--){
   c = s[i], s[i] = s[j], s[j] = c;
}
```

**Exercício 3.3** Escreva uma função *expande(s1, s2)* que expanda notações abreviadas como $a-z$ na string *s1* para a lista completa equivalente *abc...xyz* em *s2*. Permita letras maiúsculas e minúsculas, além de dígitos, e esteja preparado para lidar com casos tais como $a-b-c$ e $a-10-9$ e $-a-z$. Faça com que o $-$ inicial ou final seja tomado literalmente. ■

## 3.6 Laços - Do-while

Como dissemos no Capítulo 1, os laços *while* e *for* testam a condição de término no início. Ao contrário, o terceiro loop em **C** , o *do-while*, testa ao final, depois de fazer cada passagem pelo corpo do laço; o corpo é sempre executado pelo menos uma vez.

A sintaxe do laço do é

```
do
  comando
while (expressão)
```

O *comando* é executado, então *expressão* é avaliada. Se for verdadeira, *comando* é executado novamente, e assim por diante. Se a *expressão* se tornar falsa, o laço termina. Exceto pelo sentido do teste, *do-while* é equivalente ao comando *repeat-until* do **Pascal**.

A experiência mostra que *do-while* é muito menos usado do que *while* e *for*. Apesar disso, em certas ocasiões ele é valioso, como na seguinte função *itoa*, que converte um número para uma string de caracteres (o inverso de *atoi*). A tarefa é ligeiramente mais complicada do que se poderia pensar a princípio, pois os métodos fáceis de gerar dígitos os geram na ordem errada. Escolhemos gerar a string ao contrário, e depois invertê-la.

Programa 3.6: Função itoa().

```
/* itoa: converte n para caracteres em s */
void itoa (int n, char s[])
{
    int i, sinal;

    if ((sinal =  n) < 0) { // registra sinal
        n = -n;             // torna n positivo
    }

    i = 0;

    do {  // gera dígitos em ordem invertida
        s[i++] = n % 10 + '0'; // obtém prox.  dígito
    } while ((n /= 10) > 0);   // deleta-o

    if (sinal < 0) {
        s[i++] = '-';
    }

    s[i] = '\0';
    inverte (s);
}
```

O *do-while* é necessário, ou pelo menos conveniente, pois no mínimo um caractere deve ser instalado no vetor *s*, mesmo que *n* seja zero. Também usamos chaves ao redor do único comando que compõe o corpo do *do-while*, mesmo não sendo necessárias, de modo que o leitor apressado não confunda a parte *while* com o início de um laço *while*.

**Exercício 3.4** Em uma representação numérica de complemento de dois, nossa versão de *itoa* não lida com o maior número negativo, isto é, o valor de *n* igual a $-2^{tamanhodapalavra-1}$. Explique o motivo. Modifique-o para gerar esse valor corretamente, não importa a máquina em que esteja rodando. ■

**Exercício 3.5** Escreva uma função *itob(n, s, b)* que converta o inteiro *n* em uma representação de caracteres com base *b* na string *s*. Em particular, *itob(n, s, 16)* formata *n* como um inteiro hexadecimal em *s*. ■

**Exercício 3.6** Escreva uma versão de *itoa* que aceita três argumentos ao invés de dois. O terceiro argumento é o tamanho mínimo do campo; o número convertido deve ser preenchido com espaços à esquerda, se for necessário, de forma a torná-lo do comprimento desejado. ■

## 3.7 Break e Continue

É conveniente, às vezes, controlarmos a saída de um laço de outro modo além do teste, no início ou no fim do mesmo. O comando *break* permite uma saída antecipada de um *for*, *while* e *do*, bem como de um *switch*. Um *break* faz com que o laço (ou *switch*) mais interno seja terminado imediatamente.

O programa seguinte, *apara*, remove espaços, tabulações e caracteres de nova-linha do final de uma string, usando um *break* para sair de um laço quando um desses caracteres de controle for encontrado à direita de uma string.

Programa 3.7: Função apara().

```
/* apara: remove brancos, tabulações, novas-linhas */
int apara (char s[])
{
    int n;

    for (n = strlen(s) - 1; n >= 0; n--) {
        if(s[n] != ' ' && s[n] != '\t' && s[n] != '\n') {
            break;
        }
    }

    s[n + 1] = '\0';
    return n;
}
```

*strlen* retorna o tamanho da string. O laço *for* começa no fim da string e procura para trás o primeiro caractere que não é um espaço ou tabulação ou nova-linha. O laço é interrompido quando um deles for encontrado, ou quando *n* se tornar negativo (isto é, quando a string inteira tiver sido verificada). Você deve verificar que este é o comportamento correto mesmo quando a string está vazia ou contém somente caracteres em branco.

O comando *continue* é relacionado com o *break*, mas é menos usado; ele inicia a próxima iteração do laço, *while* ou *do*. No *while* e *do*, isso significa que a parte do teste será executada imediatamente; no *for*, o controle passa para o passo de incremento. O comando *continue* aplica-se somente a laços, e não a *switch*. Um continue num *switch* dentro de um laço provoca a próxima iteração do laço.

Como um exemplo, o fragmento a seguir processa somente números positivos no vetor *a*; valores negativos são saltados.

```
for (i = 0; i < n; i++){
  if (a[i] < 0)/*salta elementos negativos */
      continue;
  ...   /* processa elementos positivos */
  }
```

O comando *continue* é normalmente usado quando a parte do laço que se segue é complicada. de modo que a inversão do teste e a identação de um nível adicional aninharia o programa muito profundamente.

## 3.8 Goto e Rótulos

**C** fornece o comando infinitamente mal utilizável *goto* e rótulos para desvios. Formalmente, o *goto* nunca é necessário e na prática é sempre fácil escrever código sem usá-lo. Nós não usamos o *goto* neste livro.

No entanto sugerimos algumas poucas situações onde o *goto* possa ser útil. O uso mais comum é para abandonar o processamento numa estrutura encaixada profundamente, tal como o caso em que se deseja sair de dois laços ao mesmo tempo. O comando *break* não pode ser usado diretamente, já que abandona somente o laço mais interno. Então:

```
  for ( ... )
    for ( ... ) {
        ...
        if (desastre)
           goto erro;
    }
```

```
    ...
erro:
  /* arruma a casa */
```

Esta organização é prática se o código de manipulação de erro não for trivial, e se erros puderem ocorrer em vários lugares.

Um *rótulo* tem a mesma forma de um nome de variável, e é seguido por um sinal de dois pontos. Ele pode aparecer antes de qualquer comando na mesma função onde o *goto* apareceu. O escopo de um rótulo é a função inteira.

Como outro exemplo, considere o problema de determinar se dois vetores $a$ e $b$ possuem um elemento em comum. Uma possibilidade seria

```
  for (i = 0; i < n; i++)
     for (j = 0; j < m; j++)
        if (a[i] == b[j])
           goto achou;
  /* não achou elemento comum */
  ...
achou:
  /* achou na posição: a[i] == b[j] */
  ...
```

O código envolvendo um *goto* sempre pode ser escrito sem o mesmo, embora talvez com o custo de alguns testes repetidos ou de uma variável adicional. Por exemplo, a pesquisa do vetor ficaria

```
achou = 0;
for (i = 0; i < n \&\& !achou; i++)
   for (j = 0; j < m \&\& !achou; j++)
       if (a[i] == b[j])
          achou = 1;
if (achou)
  /* achou um: a[i-1] == b[j-1] */
  ...
else
  /* não achou elemento comum */
  ...
```

Com algumas exceções, como aquelas citadas, o código que se baseia no comando *goto* é geralmente mais difícil de ser entendido e mantido do que o código sem *goto*s. Embora não sejamos dogmáticos sobre o assunto, parece-nos que o *goto* deva ser usado raramente, ou nunca.

# 4. Funções e Estrutura de Programa

As funções dividem grandes tarefas de computação em tarefas menores, e permitem às pessoas trabalharem sobre o que outras já fizeram, ao invés de partirem do nada. Funções apropriadas escondem detalhes de operação de partes do programa que não necessitam conhecê-las, esclarecendo o todo, e facilitando as mudanças.

**C** foi projetada com funções eficientes e fáceis de usar; programas em **C** geralmente consistem em várias pequenas funções ao invés de poucas de maior tamanho. Um programa pode residir em um ou mais arquivos-fonte. Os arquivos-fonte podem ser compilados separadamente e carregados juntos, com funções de bibliotecas previamente compiladas. Não discutiremos este processo aqui, já que os detalhes variam de um sistema para outro.

A declaração e definição de função é a área onde o padrão **ANSI** realizou as mudanças mais visíveis ao **C**. Como vimos no Capítulo 1, agora é possível declarar os tipos dos argumentos quando uma função é declarada. A sintaxe da definição de função também mudou, de forma que as declarações e definições se casem. Isso possibilita a um compilador detectar muito mais erros do que podia antes. Além do mais, quando os argumentos são declarados apropriadamente, as moldagens adequadas de tipo são feitas automaticamente.

O padrão esclarece as regras de escopo de nomes; em particular, ele requer que haja somente uma definição de cada objeto externo. A inicialização é mais geral: os vetores e estruturas automáticas agora podem ser inicializados.

O *pré-processador* **C** também foi melhorado. Novas facilidades do pré-processador incluem um conjunto mais completo de diretivas condicionais de compilação, uma forma de criar cadeias entre aspas a partir de argumentos de macro e um melhor controle sobre o processo de expansão de macro.

## 4.1 Conceitos Básicos

Para iniciar, vamos projetar e escrever um programa para imprimir cada linha da sua entrada que contenha um "padrão" ou string de caracteres particular. (Isto é um caso especial do programa *grep* disponível no **UNIX**.) Por exemplo, a pesquisa do padrão "ar" no conjunto de linhas.

```
Ah, Amor! Pudéssemos nós com o Destino conspirar
para capturar o triste Modelo de Todas as Coisas,
para o partir em pedaços - e depois refazê-lo,
mais próximo dos desejos do coração!
```

produzirá a saída

```
Ah, Amor! Pudéssemos nós com o Destino conspirar
para capturar o triste Modelo de Todas as Coisas,
para o partir em pedaços - e depois refazê-lo,
```

A tarefa divide-se facilmente em três partes:

```
Enquanto (houver outra linha)
  se (a linha contém o padrão)
     imprima-a
```

Embora seja certamente possível colocar o código para tudo isso na rotina *main*, uma solução melhor é a de usar a estrutura natural fazendo de cada parte uma função separada. Três pequenas partes são mais fáceis de manipular que uma única grande, porque detalhes irrelevantes podem ser escondidos nas funções, e porque a chance de introduzir interações indesejadas é minimizada. E as partes podem até ser de utilidade por si só.

"Enquanto houver outra linha" é *lelinha*, uma função que escrevemos no Capítulo 1, e "imprima-a" é *printf*, que alguém já nos forneceu. Isso significa que temos de escrever apenas uma rotina que decida se a linha contém uma ocorrência do padrão.

Podemos resolver este problema escrevendo uma função *indicecad (s,t)* que retorne a posição ou índice na *string s* onde a *string t* começa, ou $-1$ se *s* não contém *t*. Como os vetores em **C** começam na posição zero, os índices serão zero ou positivos, e então um valor negativo como $-1$ é conveniente para indicar uma falha. Quando mais tarde precisarmos combinar padrões mais sofisticadamente, só precisaremos trocar *indicecad*; o restante do código permanece idêntico. (A biblioteca-padrão fornece uma função *strstr* que é semelhante a *indicecad*, exceto que retorna um ponteiro ao invés de um índice.)

Dado o projeto até este ponto, preencher os detalhes do programa é simples. O programa inteiro está a seguir, de modo que você possa ver como as peças se encaixam. Por ora, o padrão a ser pesquisado é uma string literal, que não é o mais geral dos mecanismos. Voltaremos em breve a uma discussão sobre como inicializar vetores de caracteres e, no Capítulo 5, mostraremos como passar o padrão como parâmetro enviado quando o programa é chamado para execução. Incluímos também uma versão ligeiramente diferente de *lelinha*; você pode achar instrutivo compará-la com a versão do Capítulo 1.

Programa 4.1: Programa que imprime cada linha que casa com um padrão.

```c
#include <stdio.h>
#define MAXLINHA 1000 // tamanho máximo da linha

int lelinha (char linha[], int max);
int indicecad (char fonte[], char procura[]);
char padrao[] = "ar"; /* padrão a procurar */
/* procura todas as linhas que combinam com o padrão */
int main()
{
    char linha[MAXLINHA];
    int achadas = 0;
```

```
    while (lelinha (linha, MAXLINHA) > 0) {
        if (indicecad (linha, padrao) >= 0) {
            printf ("%s", linha);
            achadas++;
        }
    }

    return achadas;
}
/* lelinha: obtém linha em s, retorna tamanho */
int lelinha (char s[], int lim)
{
    int c, i;
    i = 0;

    while (--lim > 0 && (c = getchar ()) != EOF && c != '\n') {
        s[i++] = c;
    }

    if (c == '\n') {
        s[i++] = c;
    }

    s[i] = '\0';
    return i;
}
/* indicecad: retorna índice de t em s, -1 se não tiver */
int indicecad (char s[], char t[])
{
    int i, j, k;

    for (i = 0; s[i] != '\0'; i++) {
        for (j = i, k = 0; t[k] != '\0' && s[j] == t[k]; j++, k++)
            ;

        if (k > 0 && t[k] == '\0') {
            return i;
        }
    }

    return -1;
}
```

Cada definição de função tem a forma

```
tipo_retorno nome_função (declarações de argumentos)
{
  declarações e comando;
}
```

Várias partes podem estar ausentes; uma função mínima é

```
vazia() {}
```

que não faz nada e não retorna nada. Uma função vazia como esta é às vezes útil como "guarda de local" durante o desenvolvimento do programa. Se o tipo de retorno for omitido, assume-se *int*.

C99 C99 não permite omitir o tipo de retorno.

Um programa é então um conjunto de definições de variáveis e funções. A comunicação entre funções é feita por argumentos e valores retornados pelas funções, além de variáveis externas. As funções podem ocorrer em qualquer ordem no arquivo-fonte, e o programa-fonte pode ser dividido em arquivos múltiplos, desde que nenhuma função seja dividida.

O comando *return* é o mecanismo para retornar um valor da função chamada para sua chamadora. Qualquer expressão pode seguir um *return*:

```
return expressão;
```

A *expressão* será convertida para o tipo de retorno da função se necessário. Parênteses são normalmente usados ao redor da *expressão*, mas são opcionais.

A função chamadora é livre para ignorar o valor retornado. Além do mais, não precisa haver uma expressão após o *return*; nesse caso, nenhum valor é retornado da função. O controle também retorna à função chamadora sem valor quando a execução "segue até o fim" de uma função passando pelo fecha-chave final da mesma. Isso não é ilegal, mas provavelmente um sinal de problema, se a função retornar um valor em alguns lugares e em outros não. Em qualquer caso, se uma função retorna sem um valor, seu "valor" certamente será lixo.

O programa de pesquisa de padrão retorna um estado em *main*, o número de combinações achadas. Este valor está disponível para o uso pelo ambiente que chamou o programa.

O mecanismo de compilação e carga de um programa em **C** que reside em múltiplos arquivos-fonte varia de um sistema para outro. No sistema **UNIX/LINUX**, por exemplo, o comando *gcc* mencionado no Capítulo 1 faz esse serviço. Suponha que as três funções estejam guardadas em três arquivos chamados *main.c*, *lelinha.c* e *indicecad.c*. Então o comando

```
gcc main.c lelinha.c indicecad.c
```

compila os três arquivos, colocando o código-objeto resultante nos arquivos *main.o*, *lelinha.o* e *indicecad.o*, e então os carrega para um único arquivo executável chamado *a.out*. Se houver um erro, digamos em *main.c*, esse arquivo pode ser recompilado sozinho e o resultado carregado com os outros arquivos-objeto (com extensão .o), com o comando

```
gcc main.c lelinha.o indicecad.o
```

O comando *gcc* usa a convenção de nomeação ".c" versus ".o" para distinguir arquivos-fonte de arquivos-objeto.

**Exercício 4.1** Escreva a função *indicecad (s,t)*, que retorna a posição da ocorrência mais à direita de $t$ em $s$, ou $-1$ se $t$ não ocorre em $s$. ■

## 4.2 Funções que retornam valores não Inteiros

Até aqui os exemplos de funções retornaram ou nenhum valor (*void*) ou um *int*. E se uma função tiver que retornar algum outro tipo? Muitas funções numéricas como *sqrt*, *sin* e *cos* retornam valores do tipo *double*; outras funções especializadas retornam outros tipos. Para ilustrar como podemos lidar com isto, vamos escrever e usar a função *atof(s)*, que converte a *string s* no seu equivalente em ponto flutuante de dupla precisão. *atof* é uma extensão de *atoi*, do qual mostramos versões nos Capítulo 1 e Capítulo 2. Ela manipula um sinal e ponto decimal opcionais, e a presença ou ausência de uma parte inteira ou fracionária. Nossa versão não é uma rotina de conversão de alta qualidade; tal rotina exigiria mais espaço do que queremos usar. A biblioteca-padrão inclui uma função *atof*; o arquivo de cabeçalho <*stdlib.h*> a declara.

Primeiro, *atof* deve declarar o tipo de valor que retorna, pois este não é *int*. O nome do tipo precede o nome da função:

Programa 4.2: Função atof().

```
#include <ctype.h>

/* atof: converte a string s para um double */
double atof (char s[])
{
    double val, pot;
    int i, sinal;

    for (i = 0; isspace (s[i]); i++) // ignora espacos iniciais
        ;
    sinal = (s[i] == '-') ? -1:1;
    if (s[i] == '+' || s[i] == '-'){
        i++;
    }
    for (val = 0.0; isdigit (s[i]); i++){
        val = 10.0 * val + (s[i] - '0');
    }
    if (s[i] == '.'){
        i++;
    }
    for (pot = 1.0; isdigit (s[i]); i++) {
        val = 10.0 * val + (s[i] - '0');
        pot *= 10.0;
    }
    return sinal * val / pot;
}
```

Segundo, e de igual importância, a rotina chamadora deve estabelecer que *atof* retorna um valor não inteiro. Uma forma de assegurar isto é declarar *atof* explicitamente na rotina chamadora. A declaração é mostrada nesta calculadora primitiva (adequada apenas para calcular o saldo no talão de cheques), que lê um número por linha, opcionalmente precedido por um sinal, e soma tudo, imprimindo o resultado acumulado após cada entrada.

Programa 4.3: Calculadora primitiva.

```
#include <stdio.h>
#define MAXLINHA 100

/* calculadora primitiva */
int main ()
{
    double soma, atof (char []);
    char linha[MAXLINHA];
    int lelinha (char linha[], int max);

    soma = 0;
    while (lelinha(linha, MAXLINHA) > 0){
        printf ("\t%g\n", soma += atof (linha));
    }
    return 0;
}
```

A declaração

```
    double soma, atof (char []);
```

diz que *soma* é uma variável do tipo *double*, e que *atof* é uma função que usa um argumento *char[]* e retorna um valor do tipo *double*.

A função *atof* deve ser declarada e definida de formas coerentes. Se a própria função *atof* e a chamada em *main* tiverem tipos incoerentes no mesmo arquivo-fonte, o erro será detectado pelo compilador. Mas se (como é mais provável) *atof* fosse compilado separadamente, a discordância não seria detectada, *atof* retornaria um *double* que *main* trataria como um *int*, e isso geraria respostas sem significado.

À luz do que dissemos sobre como as declarações devem combinar com as definições, isso pode parecer surpresa. O motivo de uma discordância poder acontecer é que se não houver um protótipo de função, uma função é implicitamente declarada por seu primeiro aparecimento em uma expressão, como em

```
sum += atof(linha)
```

Se um nome que não foi declarado previamente acontecer em uma expressão e for seguido por um parêntese esquerdo, ele é declarado pelo contexto como um nome de função, o compilador assume que a função retorna um valor *int*, e nada é assumido sobre seus argumentos. Além do mais, se uma declaração de função não incluir argumentos, como em

```
double atof();
```

presume-se também que nada deve ser assumido sobre os argumentos de *atof*; toda a checagem de parâmetro é desativada. Este significado especial da lista de argumento vazia serve para permitir que programas mais antigos em **C** sejam compilados com os compiladores modernos. Mas não é uma boa ideia usá-la com programas novos. Se a função usa argumentos, declare-os; se ela não usa argumentos, use *void*.

C99 Em C99 é um erro não declarar ou definir uma função antes de seu uso.

Dada *atof*, apropriadamente declarada, poderíamos escrever *atoi* (converte uma string para *int*) em termos de *atof*:

Programa 4.4: Função atoi() em termos de atof().

```
/* atoi: converte string s para inteiro usando atof */
int atoi (char s[])
{
    double atof (char s[]);
    return (int) atof(s);
}
```

Observe a estrutura das declarações e o comando *return*. O valor da expressão em

```
return (expressão);
```

é convertido para o tipo da função antes que o retorno seja feito. Portanto, o valor de *atof*, um *double*, é convertido automaticamente para *int* quando ele aparece em *return*, pois a função *atoi* retorna um *int*. Entretanto, esta operação potencialmente descarta informações, de modo que alguns compiladores avisam o fato. O molde informa explicitamente que a operação é feita, e retira qualquer aviso do compilador.

**Exercício 4.2** Estenda *atof* de modo que possa ser seguido por *e* ou *E* e um expoente com sinal opcional. ■

## 4.3 Variáveis Externas

Um programa **C** consiste em um conjunto de objetos externos, que são ou variáveis ou funções. O adjetivo "*externo*" é usado principalmente em contraste com "*interno*", que descreve os argumentos e as variáveis automáticas definidos dentro das funções. **Variáveis externas** são definidas fora de qualquer função, e são potencialmente disponíveis para muitas funções. Funções, por sua vez, são sempre externas, porque **C** não permite que elas sejam definidas dentro de outras funções. Por *default*, variáveis externas e funções têm a propriedade de que todas as referências a elas pelo mesmo nome, mesmo de funções compiladas separadamente, são referências à mesma coisa. (O padrão chama a esta propriedade **linkedição externa**.) Neste sentido, variáveis externas são análogas aos blocos **COMMON** do **Fortran** ou às variáveis no bloco mais externo em **Pascal**. Veremos posteriormente como definir variáveis externas e funções que são visíveis somente dentro de um único arquivo-fonte.

Visto que as variáveis externas são acessíveis globalmente, elas fornecem uma alternativa para argumentos de funções e valores de retorno na comunicação de dados entre funções. Qualquer função pode acessar uma variável externa pela referência ao seu nome se este tiver sido declarado de alguma forma.

Se um número grande de variáveis devem ser compartilhadas entre funções, variáveis externas são mais convenientes e eficientes que longas listas de argumentos. Como indicado no Capítulo 1, entretanto, este raciocínio deve ser aplicado com alguma cautela, porque ele pode ter um efeito negativo na estrutura de programas, e levar a programas com muitas conexões de dados entre funções.

As variáveis externas são também úteis devido ao seu maior escopo e vida útil. As **variáveis automáticas** são internas a uma função; elas existem quando a função é entrada, e desaparecem quando a deixam. As variáveis externas, por outro lado, são permanentes, de modo que retêm valores de uma chamada de função para a seguinte. Assim, se duas funções tiverem que compartilhar os mesmos dados, e nenhuma chama a outra, geralmente é mais conveniente que os dados compartilhados sejam mantidos em variáveis externas, ao invés de passados para dentro e fora das funções por meio de argumentos.

Vamos examinar este ponto num exemplo maior. O problema é escrever um programa de calculadora que forneça os operadores $+, -, *$ e $/$. Por ser fácil de implementar, a calculadora usará a **notação polonesa reversa** ao invés da **infixada**. (A notação polonesa reversa é usada, por exemplo, pelas calculadoras de bolso da **Hewlett-Packard**, além de linguagens como **Forth** e **Postscript**.)

Em **notação polonesa reversa**, cada operador segue seus *operandos*; uma *expressão infixada* tal como

$(1-2)*(4+5)$

é entrada como

$1\ 2\ -4\ 5\ +\ *$

Parênteses não são necessários; a notação não é ambígua, desde que saibamos quantos operandos cada operador espera.

A implementação é muito simples. Cada operando é colocado numa *pilha*; quando um operador aparece, o número apropriado de *operandos* (dois para operadores binários) é *desempilhado*, o operador é aplicado aos mesmos e o resultado *empilhado*. No exemplo acima, 1 e 2 são empilhados e então trocados por sua diferença, $-1$. Depois, 4 e 5 são empilhados e então trocados pela sua soma 9. O produto de $-1$ e 9, que é $-9$, os substitui na pilha. O valor no topo da pilha é removido

e impresso quando for encontrado o final da linha de entrada.

A estrutura do programa é, então, um laço que executa a operação apropriada em cada operador e operando à medida em que for aparecendo:

```
while (próximo operador ou operando não é fim-de-arquivo)
   if (número)
     empilha-o
   else if (operador)
     desempilha operandos
     realiza operação
     empilha resultado
   else if (nova-linha)
     desempilha e Imprime topo da pilha
   else
     erro
```

As operações de *push* e *pop* de uma pilha são triviais[1], mas quando incluímos a detecção e recuperação de erros, elas são grandes o bastante para que sejam colocadas em uma função separada, ao invés de repetir o código em todo o programa. Também deve haver uma função separada para trazer o próximo operador ou operando da entrada.

A principal decisão do projeto que ainda não foi discutida é onde colocar a pilha, isto é, que rotinas a acessarão diretamente. Uma possibilidade é mantê-la em *main*, e passar a pilha e a posição corrente da mesma para as rotinas que empilham e desempilham. Mas *main* não necessita conhecer as variáveis que controlam a pilha; ela deve pensar apenas em termos de empilhamento e desempilhamento. Assim, decidimos declarar a pilha e sua informação associada como variáveis externas acessíveis às funções *push* e *pop* mas não a *main*.

A tradução deste esboço em código é fácil. Se por ora pensarmos no programa como estando em um único arquivo-fonte, ele se parecerá com este esboço:

```
#include's
#define's

declarações de função para main

main () { ... }

variáveis externas para push e pop

  void push(double f) { ... }
  double pop(void) { ... }
  int getop(char s[]) { ... }
rotinas chamadas por getop
```

Mais tarde discutiremos como isto pode ser dividido em dois ou mais arquivos-fonte. A função *main* é um laço contendo um grande *switch* no tipo de operador ou operando; este é um uso mais típico do *switch* que o visto na Seção 3.4.

Programa 4.5: Função main() da calculadora polonesa reversa.

```
#include <stdio.h>
#include <stdlib.h> // para atof ()
```

[1] NT: usamos aqui os nomes clássicos para operações de pilha para empilhar (push) e desempilhar (pop).

```c
#define MAXOP 100 // tamanho máximo operando ou operador
#define NUMERO '0' // sinaliza que número foi encontrado

int getop(char []);
void push(double);
double pop(void);
/* calculadora polonesa reversa */
int main ()
{
    int tipo;
    double op2;
    char s[MAXOP];

    while ((tipo = getop(s)) != EOF) {
        switch (tipo) {
        case NUMERO:
            push(atof(s));
            break;

        case '+':
            push(pop() + pop());
            break;

        case '*':
            push(pop () * pop());
            break;

        case '-':
            op2 = pop();
            push (pop() - op2);
            break;

        case '/':
            op2 = pop();

            if (op2 != 0.0) {
                push(pop() / op2);
            } else {
                printf ("erro: divisor zero\n");
            }

            break;

        case '\n':
            printf("\t%.8g\n", pop());
            break;

        default:
            printf ("Erro: comando desconhecido %s\n", s);
            break;
        }
    }

    return 0;
}
```

Como $+$ e $*$ são operadores comutativos, a ordem em que os operandos desempilhados são combinados é irrelevante, mas para $-$ e $/$ os operandos esquerdo e direito devem ser distinguidos. Em

```
push (pop() - pop());  // ERRADO
```

a ordem em que as duas chamadas de *pop* são avaliadas não é definida. Para garantir a ordem correta, é necessário desempilhar o primeiro valor para uma variável temporária como fizemos em *main*.

Programa 4.6: Funções push() e pop().

```
#define MAXVAL 100  // profundidade max.  da pilha

int pp = 0;         // próxima posicão livre na pilha
double val[MAXVAL]; // pilha de valores

/* push: empilha f na pilha de valores */
void push(double f)
{
    if (pp < MAXVAL) {
        val [pp++] = f;
    } else {
        printf ("Erro: pilha cheia %g\n", f);
    }
}

/* pop: retira e retorna valor do topo da pilha */
double pop(void)
{
    if (pp > 0) {
        return val [--pp];
    } else {
        printf ("Erro: pilha vazia\n");
        return 0.0;
    }
}
```

Uma variável é considerada externa se for definida fora de qualquer função. Assim, a pilha e seu índice, que devem ser compartilhados por *push* e *pop*, são definidos fora dessas funções. Mas *main* não faz referência à pilha ou a seu índice — a representação pode ficar escondida.

Vamos agora passar à implementação de *getop*, a função que apanha o próximo operador ou operando. A tarefa é fácil. Pule os espaços e tabulações. Se o próximo caractere não for um dígito ou ponto decimal, retorne-o. Caso contrário, agrupe uma string de dígitos (que podem incluir um ponto decimal), e retorne *NUMERO*, o sinal de que um número foi apanhado.

Programa 4.7: Função getop().

```
#include <ctype.h>

int getch(void);
void ungetch(int);

/* getop: obtém próximo operador ou operando numérico */
int getop(char s[])
{
    int i, c;
```

```
    while ((s[0] = c = getch()) == ' ' || c == '\t')
        ;

    s[1] = '\0';

    if (!isdigit(c) && c != '.') {
        return c;    // não é um número
    }

    i = 0;

    if (isdigit(c)) // agrupa parte inteira
        while (isdigit(s[++i] = c = getch ()))
            ;

    if (c  == '.') // agrupa parte fracionária
        while (isdigit(s[++i] = c = getch()))
            ;

    s[i] = '\0';

    if (c != EOF) {
        ungetch(c);
    }

    return NUMERO;
}
```

O que são *getch* e *ungetch*? É frequente o caso em que um programa lendo sua entrada não pode determinar se leu o suficiente até que tenha lido demais. Um exemplo é agrupar caracteres que compõem um número: até o primeiro caractere que não seja um dígito ser visto, o número não está completo. Mas quando ele aparece, o programa leu um caractere a mais, o qual não estava preparado para receber.

O problema seria resolvido se fosse possível "*devolver*" para a entrada o caractere indesejado. Então, toda vez que o programa lesse um caractere a mais, este poderia ser devolvido à entrada, de modo que o resto do programa pudesse comportar-se como se nunca o tivesse visto. Felizmente, é fácil simular isto, escrevendo-se um par de funções cooperantes. *getch* obtém o próximo caractere a ser considerado; *ungetch* devolve um caractere para a entrada, de modo que a próxima chamada a *getch* o retorne.

Seu modo de funcionamento conjunto é simples. *ungetch* coloca o caractere devolvido num *buffer* compartilhado — um vetor de caracteres. *getch* lê a partir do *buffer*, se houver caracteres ali; e chama *getchar* caso contrário. Deve haver também uma variável de índice que indique a posição do caractere corrente no *buffer*.

Desde que o *buffer* e o índice são compartilhados por *getch* e *ungetch* e devem reter seus valores entre chamadas, eles devem ser externos a ambas as rotinas. Então podemos escrever *getch*, *ungetch* e suas variáveis compartilhadas como:

Programa 4.8: Funções getch() e ungetch().

```
#define TAMBUF 100

char buf[TAMBUF]; // buffer para ungetch
int pbuf = 0;     // próxima posição livre em buf
```

```
int getch(void) // obtém um caractere (talvez retornado)
{
    return (pbuf > 0) ? buf[--pbuf] : getchar ();
}

void ungetch(int c) // retorna caractere à entrada
{
    if (pbuf >= TAMBUF) {
        printf("ungetch: caracteres demais\n");
    } else {
        buf[pbuf++] = c;
    }
}
```

A biblioteca-padrão inclui uma função *ungetc* que fornece um caractere devolvido; veremos isso no Capítulo 7. Usamos um vetor para armazenar caracteres devolvidos ao invés de um único caractere, para ilustrar um enfoque mais genérico.

**Exercício 4.3** Dada uma infra-estrutura básica, é simples melhorar a calculadora. Inclua o operador de módulo (%) e previsão para números negativos. ■

**Exercício 4.4** Inclua comandos para imprimir o elemento do topo da pilha sem retirá-lo, para duplicá-lo, e para trocar a posição dos dois elementos do topo. Inclua um comando para apagar a pilha. ■

**Exercício 4.5** Inclua acesso às funções de biblioteca como *sin*, *exp* e *pow*. Veja <math.h> no Apêndice B, Seção B.4. ■

**Exercício 4.6** Inclua comandos para manipular variáveis. (Podemos facilmente incluir vinte e seis variáveis com nomes de única letra.) Inclua uma variável para o último valor impresso. ■

**Exercício 4.7** Escreva uma rotina *ungets(s)* que devolva uma *string* inteira para a entrada. *ungets(s)* precisa conhecer *buf* e *pbuf* ou precisa usar somente *ungetch*? ■

**Exercício 4.8** Suponha que nunca haverá mais que um caractere a ser devolvido para a entrada. Modifique *getch* e *ungetch* para acomodar esta suposição. ■

**Exercício 4.9** Nossas rotinas *getch* e *ungetch* não manipulam a devolução de *EOF* corretamente. Decida que propriedades devem existir se um *EOF* é devolvido para a entrada, e implemente seu projeto. ■

**Exercício 4.10** Uma alternativa para organização usa *lelinha* para ler toda uma linha de entrada; isso torna *getch* e *ungetch* desnecessários. Revise a calculadora para usar este método. ■

## 4.4 Regras de Escopo

As funções e variáveis externas que compõem um programa em **C** não precisam ser todas compiladas ao mesmo tempo; o texto-fonte do programa pode ser mantido em diversos arquivos, e podem ser

carregadas rotinas previamente compiladas nas bibliotecas. Entre as perguntas de interesse estão

- Como as declarações são escritas para que as variáveis sejam declaradas apropriadamente durante a compilação?
- Como são arrumadas as declarações para que todas as partes sejam adequadamente conectadas quando o programa é carregado?
- Como são organizadas as declarações para que haja somente uma cópia das mesmas?
- Como são inicializadas as variáveis externas?

Vamos discutir esses tópicos reorganizando o programa de calculadora em diversos arquivos. Na prática, a calculadora é muito pequena para que precise ser dividida, mas é uma boa ilustração dos problemas que surgem em grandes programas.

O *escopo* de um nome é a parte do programa dentro da qual o nome pode ser usado. Para uma variável automática declarada no início de uma função, o *escopo* é a função em que o nome é declarado. As variáveis locais do mesmo nome em diferentes funções não estão relacionadas. O mesmo acontece com os parâmetros da função, que são também variáveis locais.

O *escopo* de uma variável externa ou uma função dura desde o ponto em que é declarada até o final do arquivo sendo compilado. Por exemplo, se *main*, *pp*, *val*, *push* e *pop* são definidos em um arquivo, nesta ordem, isto é,

```
main(){...}

int pp = 0;
double val[MAXVAL];

void push(double f){ ... }

double pop(void) { ... }
```

então as variáveis *pp* e *val* podem ser usadas em *push* e *pop* simplesmente nomeando-as; nenhuma outra declaração é necessária. Mas estes nomes não são visíveis em *main*, assim como *push* e *pop*.

Por outro lado, se uma variável externa deve ser referenciada antes que seja definida, ou se ela for definida em um arquivo-fonte diferente daquele onde está sendo usada, então uma declaração *extern* é obrigatória.

É importante a distinção entre a declaração de uma *variável externa* e sua *definição*. Uma **declaração** anuncia as propriedades de uma variável (principalmente seu tipo); uma **definição** também separa uma área da memória. Se as linhas

```
int pp
double val[MAXVAL];
```

aparecerem fora de qualquer função, elas definem as variáveis externas *pp* e *val*, fazem com que uma área da memória seja separada, e também servem como declaração para o restante do arquivo-fonte. Por outro lado, as linhas

```
extern int pp;
extern double val[];.
```

declaram para o restante do arquivo-fonte que *pp* é um valor *int* e que *val* é um vetor de elementos do tipo *double* (cujo tamanho é determinado em outro lugar), mas não criam as variáveis ou reservam memória para elas.

Deve haver somente uma definição de uma variável externa entre todos os arquivos que compõem o programa-fonte; outros arquivos podem conter declarações *extern* para acessá-la. (Também pode haver declarações *extern* no arquivo contendo a definição.) Os tamanhos de vetor devem ser especificados com a definição, mas são opcionais com uma declaração *extern*.

A inicialização de uma variável externa é feita somente na definição.

Embora esta não seja uma organização provável para este programa, as funções *push* e *pop* poderiam ser definidas em um arquivo, e as variáveis *val* e *pp* definidas e inicializadas em outro. Depois estas definições e declarações seriam necessárias para uni-las:

*No arquivo1*:

```
extern int pp;
extern double val[];
void push(double f) { ... }
double pop(void) { ... }
```

*No arquivo2*:

```
int pp = 0;
double val[MAXVAL];
```

Como as declarações *extern* em *arquivo1* encontram-se à frente e fora das definições da função, elas aplicam-se a todas as funções; um conjunto de declarações é suficiente para todo o *arquivo1*. Esta mesma organização também seria necessária se as definições de *pp* e *val* estivessem depois do seu uso no arquivo.

## 4.5 Arquivos de Cabeçalho

Vamos agora considerar a divisão do programa calculadora em diversos arquivos-fonte, como seria se cada um dos componentes fosse substancialmente maior. A função *main* iria em um arquivo, que chamaremos *main.c*; *push*, *pop* e suas variáveis iriam em um segundo arquivo, *pilha.c*; *getop* iria em um terceiro, *getop.c*. Finalmente, *getch* e *ungetch* ficariam em um quarto arquivo, *getch.c*; nós os separamos um do outro porque, num programa real, eles viriam de uma biblioteca compilada separadamente.

Há mais uma coisa para nos preocuparmos — as definições e declarações compartilhadas entre os arquivos. Queremos centralizar isso o máximo possível, de modo que haja apenas uma cópia para obter e manter à medida que o programa evolui. Por isso, colocaremos este material comum em um arquivo de cabeçalho, *calc.h*, que será incluído onde for necessário. (A linha *#include* é descrita na Seção 4.11.) O esboço do programa resultante você encontrará na Figura 4.1.

Há um preço a pagar entre o desejo que cada arquivo tenha acesso somente à informação que precisa para o seu trabalho e a realidade prática de que é mais difícil se manter mais arquivos de cabeçalho. Até um certo tamanho moderado de programa, é provavelmente melhor ter um arquivo de cabeçalho que contenha tudo a ser compartilhado entre duas partes quaisquer do programa; essa é a decisão que fizemos aqui. Para um programa muito maior, mais organização e mais cabeçalhos seriam necessários.

## 4.6 Variáveis Estáticas

As variáveis *pp* e *val* em *pilha.c*, e *buf* e *pbuf* em *getch.c*, servem para uso privado das funções em seus arquivos-fonte respectivos, e não foram feitas para que sejam acessadas de nenhum outro lugar. A declaração *static*, aplicada a uma variável externa ou função, limita o escopo desse objeto ao restante do arquivo-fonte sendo compilado. As variáveis externas *static*, portanto, fornecem uma forma de esconder nomes como *buf* e *pbuf* na combinação *getch-ungetch*, que deve ser externa para que possam ser compartilhadas, mas que não devem ser visíveis aos usuários de *getch* e *ungetch*.

O armazenamento estático é especificado prefixando-se a declaração normal com a palavra *static*. Se as duas rotinas e as duas variáveis são compiladas em um arquivo, como em

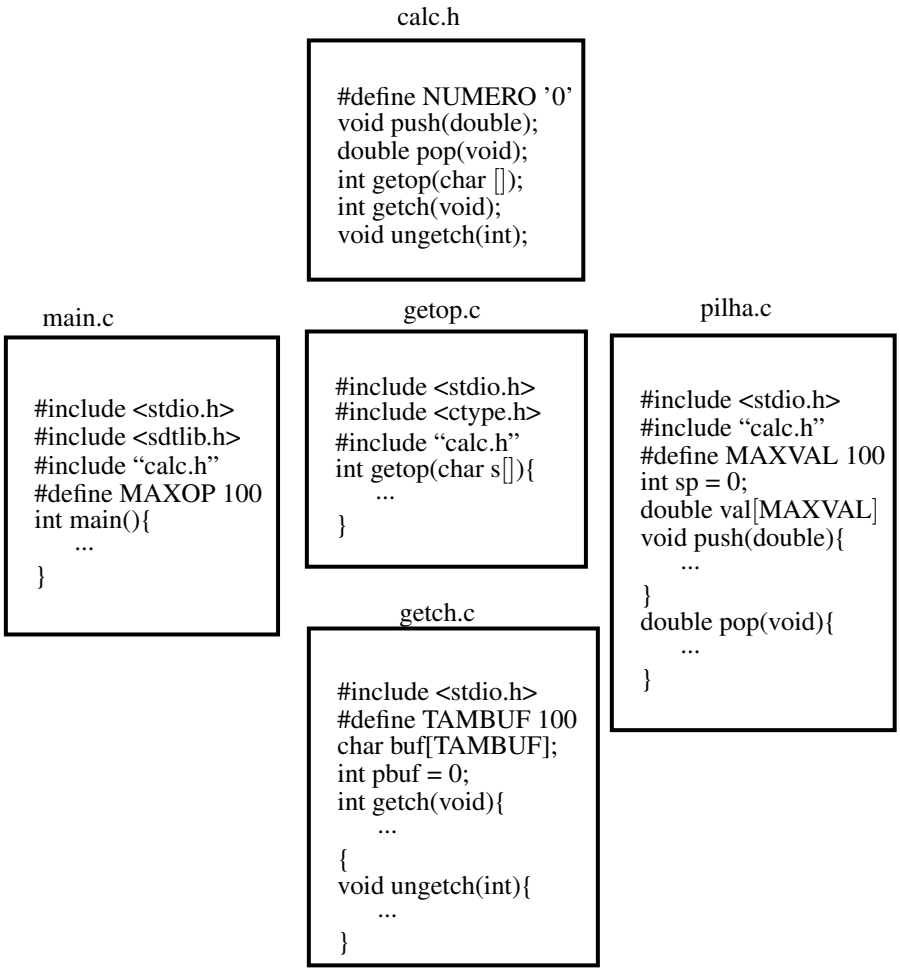



Figura 4.1: Programa calc.c.

```
static char buf[TAMBUF]; // buffer para ungetch
static int pbuf = 0;  // próxima posição livre em buf

int getch(void) { ... }
void ungetch(int c) { ... }
```

então nenhuma outra rotina será capaz de acessar *buf* e *pbuf*, e esses nomes não entrarão em conflito com os mesmos nomes em outros arquivos do mesmo programa. Da mesma forma, as variáveis que *push* e *pop* usam para manipulação da pilha podem ficar escondidas, declarando-se *pp* e *val* como *static*.

A declaração *static* externa é mais usada para variáveis, mas também pode ser aplicada a funções. Normalmente, os nomes de função são globais, visíveis a qualquer parte do programa inteiro. Entretanto, se uma função for declarada *static*, seu nome fica invisível fora do arquivo em que está sendo declarada.

A declaração *static* pode também ser aplicada a variáveis internas. As variáveis *static* internas são locais a uma função em particular assim como as variáveis automáticas, mas, diferente das automáticas, elas permanecem em existência independente da ativação ou desativação da função. Isso significa que as variáveis *static* internas fornecem uma forma de armazenamento privado e permanente dentro de uma função.

**Exercício 4.11** Modifique *getop* de forma que não precise usar *ungetch*. Dica: use uma variável *static* interna. ■

## 4.7 Variáveis em Registradores

Uma declaração *register* diz ao compilador que a variável em questão será extensamente utilizada. A ideia é que as variáveis *register* devem ser colocadas em registradores da máquina, o que pode resultar em programas menores e mais rápidos. Mas os compiladores são livres para ignorar o aviso.

A declaração *register* tem o seguinte formato:

```
register int x;
register char c;
```

e assim por diante. A declaração *register* só pode ser aplicada a variáveis automáticas e aos parâmetros formais de uma função. Neste último caso, ela tem o formato

```
f(register unsigned m, register long n)
{
   register int i;
   ...
}
```

Na prática, há restrições para as variáveis *register*, refletindo as realidades do hardware básico. Somente algumas poucas variáveis em cada função podem ser mantidas em registradores, e somente certos tipos são permitidos. As declarações de registrador em excesso, entretanto, são inofensivas, pois a palavra *register* é ignorada para declarações em excesso ou não permitidas. E não é possível tomar o endereço de uma variável *register* (um tópico que veremos melhor no Capítulo 5), não importa se a variável está realmente colocada em um registrador. As restrições específicas sobre número e tipos de variáveis de registrador variam de máquina para máquina.

## 4.8 Estrutura de Bloco

**C** não é uma linguagem com estrutura de bloco no sentido de **Pascal** ou linguagens semelhantes, pois as funções não podem ser definidas dentro de outras funções. Por outro lado, as variáveis podem ser definidas seguindo uma estrutura de bloco. Declarações de variáveis (incluindo inicializações) podem seguir o abre-chaves que introduz qualquer comando composto, e não apenas uma que inicia uma função. Variáveis declaradas desta forma escondem quaisquer variáveis com nomes idênticos em outros blocos, e permanecem em existência até o fecha-chaves correspondente. Por exemplo, em

```
if (n > 0){
    int i; // declara um novo i

    for (i = 0; i < n: i++)
       ...
}
```

o escopo da variável *i* é o desvio "verdadeiro" de *if*; este não tem nenhuma relação com qualquer outro *i* fora do bloco. Uma variável automática declarada e inicializada em um bloco é inicializada toda vez que o bloco é entrado. Uma variável *static* é inicializada somente na primeira vez que o bloco é entrado.

As variáveis automáticas, incluindo parâmetros formais, também escondem variáveis externas e funções do mesmo nome. Dadas as declarações

```
int x;
int y;

f(double x)
{
    double y;
    ...
}
```

então dentro da função *f*, as ocorrências de *x* referem-se ao parâmetro, que é um *double*; fora de *f*, elas referem-se ao *int* externo. O mesmo é válido para a variável *y*.

Por questão de estilo, é melhor evitar nomes de variável que ocultam nomes em um escopo mais externo; o potencial para confusão e erro é muito maior.

## 4.9 Inicialização

A inicialização tem sido mencionada de passagem por muitas vezes até agora, mas sempre perifericamente quando da referência a outro tópico. A presente seção resume algumas das regras, uma vez que já discutimos as várias classes de armazenamento.

Na falta de inicialização explícita, variáveis externas e variáveis estáticas são inicializadas com o valor **zero**; variáveis automáticas e em registradores têm valores iniciais indefinidos (i.e., lixo).

As variáveis escalares podem ser inicializadas quando são definidas, seguindo o nome com sinal de igual e uma expressão:

```
int x = 1;
char apostrofo = '\'';
long dia = 1000L * 60L * 60L * 24L; /* milissegundos/dia */
```

Para variáveis externas e estáticas, o inicializador deve ser uma expressão constante; a inicialização é feita uma vez, conceitualmente, antes que o programa comece sua execução. Para variáveis automáticas e de registrador, ela é feita toda vez que a função ou bloco é iniciado. Para variáveis automáticas e em registradores, o inicializador não se restringe a uma constante: ele pode ser qualquer expressão válida envolvendo valores previamente definidos, até mesmo chamadas de funções. Por exemplo, as inicializações do programa de pesquisa binária da Seção 3.3 poderiam ser escritas como:

```
int pesqbin(int x, int v[], int n)
{
    int inicio = 0;
    int fim = n-1;
    int meio;
    ...
}
```

ao invés de

```
int inicio, fim, meio;
inicio = 0;
fim = n-1;
```

Efetivamente, inicializações de variáveis automáticas são uma forma de abreviar comandos de atribuição. Qual forma de inicialização usar é uma questão de gosto. Usamos geralmente a atribuição explícita, porque inicializações em declarações são mais difíceis de se ver e se afastam muito da finalidade do uso.

Um vetor pode ser inicializado seguindo-se sua declaração com uma lista de inicializadores entre chaves e separados por vírgulas. Por exemplo, para inicializar um vetor *dias* com o número de dias em cada mês:

```
int dias [] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
```

Quando o tamanho do vetor é omitido, o compilador calculará o tamanho contando os inicializadores, dos quais há 12 neste caso.

Se houver menos inicializadores para um vetor do que um tamanho especificado, os outros serão zerados para variáveis externas ou estáticas, mas serão lixo para as automáticas. É um erro ter inicializadores além da capacidade do vetor. Não há uma forma de especificar a repetição de um inicializador, nem inicializar um elemento no meio de um vetor sem fornecer também todos os valores anteriores. Os vetores de caractere são um caso especial de inicialização; uma *string* pode ser usada ao invés de notação de chaves e vírgulas:

```
char padrao[] = "umas";
```

é uma abreviação para o equivalente mais extenso:

```
char padrao[] = { 'u', 'm', 'a', 's', '\0' };
```

Neste caso, o tamanho do vetor é cinco (quatro caracteres mais o terminador '\0').

## 4.10 Recursividade

Funções **C** podem ser usadas recursivamente, isto é, uma função pode chamar a si própria direta ou indiretamente. Considere a impressão de um número como *string* de caracteres. Como já dissemos, os dígitos são gerados na ordem contrária: os dígitos de mais baixa ordem estão disponíveis antes que os de mais alta ordem, mas eles devem ser impressos na ordem inversa.

Existem duas soluções para este problema. Uma é a de armazenarmos dígitos num vetor quando eles são gerados e então imprimi-los na ordem inversa, como fizemos na Seção 3.6 com *itoa*. A alternativa é uma solução recursiva, em que a função *impdec* primeiro chama a si mesma para imprimir os dígitos iniciais, e então imprime o dígito final. Mais uma vez, esta versão pode falhar com o maior número negativo.

Programa 4.9: Função impdec().

```
#include <stdio.h>

/* impdec: imprime n em decimal */
void impdec(int n)
{
    if (n < 0) {
        putchar ('-');
        n = -n;
    }
    if (n/10)
        impdec(n / 10);
    putchar (n % 10 + '0');
}
```

Quando uma função chama a si mesma recursivamente, cada chamada obtém um novo conjunto de todas as variáveis automáticas, independente do conjunto anterior. Assim, em *impdec(123)* o primeiro *impdec* recebe o argumento $n = 123$. Ele passa 12 para um segundo *impdec*, que por sua vez passa 1 a um terceiro. O *impdec* em terceiro nível imprime 1, depois retorna ao segundo nível. Esse *impdec* imprime 2, depois retorna ao primeiro nível. Esse imprime 3 e termina.

Um outro bom exemplo de recursão é o *quicksort*, um algoritmo de ordenação desenvolvido por **C.A.R. Hoare** em l962. Dado um vetor, um elemento é escolhido e os outros divididos em dois subconjuntos — aqueles menores que o elemento de partição e aqueles maiores ou iguais a ele. O mesmo processo é então aplicado recursivamente aos dois subconjuntos. Quando um subconjunto tiver menos de dois elementos, ele não precisa de outra ordenação; isso encerra a recursão.

Nossa versão de *quicksort* não é a mais rápida possível, mas é uma das mais simples. Usamos o elemento do meio de cada subvetor para o partionamento.

Programa 4.10: Implementação de quicksort.

```
/* qsort: ordena v[esq]...v[dir] em ordem crescente */
void quicksort(int v[], int esq, int dir)
{
    void troca(int v[], int i, int j);

    if (esq >= dir) { // não faz nada se vetor contém
        return;       // menos de dois elementos
    }

    troca (v, esq, (esq + dir) / 2); // move elemento de partição
    int ultimo = esq;                // para v[0]

    for (int i = esq + 1; i <= dir; i++) { // particiona
        if (v[i] < v[esq]) {
            troca (v, ++ultimo, i);
        }
    }

    troca (v, esq, ultimo); // recupera elemento de partição
    quicksort (v, esq, ultimo - 1);
    quicksort (v, ultimo + 1, dir);
}
```

Movemos a operação de *troca* para uma função troca separada porque ela ocorre três vezes em quicksort.

Programa 4.11: Função troca().

```
/* troca: intercâmbio de v[i] com v[j] */
void troca (int v[], int i, int j)
{
    int temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}
```

A biblioteca-padrão inclui uma versão de *quicksort* que pode ordenar objetos de qualquer tipo.

A recursão pode não economizar memória, pois em algum lugar uma pilha de valores sendo processados deve ser mantida. E nem será mais rápida. Mas o código recursivo é mais compacto, e normalmente muito mais fácil de se escrever e entender do que o equivalente não-recursivo. A recursão é especialmente conveniente para estruturas de dados recursivamente definidas, como árvores; veremos um bom exemplo na Seção 6.5.

**Exercício 4.12** Adapte as ideias de *impdec* para escrever uma versão recursiva de *itoa*; isto é, converta um inteiro para uma string chamando uma rotina recursiva. ■

**Exercício 4.13** Escreva uma versão recursiva da função *inverte(s)*, que inverte a string *s* nela mesma. ■

## 4.11 O Pré-Processador C

C fornece certas facilidades da linguagem por meio de um simples pré-processador, que conceitualmente é um primeiro passo na compilação. As duas características mais usadas são *#include*, para incluir o conteúdo de um arquivo durante a compilação; e *#define*, para substituir um código por uma sequência arbitrária de caracteres. Outras características descritas nesta seção incluem a compilação condicional e macros com argumentos.

### 4.11.1 Inclusão de Arquivos

A inclusão de arquivos facilita a manipulação de grupos de *#define*s e declarações (entre outras coisas). Qualquer linha fonte do formato

```
#include "arquivo"
```

ou

```
#include <arquivo>
```

é substituída pelo conteúdo do arquivo. Se o arquivo estiver entre aspas, a procura pelo arquivo normalmente começa onde o programa-fonte foi encontrado; se não for achado lá, ou se o nome estiver delimitado por < e >, a busca segue uma regra definida pela implementação para encontrar o arquivo. Um arquivo incluído pode conter linhas com *#include* dentro dele mesmo.

Há normalmente diversas linhas *#include* no início de um arquivo-fonte, para incluir comandos *#define* comuns e declarações *extern*, ou para acessar as declarações de protótipo de função para funções de biblioteca de cabeçalhos como <*stdio.h*>. (Estritamente falando, estes não precisam ser arquivos; os detalhes de como os cabeçalhos são acessados dependem da implementação.)

*#include* é a forma preferida de se juntar declarações para um grande programa. Ela garante que todos os arquivos-fonte serão supridos e com as mesmas definições e declarações de variáveis e elimina assim um tipo de erro particu1armente desagradável. Evidentemente, quando um arquivo de inclusão é alterado, todos os arquivos que dependem dele devem ser recompilados.

### 4.11.2 Substituição de Macros

Uma definição da forma

```
#define SIM 1
```

especifica uma substituição de macro do tipo mais simples — substituindo um nome código por um texto substituto. O nome em um *#define* tem a mesma forma que um nome de variável; o texto substituto é arbitrário. Normalmente o texto substituto é o resto da linha, mas uma definição longa pode ser continuada em diversas linhas colocando-se uma  ao final de cada linha a ser continuada. O escopo de um nome definido com *#define* vai do seu ponto de definição até o fim do arquivo-fonte sendo compilado. Uma definição pode usar definições anteriores. As substituições são feitas somente para códigos, e não dentro de strings entre aspas. Por exemplo, se *SIM* é um nome definido não haverá substituição em *printf("SIM")* ou em *SIMPLES*.

Qualquer nome pode ser definido com qualquer texto substituto. Por exemplo,

```
#define sempre for (;;) // loop infinito
```

define uma nova palavra, *sempre*, para um loop infinito.

Também é possível definir macros com argumentos, de modo que o texto substituto pode ser diferente para diferentes chamadas da macro. Como exemplo, definamos uma macro chamada *max*:

```
#define max(A,B) ((A) > (B) ? (A) : (B))
```

Embora pareça uma chamada de função, o uso de *max* toma-se um código em linha. Cada ocorrência de um parâmetro formal (aqui *A* ou *B*) será substituída pelo argumento real correspondente. Assim, a linha

```
x = max (p+q, r+s);
```

sera substituída pela linha

```
x = ((p+q) > (r+s) ? (p+q) : (r+s));
```

Enquanto os argumentos forem tratados de forma coerente, esta macro servirá para qualquer tipo de dado; não há necessidade de diferentes tipos de *max* para diferentes tipos de dados, como haveria para funções. Se você examinar a expansão de *max*, notará algumas armadilhas. As expressões são avaliadas duas vezes; isso é mau se elas envolverem efeitos colaterais como operadores de incremento ou entrada e saída. Por exemplo,

```
max(i++, j++) // ERRADO
```

incrementará o maior valor duas vezes. Algum cuidado deve ser tomado com parênteses para assegurar a preservação da ordem de avaliação; considere o que acontece quando a macro

```
#define quad(x) x * x // ERRADO
```

é chamada como *quad(z+1)*.

Apesar disso, as macros são valiosas. Um exemplo prático vem de <*stdio.h*>, em que *getchar* e *putchar* são normalmente definidos como macros para evitarem o trabalho extra de uma chamada de função, em tempo de execução, por caractere processado. As funções em <*ctype.h*> são também geralmente implementadas como macros.

Os nomes podem ser indefinidos com *#undef*, geralmente para assegurar que a rotina seja realmente uma função, e não uma macro:

```
#undef getchar

int getchar (void) {...}
```

Os parâmetros formais não são substituídos dentro de strings. Se, entretanto, um nome de parâmetro for precedido por um # no texto substituto, a combinação será expandida em uma string entre aspas com o parâmetro substituído pelo argumento real. Isso pode ser combinado com a concatenação de string para fazer, por exemplo, uma macro de depuração de impressão:

```
#define dprint(expr) printf(#expr " = %g\n", expr)
```

Quando esta é chamada, como em

```
dprint(x/y);
```

a macro é expandida para

```
printf("x/y" "=%g\n", x/y);
```

e as strings são concatenados, de forma que o efeito é

```
printf("x/y=%g\n", x/y);
```

Dentro do argumento real, cada " é substituída por \" e cada \ por \\, de forma que o resultado é uma constante de string de caracteres válida. O operador de pré-processador ## fornece uma forma de concatenar argumentos reais durante expansão de macro. Se um parâmetro no texto substituto for adjacente a ##, o parâmetro é substituído pelo argumento real, o ## e espaços em branco ao redor são removidos, e o resultado é reanalisado. Por exemplo, a macro *junta* concatena seus dois argumentos:

```
#define junta(frente, fundo) frente ## fundo
```

assim, *junta(nome, 1)* cria o código *nome1*.

As regras para os usos encaixados de ## são um tanto obscuras; maiores detalhes podem ser encontrados no Apêndice A.

**Exercício 4.14** Defina uma macro *troca(t,x,y)* que troque o valor de dois argumentos do tipo *t*. (A estrutura em bloco ajudará.) ■

### 4.11.3 Inclusão Condicional

É possível controlar o próprio pré-processamento com comandos condicionais que são avaliados durante o pré-processamento. Isto gera uma forma de incluir o código seletivamente, dependendo do valor de condições avaliadas durante a compilação.

A linha *#if* avalia uma expressão inteira constante (que não pode incluir *sizeof*, moldes ou constantes enumeradas). Se a expressão for diferente de zero, as linhas subsequentes até um *#endif* ou *#elif* ou *#else* são incluídas. (O comando do pré-processador *#elif* é parecido com *else-if*.) A expressão *defined(nome)* em um *#if* é 1 se o nome tiver sido definido, e 0 caso contrário.

Por exemplo, para termos certeza de que o conteúdo de um arquivo *cabec.h* está incluído somente uma vez, o conteúdo do arquivo é rodeado por uma condicional como esta:

```
#if !defined (CABEC)
#define CABEC
/* conteúdo de cabec.h segue aqui */
#endif
```

A primeira inclusão *de cabec.h* define o nome *CABEC*; as inclusões subsequentes encontrarão o nome definido e saltarão para o *#endif*. Um estilo semelhante pode ser usado para evitar a inclusão de arquivos múltiplas vezes. Se este estilo for usado coerentemente, então cada cabeçalho pode incluir quaisquer outros cabeçalhos dos quais ele dependa, sem que o usuário do cabeçalho tenha que lidar com a interdependência.

Esta sequência testa o nome *SISTEMA* para decidir que versão de um cabeçalho deverá ser incluída:

```
#if SISTEMA == SYSV
#define CABEC "sysv.h"
#elif SISTEMA == BSD
#define CABEC "bsd.h"
#elif SISTEMA == MSDOS
#define CABEC "msdos.h"
#else
#define CABEC "default.h"
#endif
#include CABEC
```

As linhas *#ifdef* e *#ifndef* são formas especializadas que testam se um nome é definido. O primeiro exemplo de *#if* acima poderia ter sido escrito

```
#ifndef HDR
#define HDR

/* conteúdo de cabec.h segue aqui */

#endif
```

# 5. Ponteiros e Vetores

Um *ponteiro* é uma variável que contém o endereço de outra variável. Ponteiros são muito usados em **C**, em parte porque eles são, às vezes, a única forma de se expressar uma computação, e em parte porque normalmente levam a um código mais compacto e eficiente que o obtido de outras formas. Os ponteiros e vetores são intimamente relacionados; este capítulo também explora esta relação, e mostra como podemos tirar proveito dela.

Ponteiros têm sido comparados ao comando *goto* como uma forma maravilhosa de se criar programas impossíveis de entender. Isto é certamente verdade quando eles são usados sem cuidado, e é fácil criar ponteiros que apontem para algum lugar inesperado. Com disciplina, entretanto, ponteiros podem ser usados para se obter clareza e simplicidade. Este é o aspecto que tentaremos ilustrar.

A principal mudança no **ANSI C** é tornar explícitas as regras sobre como os ponteiros podem ser manipulados, efetivamente afirmando o que os bons já praticam e bons compiladores já forçam. Além disso, o tipo *void* * (ponteiro para *void*) substitui *char* * como tipo apropriado para um ponteiro genérico.

## 5.1 Ponteiros e Endereços

Vamos começar com um esquema simplificado de como a memória é organizada. Uma máquina típica tem um vetor de células de memória consecutivamente numeradas ou endereçadas, que podem ser manipuladas individualmente ou em grupos contíguos. Uma situação comum é que qualquer *byte* pode ser um *char*, um par de células de um *byte* pode ser tratado como um inteiro *short*, e quatro *bytes* adjacentes formam um inteiro *long*. Um ponteiro é um grupo de células (normalmente duas ou quatro) que podem conter um endereço. Assim, se *c* é um *char* e *p* um ponteiro que aponta para ele, poderíamos representar a situação como na Figura 5.1.

O operador unário & fornece o endereço de um objeto, de forma que o comando

```
p = &c;
```

atribui o endereço de *c* à variável *p*, e diz-se que p "*aponta para*" *c*. O operador & só se aplica a objetos na memória: variáveis e elementos de vetor. Ele não pode ser aplicado a expressões,

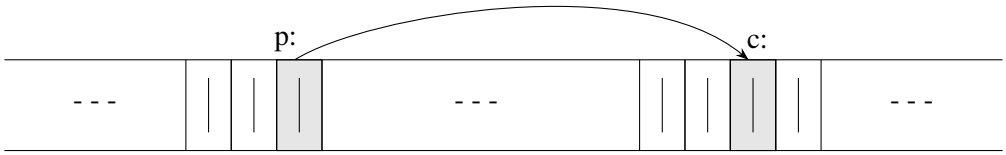



Figura 5.1: Ponteiro

constantes ou variáveis da classe registrador.

O operador unário $*$ é o operador de indireção ou desreferenciação; quando aplicado a um ponteiro, ele acessa o objeto que o ponteiro aponta. Suponha que *x* e *y* sejam inteiros e que *ip* seja um ponteiro para *int*. Esta sequência artificial mostra como declarar um ponteiro e como usar *&* e $*$:

```
int x = 1, y = 2, z[10];
int *ip;    // ip é um ponteiro para int

ip = &x;    // ip agora aponta para x

y = *ip;    // y agora é 1
*ip= 0;     // x agora é 0
ip = &z[0]; // ip agora aponta para z[0]
```

As declarações de *x*, *y* e *z* já nos são familiares. A declaração do ponteiro *ip*,

```
int *ip;
```

é um **mnemônico**; ela diz que a expressão **ip* é um *int*. A sintaxe da declaração para uma variável tem forma similar à sintaxe de expressões em que a variável pode aparecer. Este raciocínio é útil em todos os casos envolvendo também declarações de função. Por exemplo,

```
double *dp, atof(char *);
```

diz que na expressão $*dp$ e *atof(s)* possuem valores do tipo *double*, e que o argumento de *atof* é um ponteiro para *char*.

Você também deve observar que a declaração implica que um ponteiro restringe-se a apontar para algum tipo particular de objeto: cada ponteiro aponta para um tipo de dado especifico. (Existe uma exceção: um "ponteiro para *void*" é usado para conter qualquer tipo de ponteiro mas não pode ser desreferenciado por si mesmo. Voltaremos a isto na Seção 5.11.)

Se *ip* aponta para o inteiro *x*, então **ip* pode ocorrer em qualquer contexto onde *x* poderia, então

```
*ip = *ip + 10;
```

incrementa **ip* de 10.

Os operadores unários $*$ e *&* tem precedência maior que os operadores aritméticos, de forma que a atribuição

```
y = *ip + 1
```

toma o valor do objeto apontado por *ip*, adiciona 1 e atribui o valor a *y*, enquanto

```
*ip += 1
```

incrementa o objeto apontado por *ip*, da mesma forma que

```
++*ip
```

e

```
(*ip)++
```

Os parênteses são necessários neste último exemplo; sem eles, a expressão incrementaria *ip* ao invés do que ele aponta, pois os operadores unários como * e ++ são avaliados da direita para a esquerda.

Finalmente, como ponteiros são variáveis, eles podem ser usados sem desreferenciação. Por exemplo, se *iq* é um outro ponteiro para *int*.

```
iq = ip
```

copia o conteúdo de *ip* para *iq*, fazendo com que *iq* aponte para qualquer coisa apontada por *ip*.

## 5.2 Ponteiros e Argumentos de Funções

Como **C** passa argumentos para funções usando passagem "*por valor*", a função chamada não pode alterar diretamente uma variável na função chamadora. Por exemplo, uma rotina de ordenação poderia trocar dois elementos fora de ordem com uma função chamada troca. Não é suficiente escrever

```
troca (a,b);
```

onde a função de troca é definida como

Programa 5.1: Função troca com erro.

```
void troca (int x, int y) /* ERRADO */
{
    int temp;

    temp = x;
    x = y;
    y =temp;
}
```

Por causa da chamada por valor, *troca* não pode afetar os argumentos *a* e *b* na rotina que a chamou. A função acima troca somente cópias de *a* e *b*.

A forma de obter o efeito desejado é fazer com que o programa que chamou passe ponteiros para os valores a serem alterados:

```
troca (&a, &b);
```

Como o operador & produz o endereço de uma variável, *&a* é um ponteiro para *a*. Na função *troca*, os parâmetros são declarados como sendo ponteiros, e os operandos são acessados indiretamente por meio deles.

Programa 5.2: Função troca correta.

```
void troca (int *px, int *py) // troca *px por *py
{
    int temp;

    temp = *px;
    *px = *py;
    *py = temp;
}
```

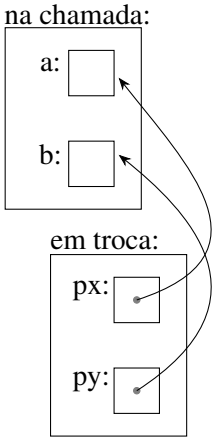



Figura 5.2: Referência

A Figura 5.2 mostra isso esquematicamente.

Os argumentos ponteiros permitem que uma função tenha acesso e altere objetos na função que a chamou. Como exemplo, considere uma função *leint* que execute conversão de entrada em formato livre dividindo um fluxo de caracteres em valores inteiros, um inteiro por chamada. *leint* deve retornar o valor que encontrou e também indicar o fim do arquivo quando não houver mais entrada. Esses valores devem ser passados de volta por vias separadas, porque não importa qual seja o valor usado para *EOF*, esse também poderia ser o valor de um inteiro da entrada.

Uma solução é fazer com que *leint* retorne o estado de fim de arquivo como seu valor da função, enquanto usa um argumento ponteiro para armazenar o inteiro convertido de volta à função que a chamou. Este também é o esquema usado por *scanf*; veja na Seção 7.4.

O laço a seguir preenche um vetor com inteiros por meio de chamadas a *leint*:

```
int n, vetor [TAMANHO], leint (int *);

for (n = 0; n < TAMANHO && leint (&vetor[n]) != EOF; n++)
   ;
```

Cada chamada define *vetor[n]* para o próximo inteiro encontrado na entrada e incrementa *n*. Observe que é essencial passar o endereço de *vetor[n]* para *leint*. Caso contrário, não há uma forma de *leint* comunicar o inteiro convertido de volta para a função que a chamou.

Nossa versão de *leint* retorna *EOF* para indicar o fim de arquivo, zero se a próxima entrada não for um número, e um valor positivo se a entrada contém um número válido.

Programa 5.3: Função leint().

```
#include <ctype.h>
int getch (void);
void ungetch (int);

/* leint: lê próximo inteiro da entrada para *pn */
int leint(int* pn)
{
    int c, sinal;

    while (isspace (c = getch())) // salta espaço
        ;
```

```
    if (!isdigit(c) && c != EOF && c != '+' && c != '-') {
        ungetch (c); // não é um número
        return 0;
    }

    sinal = (c == '-') ? -1 : 1;

    if (c == '+' || c == '-') {
        c = getch();
    }

    for (*pn = 0; isdigit(c); c = getch()) {
        *pn = 10 * *pn + (c - '0');
    }

    *pn *= sinal;

    if (c != EOF) {
        ungetch(c);
    }

    return c;
}
```

Em *leint*, **pn* é usado como uma variável comum do tipo *int*. Usamos também *getch* e *ungetch* (descritas na Seção 4.3) de modo que um caractere adicional lido possa ser devolvido para a entrada.

**Exercício 5.1** Como está escrito, *leint* trata um + ou − não seguido por um dígito como uma representação válida de zero. Conserte-o para colocar este caractere de volta na entrada. ■

**Exercício 5.2** Escreva *lefloat*, análoga a *leint* para ponto flutuante. Qual é o tipo do valor retornado por *lefloat*? ■

## 5.3 Ponteiros e Vetores

Em **C**, há uma forte relação entre ponteiros e vetores, tão forte que ponteiros e vetores deveriam ser discutidos juntos. Qualquer operação que possa ser feita com subscritos de um vetor pode ser feita com ponteiros. A versão com ponteiro será, em geral, mais rápida, mas, pelo menos para os iniciantes, mais difícil de compreender imediatamente.

A declaração

```
int a[10];
```

define um vetor *a* de tamanho 10, isto é, um bloco de *10* objetos consecutivos chamados *a[0], a[1], ..., a[9]*, como pode ser visto na Figura 5.3.

A notação *a[i]* refere-se ao elemento da *i-ésima* posição do vetor. Se *pa* for um ponteiro para um inteiro, declarado por

```
int *pa;
```

então a atribuição

```
pa = &a[0];
```

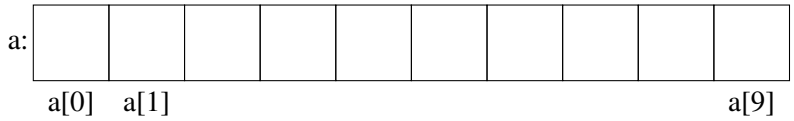



Figura 5.3: Array com 10 posições.

define *pa* de modo que aponte para o elemento zero de *a*; isto é, *pa* contém o endereço de *a[0]* (Figura 5.4).

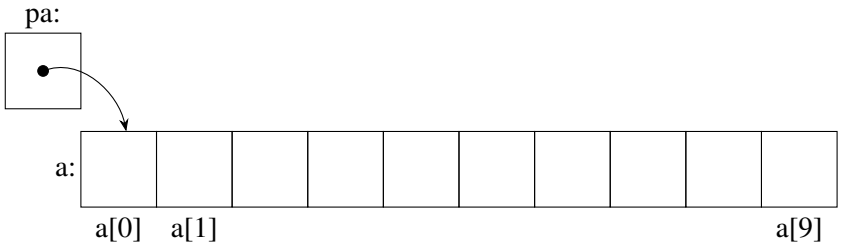



Figura 5.4: Ponteiro para a[0].

Agora a atribuição

```
x = *pa;
```

copiará o conteúdo de *a[0]* em *x*.

Se *pa* aponta para um elemento particular de um vetor, então por definição *pa + 1* aponta para o próximo elemento, *pa + i* aponta para *i* elementos após *pa*, e *pa - i* aponta para *i* elementos antes de *pa*. Assim, se *pa* aponta para *a[0]*,

```
*(pa+1)
```

refere-se ao conteúdo de *a[1]*. *pa + i* é o endereço de *a[i]*, e **(pa + i)* é o conteúdo de *a[i]* (Figura 5.5).

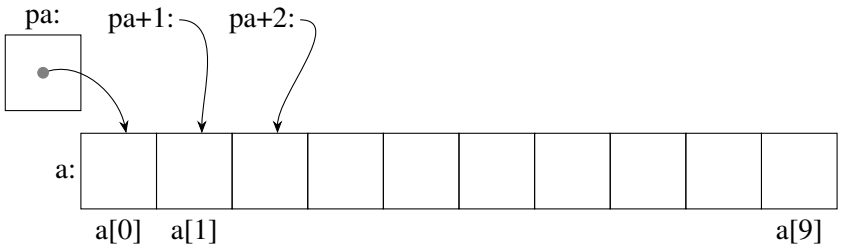



Figura 5.5: Incremento de ponteiro.

Essas observações aplicam-se independentemente do tipo ou tamanho das variáveis no vetor *a*. A definição de "*somar 1 a um ponteiro*", e por extensão, toda a aritmética com ponteiros, é a de que *pa + 1* aponta para o próximo objeto, e *pa + i* aponta para o *i-ésimo* objeto após *pa*.

A correspondência entre indexação e aritmética com ponteiros é evidentemente muito estreita. Por definição, o valor de uma variável ou expressão do tipo vetor é o endereço do elemento zero do vetor. Assim, após a atribuição

```
pa = &a[0];
```

*pa* e *a* possuem valores idênticos. Como o nome de um vetor é sinônimo para o local do elemento inicial, a atribuição *pa=&a[0]* também pode ser escrita como

```
pa=a;
```

Ainda mais surpreendente, pelo menos à primeira vista, é o fato de que uma referência a *a[i]* pode ser escrita como **(a + i)*. Na avaliação de *a[i]*, **C** a converte para **(a + i)* imediatamente; as duas formas são equivalentes. Aplicando-se o operador *&* a ambas as partes dessa equivalência, segue que *&a[i]* e *a + i* também são idênticos: *a + i* é o endereço do *i-ésimo* elemento após *a*. Como outro lado da moeda, se *pa* é um ponteiro, expressões podem usá-lo como um subscrito; *pa[i]* é idêntico a **(pa + 1)*. Em suma, qualquer expressão de vetor e índice é equivalente a uma escrita com um ponteiro e um deslocamento.

Há uma diferença entre o nome de um vetor e um ponteiro que deve ser lembrada. Um ponteiro é uma variável, de forma que *pa=a* e *pa++* são operações válidas. Mas o nome de um vetor não é uma variável; construções como *a = pa* e *a++* são **ilegais**.

Quando o nome de um vetor é passado para uma função, o que é passado é a posição do elemento inicial. Dentro da função chamada, este argumento é uma variável local, e assim um argumento do tipo nome de vetor é um ponteiro, isto é, uma variável contendo um endereço. Podemos usar este fato para escrever uma outra versão de *strlen*, que calcula o tamanho de uma string de caracteres.

Programa 5.4: Outra versão de strlen().

```
/* strlen: retorna o tamanho de uma string s */
int strlen(char* s)
{
    int n;

    for (n = 0; *s != '\0'; s++) {
        n++;
    }

    return n;
}
```

Como *s* é um ponteiro, incrementá-lo é perfeitamente legal; *s++* não tem qualquer efeito na string de caracteres na função que chamou *strlen*, mas simplesmente incrementa a cópia privativa do ponteiro em *strlen*. Isso significa que todas as chamadas do tipo de

```
strlen ("primeiro programa"); // uma constante de string
strlen (vetor);               // char vetor[100];
strlen (ptr);                 // char *ptr;
```

funcionam.

Como parâmetros formais na definição de uma função, as construções

```
char s[];
```

e

```
char *s;
```

são equivalentes; preferimos a última porque diz mais explicitamente que o parâmetro é um ponteiro. Quando um nome de vetor é passado para a função, esta pode, de acordo com sua conveniência, acreditar que ela está manipulando um vetor ou um ponteiro, e manipulá-lo de acordo. Ela pode até mesmo usar as duas notações se lhe parecer apropriado e claro.

É possível passar parte de um vetor para uma função, passando um ponteiro para o início do subvetor. Por exemplo, se *a* é um vetor,

```
f (&a[2])
```

e

```
f (a+2)
```

passam para a função *f* o endereço do subvetor que inicia no elemento *a[2]*. Dentro de *f*, a declaração de parâmetros pode ser

```
f(int vet[]) {...}
```

ou

```
f(int *vet) {...}
```

Assim, do ponto de vista da função *f*, o fato de que o parâmetro se refere a uma parte de um vetor maior não traz nenhuma consequência.

Se pudermos ter certeza de que o elemento existe, também é possível a indexação invertida no vetor: $p[-1]$, $p[-2]$ e assim por diante são sintaticamente válidos, e referem-se aos elementos que precedem imediatamente a $p[0]$. Logicamente, é ilegal referir-se a objetos que não estejam dentro dos limites do vetor maior.

## 5.4 Aritmética com Endereços

Se *p* é um ponteiro para algum elemento de um vetor, então *p++* incrementa *p* para que aponte ao próximo elemento, e *p += i* o incrementa para apontar *i* elementos além do objeto para o qual *p* atualmente aponta. Estas e outras construções similares são as formas mais simples e comuns da aritmética com ponteiros ou endereços.

**C** é consistente e regular no que tange à aritmética com endereços; a integração de ponteiros, vetores e aritmética com endereços é uma das maiores vantagens da linguagem. Vamos ilustrar escrevendo um alocador de memória elementar. Existem duas rotinas. A primeira, *aloca(n)*, retorna um ponteiro *p* para *n* posições consecutivas de caracteres, as quais podem ser usadas pelo chamador de *aloca* para armazenar caracteres. A segunda, *libera(p)* libera a área adquirida para que possa ser reutilizada. As rotinas são "elementares" porque as chamadas a *libera* devem ser feitas na ordem inversa das chamadas a *aloca*. Isto é, a área de armazenamento gerenciada por *aloca* e *libera* é uma pilha. A biblioteca-padrão de **C** fornece funções análogas chamadas *malloc* e *free* que não têm tais restrições; na Seção 8.7 mostraremos como elas podem ser implementadas.

A implementação mais simples é fazer com que *aloca* forneça pedaços de um grande vetor de caracteres que nós chamaremos *bufaloc*. Este vetor é privado a *aloca* e *libera*. Como elas manipulam ponteiros, e não índices de vetores, nenhuma outra rotina precisa conhecer o nome do vetor, o qual pode ser declarado como sendo do tipo *static* no arquivo-fonte contendo *aloca* e *libera*, e invisível fora dele. Em implementações práticas, o vetor pode até mesmo não ter um nome; ele poderia ser obtido pela chamada a *malloc* ou pedindo-se ao sistema operacional um ponteiro para algum bloco de armazenamento não nomeado.

A outra informação necessária é saber quanto de *bufaloc* já foi usado. Usamos um ponteiro para o próximo elemento livre, chamado *aaloc*. Quando se pede *n* caracteres a *aloca*, ele verifica se há espaço em *bufaloc*. Se houver, *aloca* retorna o valor corrente de *aaloc* (i.e., o início do bloco

livre), e depois incrementa seu valor de *n* para apontar para o próximo bloco livre (Figura 5.6). Se não houver espaço, *aloca* retorna zero. *libera(p)* simplesmente define *aaloca* para *p* se este estiver dentro de *bufaloc*.

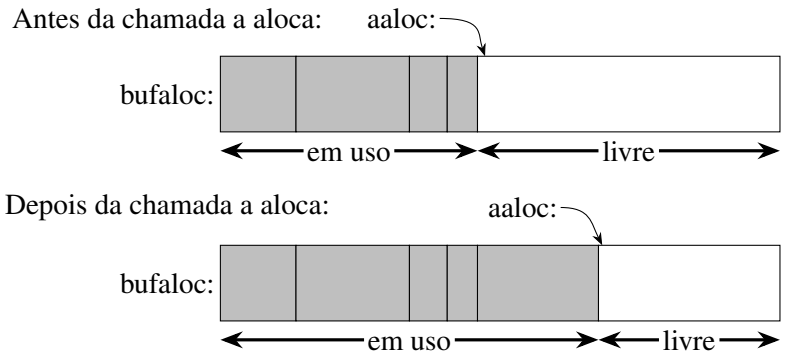



Figura 5.6: Alocação de memória.

Programa 5.5: Funções aloca() e libera().

```
#define TAMALOC 10000 // tamanho do espaço disponível

static char bufaloc[TAMALOC]; // espaço para aloca
static char* aaloc = bufaloc; // próxima posição livre

char* aloca(int n) //retorna ponteiro para n caracteres
{
    if (bufaloc + TAMALOC - aaloc >= n) {// se tem espaço
        aaloc += n;
        return aaloc - n; // ponteiro antigo
    } else { // não tem espaço
        return 0;
    }
}

void libera(char* p) // libera espaço apontado por p
{
    if (p >= bufaloc && p < bufaloc + TAMALOC) {
        aaloc = p;
    }
}
```

Em geral, um ponteiro pode ser inicializado assim como qualquer outra variável, embora os únicos valores significativos sejam zero ou uma expressão envolvendo os endereços de dados previamente definidos do tipo apropriado. A declaração

```
static char *aaloc = bufaloc;
```

define *aaloc* como sendo um ponteiro de caractere e o inicializa para apontar para o início de *bufaloc*, que é a próxima posição livre quando o programa se inicia. Isto também poderia ser escrito por

```
static char *aaloc = &bufaloc[0];
```

pois o nome do vetor é o endereço do elemento zero.

O teste

```
if (bufaloc + TAMALOC - aaloc >= n) { // se tem espaço
```

verifica se há espaço suficiente para satisfazer um pedido de *n* caracteres. Se houver, o novo valor de *aaloc* seria no máximo um além do final de *bufaloc*. Se o pedido puder ser satisfeito, *aloca* retorna um ponteiro para o início de um bloco de caracteres (observe a declaração da própria função). Se não, *aloca* deve retornar algum sinal de que não houve espaço. **C** garante que zero nunca é um endereço válido para dados, de modo que um valor de retorno com zero pode ser usado para sinalizar um evento anormal, neste caso, falta de espaço.

Os ponteiros e inteiros não são intercambiáveis. Zero é a única exceção: a **constante zero** pode ser atribuída a um ponteiro, e um ponteiro pode ser comparado com a constante zero. A constante simbólica *NULL* é normalmente usada no lugar de zero, como mnemônico para indicar mais claramente que este é um valor especial para um ponteiro. *NULL* é definido em <*stddef.h*>, porém aparece também em <*stdio.h*> e <*stdlib.h*>. Usaremos *NULL* daqui por diante.

Testes como

```
if (bufaloc + TAMALOC - aaloc >= n) { // se tem espaço
```

e

```
if (p >= bufaloc && p < bufaloc + TAMALOC)
```

mostram várias facetas importantes da aritmética com ponteiros. Primeiro, ponteiros podem ser comparados sob certas circunstâncias. Se *p* e *q* apontam para membros do mesmo vetor, então relações tais como ==, ! =, <, >=, etc. funcionam corretamente. Por exemplo,

```
p < q
```

é verdadeiro se *p* aponta para um elemento do vetor anterior ao apontado por *q*. Qualquer ponteiro pode ser comparado para testar igualdade ou desigualdade com o valor *NULL*. Entretanto o comportamento é indefinido para aritmética ou comparações com ponteiros que não apontem para membros do mesmo vetor. (Há uma exceção: o endereço do primeiro elemento após o fim de um vetor pode ser usado em aritmética de ponteiro.)

Segundo, já observamos que um ponteiro e um inteiro podem ser adicionados ou subtraídos. A construção

```
p + n
```

significa o *n-ésimo* elemento a partir da posição apontada por *p*. Isto é verdade independentemente do tipo de objeto para o qual *p* aponta; o compilador ajusta *n* de acordo com o tamanho dos objetos apontados por *p*; este tamanho é determinado pela declaração de *p*. Se um *int* tiver **quatro** bytes, por exemplo, o *int* será ajustado de **quatro** em **quatro**.

A subtração de ponteiros também é válida: se *p* e *q* apontam para membros de um mesmo vetor, e $p < q$, então $q - p + 1$ é o número de elementos de *p* a *q* inclusive. Este fato pode ser usado para escrevermos uma outra versão de *strlen*:

Programa 5.6: Função strlen() usando subtração de ponteiros.

```
/* strlen: retorna o tamanho da string s */
int strlen(char* s)
{
    char* p = s;

    while (*p != '\0') {
```

```
        p++;
    }

    return p - s;
}
```

na sua declaração, *p* é inicializado para *s*, isto é, para apontar para o primeiro caractere da string. No laço *while*, cada caractere por sua vez é examinado até que o '\0' ao final seja visto. Como *p* aponta para caracteres, *p++* avança *p* para o próximo caractere a cada vez, e *p - s* dá o número de caracteres avançados, isto é, o tamanho da string. (O número de caracteres na string poderia ser muito grande para ser armazenado em um *int*. O cabeçalho <*stddef.h*> define um tipo *ptrdiff_t* que é grande o suficiente para conter a diferença sinalizada de dois valores de ponteiro. Se fomos mais cuidadosos, entretanto, usaremos *size_t* para o tipo do retorno de *strlen*, combinando com a versão da biblioteca-padrão. *size_t* é o tipo inteiro não sinalizado retornado pelo operador *sizeof*.

A aritmética de ponteiros é coerente: se estivéssemos lidando com *float*, que ocupam mais espaço do que *char*, e se *p* fosse um ponteiro para *float*, *p++* avançaria para o próximo *float*. Assim, poderíamos escrever uma outra versão de *aloca* que mantenha valores *float* no lugar de *char*, simplesmente alterando *char* para *float* em *aloca* e *libera*. Todas as manipulações de ponteiro automaticamente levariam em consideração o tamanho do objeto apontado.

As operações válidas com ponteiros são atribuição de ponteiros do mesmo tipo, adição ou subtração de um ponteiro e um inteiro, subtração ou comparação de dois ponteiros com membros do mesmo vetor, e atribuição ou comparação com zero. Todas as outras operações aritméticas com ponteiros são **ilegais**. Não é válido somar dois ponteiros, ou multiplicar ou dividir ou deslocar ou mascarar ou adicionar *float* ou *double* aos mesmos, ou até mesmo, exceto por *void* *, atribuir um ponteiro de um tipo a um ponteiro de outro tipo sem um molde.

## 5.5 Ponteiros de Caractere e de Funções

Uma constante do tipo string de caracteres, escrita pela forma

```
"Eu sou uma string de caracteres"
```

é um vetor de caracteres. Na representação interna, o vetor é terminado com o caractere nulo '\0' para que programas possam encontrar o fim do mesmo. O tamanho de armazenamento é portanto um a mais do que o número de caracteres entre aspas.

Talvez a ocorrência mais comum de constantes do tipo string seja como argumento para funções, como em

```
printf("primeiro programa\n");
```

Quando uma string de caracteres como esta aparece num programa, o acesso a ela é feito por meio de um ponteiro de caractere; *printf* recebe um ponteiro para o início do vetor de caracteres. Isto é, uma constante do tipo string é acessada por um ponteiro para o seu primeiro elemento.

As constantes de string não precisam ser argumentos de função. Se mensagem é declarada como segue

```
char *mensagem;
```

então o comando

```
mensagem = "chegou a hora";
```

atribui a mensagem um ponteiro para o vetor de caracteres. Esta não é uma cópia da string: somente ponteiros estão envolvidos. **C** não fornece quaisquer operadores para processamento de toda uma string de caracteres como uma só unidade.

Há uma diferença importante entre estas definições:

```
char vmensagem[] = "chegou a hora"; // um vetor
char pmensagem = "chegou a hora";   // um ponteiro
```

*vmensagem* é um vetor com tamanho suficiente para conter a sequência de caracteres e '\0' que a inicializa. Os caracteres individuais dentro do vetor podem ser alterados, mas *vmensagem* sempre irá referir-se ao mesmo local de armazenamento. Por outro lado, *pmensagem* é um ponteiro, inicializado para apontar para uma constante do tipo string; o ponteiro pode subsequentemente ser modificado para que aponte para outro lugar, mas o resultado é indefinido se você tentar modificar o conteúdo da string.

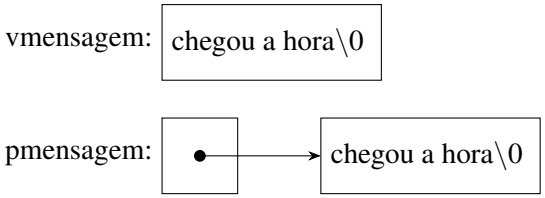



Figura 5.7: Ponteiros e vetores.

Ilustraremos mais aspectos de ponteiros e vetores estudando versões de duas funções úteis adaptadas da biblioteca-padrão. A primeira função é $strcpy(s,t)$, que copia a string $t$ para a string $s$. Seria ótimo apenas fazermos $s = t$, mas isto copia o ponteiro, e não os caracteres. Para copiar os caracteres, precisamos de um laço. A versão com vetor é a primeira:

Programa 5.7: Função strcpy() com vetor.

```
/* strcpy: copia t para s; versão com vetor subscrito */
void strcpy( char* s, char* t)
{
    int i;
    i = 0;

    while ((s[i] = t[i]) != '\0') {
        i++;
    }
}
```

Em contraste, aqui está uma versão de *strcpy* que usa ponteiros:

Programa 5.8: Função strcpy() com ponteiro.

```
/* strcpy: copia t para s; versão ponteiro */
void strcpy(char* s, char* t)
{
    while ((*s = *t) != '\0') {
        s++;
        t++;
    }
}
```

Como os argumentos são passados por valor, *strcpy* pode usar os parâmetros $s$ e $t$ de qualquer forma desejada. Aqui eles são ponteiros inicializados convenientemente, passados de um vetor para o outro com um caractere por vez, até que o '\0' que termina $t$ seja copiado para $s$.

Na prática, *strcpy* não seria escrito como escrevemos aqui. Os programadores experientes em **C** prefeririam usar

Programa 5.9: Função strcpy(), versão aprimorada.

```
/* strcpy: copia t para s; versão ponteiro 2 */
void strcpy(char* s, char* t)
{
    while ((*s++ = *t++) != '\0')
        ;
}
```

Isto move o incremento de *s* e *t* para a parte de teste do laço. O valor de `*t++` é o caractere que *t* apontava antes que *t* fosse incrementado; o ++ pós-fixado não modifica *t* antes que este caractere tenha sido trazido. Da mesma forma, o caractere é armazenado na antiga posição de *s* antes que *s* seja incrementado. Este caractere é também o valor que é comparado contra '\0' para controlar o laço. O resultado é a cópia dos caracteres de *t* para *s*, até o término '\0', inclusive.

Como abreviação final, observe que a comparação contra '\0' é redundante, pois a questão é simplesmente se a expressão é zero. Assim, a função provavelmente seria escrita por

Programa 5.10: Função strcpy(), versão final.

```
/* strcpy: copia t para s; versão ponteiro 3 */
void strcpy(char* s, char* t)
{
    while (*s++ = *t++)
        ;
}
```

Embora isso possa parecer enigmático à primeira vista, a conveniência na notação é considerável e o idioma deve ser dominado, inclusive porque você verá esta construção constantemente em programas **C**.

A função *strcpy* na biblioteca-padrão (<*string.h*>) retorna a string de destino como seu valor de função.

A segunda rotina que examinaremos é *strcmp(s,t)*, que compara as cadeias de caracteres *s* e *t*, e retorna, negativo, zero ou positivo se *s* é lexicograficamente menor, igual ou maior que *t*. O valor é obtido subtraindo-se os caracteres na primeira posição onde *s* e *t* são diferentes.

Programa 5.11: Função strcmp().

```
/* strcmp: retorna <0 se s<t, 0 se s==t, >0 se s>t */
int strcmp(char* s, char* t )
{
    int i;

    for (i = 0; s[i] == t[i]; i++) {
        if (s[i] == '\0') {
            return 0;
        }
    }

    return s[i] - t[i];
}
```

A versão de *strcmp* com ponteiros:

Programa 5.12: Função strcmp() com ponteiros.

```
/* strcmp: retorna <0 se s<t, 0 se s==t, >0 se s>t */
int strcmp(char* s, char* t )
{
    for ( ; *s == *t; s++, t++) {
        if (*s == '\0') {
            return 0;
        }
    }

    return *s - *t;
}
```

Desde que $++$ e $--$ são ambos operadores pré ou pós-fixados, outras combinações de $*$ com $++$ e $--$ ocorrem, embora com menos frequência. Por exemplo.

```
*--p
```

decrementa *p* antes de obter o caractere para o qual *p* aponta. De fato, o par de expressões

```
*p++ = val; // coloca val na pilha
val = *--p; // retira topo da pilha para val
```

são as formas-padrão para se colocar e retirar valores da pilha: veja a Seção 4.3.

O arquivo de cabeçalho <*string.h*> contém declarações para as funções mencionadas nesta seção, e mais uma série de outras funções de manipulação de string da biblioteca-padrão.

**Exercício 5.3** Escreva uma versão com ponteiros da função *strcat* que mostramos no Capítulo 2. $strcat(s,t)$ copia a string $t$ no fim da string $s$. ■

**Exercício 5.4** Escreva a função $strend(s,t)$, que retorna *1* se a string $t$ ocorrer no final da string $s$, e zero em caso contrário. ■

**Exercício 5.5** Escreva versões das funções de biblioteca *strncpy*, *strncat* e *strncmp*, que operam com os primeiros $n$ caracteres das suas cadeias no argumento. Por exemplo, `strncpy(s,t,n)` copia no máximo $n$ caracteres de $t$ para $s$. Descrições completas encontram-se no Apêndice B. ■

**Exercício 5.6** Reescreva programas apropriados dos capítulos e exercícios anteriores com ponteiros ao invés de indexação de vetores. Boas possibilidades incluem *lelinha* (Capítulo 1 e Capítulo 4), *atoi*, *itoa*, e suas variantes (Capítulo 2, Capítulo 3 e Capítulo 4), *inverte* (Capítulo 3) e *obtemop* (Capítulo 4). ■

## 5.6 Vetores de Ponteiros; Ponteiros para Ponteiros

Visto que ponteiros são variáveis, pode-se esperar o uso de vetores de ponteiros assim como outras variáveis. Vamos ilustrar tais construções escrevendo um programa que ordene um conjunto de linhas de texto em ordem alfabética, uma versão reduzida do utilitário *sort* do **UNIX**.

No Capítulo 3 apresentamos a função de ordenação **Shell** que ordena um vetor de inteiros, e no Capítulo 4 a melhoramos com uma ordenação rápida. O mesmo algoritmo funcionará, exceto que agora temos de manipular linhas de texto de tamanhos diferentes, e que, ao contrário dos inteiros, não podem ser comparadas ou movidas com uma única operação. Precisamos de uma representação de dados que seja eficiente e conveniente para linhas de tamanho variável.

E é neste ponto que o vetor de ponteiros é útil. Se as linhas a serem ordenadas estiverem armazenadas contiguamente num longo vetor de caracteres, então cada linha pode ser acessada por meio de um ponteiro para seu primeiro caractere. Os ponteiros podem ser armazenados num vetor. Duas linhas podem ser comparadas passando-se seus ponteiros para *strcmp*. Quando duas linhas fora de ordem tiverem de ser trocadas, os ponteiros no vetor de ponteiros são trocados, e não as próprias linhas de texto (Figura 5.8).

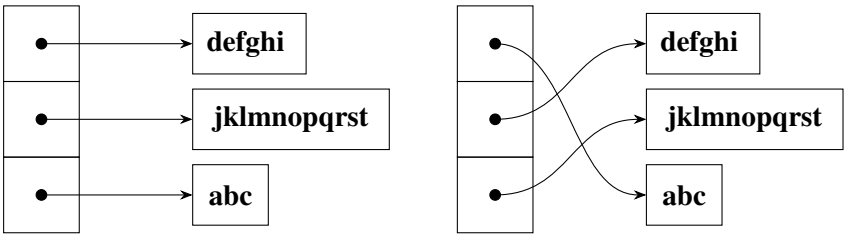



Figura 5.8: Vetor de ponteiros.

Isso elimina o duplo problema de um gerenciamento de espaço complicado e uma alta sobrecarga de trabalho resultante da movimentação das próprias linhas.

O processo de ordenação envolve três passos:

1. ler todas as linhas de entrada
2. ordená-las
3. imprimi-las em ordem

Como sempre, é melhor dividir o programa em funções que reflitam esta divisão natural, com a rotina principal controlando as outras funções. Vamos deixar de lado o processo de ordenação por um momento, e nos concentrar na estrutura de dados e a entrada e saída.

A rotina de entrada tem de coletar e armazenar os caracteres de cada linha, montando um vetor de ponteiros para as linhas. Ela também terá de contar o número de linhas de entrada, pois essa informação é necessária para a ordenação e impressão. Como a função de entrada da possui apenas de um número finito de linhas de entrada, ela pode retornar algum contador de linha ilegal como $-1$ se muitas linhas de entrada forem apresentadas.

A rotina de saída só precisa imprimir as linhas na ordem em que aparecem no vetor de ponteiros.

Programa 5.13: Programa para ordenar linhas de texto.

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

#define MAXLIN 5000     // máximo de linhas a ordenar

char* ptrlinha[MAXLIN]; // ponteiros para linhas texto

int lelinhas(char* ptrlinha[], int nlinhas );
void imprlinhas(char* ptrlinha[], int nlinhas);

void qsort(char* ptrlinha[], int esq, int dir);

/* ordena linhas de entrada */
int main ()
{
    int nlinhas; // número de linhas lidas

    if ((nlinhas = lelinhas(ptrlinha, MAXLIN)) >= 0) {
```

```
        qsort(ptrlinha, 0, nlinhas - 1);
        imprlinhas(ptrlinha, nlinhas);
        return 0;
    } else {
        printf("Erro: entrada muito grande\n");
        return 1;
    }
}

#define TAMMAX 1000 // tamanho máximo da linha
int lelinha(char*, int);
char* aloca(int);

/* lelinhas: lê linhas da entrada */
int lelinhas(char* ptrlinha[], int maxlin)
{
    int tam, nlinhas;
    char* p, linha[TAMMAX];
    nlinhas = 0;

    while ((tam = lelinha(linha, TAMMAX)) > 0)
        if (nlinhas >= maxlin || (p = aloca(tam)) == NULL) {
            return -1;
        } else {
            linha[tam - 1] = '\0'; // deleta nova-linha
            strcpy(p, linha);
            ptrlinha[nlinhas++] = p;
        }

    return nlinhas;
}

/* imprlinhas: imprime linhas na saída */
void imprlinhas(char* ptrlinha[], int nlinhas)
{
    for (int i = 0; i < nlinhas; i++) {
        printf("%s\n", ptrlinha[i]);
    }
}
```

A função *lelinha* vem da Seção 1.9. A principal mudança é na declaração de *ptrlinha*:

```
char *ptrlinha[MAXLIN];
```

diz que *ptrlinha* é um vetor de *MAXLIN* elementos, cada um deles sendo um ponteiro para um *char*. Isto é, *ptrlinha[i]* é um ponteiro de caractere, e **ptrlinha[i]* é o caractere para o qual ele aponta, o primeiro caractere da *i-ésima* linha de texto armazenada.

Como *ptrlinha* é o próprio nome do vetor, ele pode ser tratado como um ponteiro da mesma forma que nos nossos exemplos anteriores, e *imprlinhas* pode ser escrito como:

Programa 5.14: Outra versão de imprlinhas

```
/* imprlinhas: imprime linhas na saída */
void imprlinhas(char* ptrlinha[], int nlinhas)
{
    while (nlinhas-- > 0) {
       printf("%s\n", *ptrlinha++);
```

```
    }
}
```

Inicialmente, **ptrlinha* aponta para a primeira linha: cada incremento avança para o próximo ponteiro de linha enquanto *nlinhas* é decrementado.

Com a entrada e saída sob controle, podemos prosseguir para a ordenação. A ordenação rápida do Capítulo 4 precisa de pequenas mudanças: as declarações precisam ser modificadas, e a operação de comparação deve ser feita chamando-se *strcmp*. O algoritmo permanece igual, o que nos dá alguma confiança de que funcionará corretamente.

Programa 5.15: Função qsort().

```
/* qsort: ordena v[esq]...v[dir] em ordem crescente */
void qsort (char* v[], int esq, int dir)
{
    int ultimo;
    void troca(char* v[], int i, int j);

    if (esq >= dir) { // não faz nada se vetor contém
        return;       // menos de dois elementos
    }

    troca (v, esq, (esq + dir) / 2);
    ultimo = esq;

    for (int i = esq + 1; i <= dir; i++) {
        if (strcmp (v[i], v[esq]) < 0) {
            troca (v, ++ultimo, i);
        }
    }

    troca (v, esq, ultimo);
    qsort (v, esq, ultimo - 1);
    qsort (v, ultimo + 1, dir);
}
```

Da mesma forma, a rotina troca só precisa de mudanças triviais:

Programa 5.16: Função troca().

```
/* troca: substitui v[i] por v[j] */
void troca (char* v[], int i, int j)
{
    char* temp;
    temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}
```

Como qualquer elemento individual de *v* (nosso *ptrlinha*) é um ponteiro de caractere, **temp* também o deve ser, de forma que um possa ser copiado no outro.

**Exercício 5.7** Reescreva *lelinhas* de forma que armazene linhas em um vetor fornecido por *main*, ao invés de chamar *aloca*. Quão mais rápido é o programa? ■

## 5.7 Vetores Multidimensionais

**C** fornece vetores multidimensionais retangulares, embora na prática eles sejam muito menos usados do que os vetores de ponteiros. Nesta seção, mostraremos algumas de suas propriedades características.

Considere o problema de conversão de data, de dia do mês para dia do ano e vice-versa. Por exemplo, **primeiro de março** é o sexagésimo dia de um ano não-bissexto, e o sexagésimo-primeiro dia de um ano bissexto. Vamos definir duas funções para fazer as conversões: *dia_do_ano* converte o mês e dia para o dia do ano, e *dia_do_mes* converte o dia do ano para mês e dia. Como esta última função calcula dois valores, os argumentos do mês e dia serão ponteiros:

```
dia_do_mes(1988, 60, &m, &d)
```

define *m* para 2 e *d* para 29 (29 de fevereiro).

Estas funções precisam da mesma informação, uma tabela de número de dias em cada mês ( "trinta dias tem setembro, ... " ). Como o número de dias por mês é diferente para anos bissextos, é mais fácil separá-los em duas linhas de um vetor bidimensional do que cuidar do que acontece com fevereiro durante o cálculo. O vetor e as funções para executar as transformações são os seguintes:

Programa 5.17: Funções para calcular o dia do ano.

```
static char tabdia[2][13] = {
    {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
    {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}
};

/* dia_do_ano: define o dia do ano a partir de mês & dia */
int dia_do_ano (int ano, int mes, int dia)
{
    int i, bissexto;
    bissexto = ano % 4 == 0 && ano % 100 != 0 || ano % 400 == 0;

    for (i = 1; i < mes; i++) {
        dia += tabdia[bissexto][i];
    }

    return dia;
}

/* dia_do_mes: define mês, dia a partir do dia do ano */
void dia_do_mes (int ano, int diaano, int* ames, int* adia)
{
    int i, bissexto;
    bissexto = ano % 4 == 0 && ano % 100 != 0 || ano % 400 == 0;

    for (i = 1; diaano > tabdia[bissexto][i]; i++) {
        diaano -= tabdia[bissexto][i];
    }

    *ames = i;
    *adia = diaano;
}
```

Lembre-se de que o valor aritmético de uma expressão lógica, como aquela para o cálculo de bissexto, é **zero** (falso) ou **um** (verdadeiro), de modo que pode ser usado como subscrito do vetor *tabdia*.

O vetor *tabdia* deve ser externo tanto a *dia_do_ano* quanto a *dia_do_mes*, de forma que ambos possam usá-lo. Nós o tornamos *char* para ilustrar um uso legítimo de *char* para armazenar pequenos inteiros não-caracteres.

*tabdia* é o primeiro vetor bidimensional com que lidamos. Em **C**, um vetor bidimensional é na realidade um vetor unidimensional, sendo que cada um dos elementos é um vetor. Portanto, os subscritos são escritos com

```
tabdia[i][j] // [linha][coluna]
```

ao invés de

```
tabdia[i,i] // ERRADO
```

Além desta distinção de notações, um vetor bidimensional pode ser tratado de forma semelhante a outras linguagens. Elementos são armazenados em linhas, de modo que o subscrito da direita, ou coluna, varia mais rapidamente à medida que elementos são acessados na ordem do armazenamento.

Um vetor é inicializado por uma lista de inicializadores entre chaves: cada linha de um vetor bidimensional é inicializada por uma sublista correspondente. Iniciamos o vetor *tabdia* com uma coluna de zero para que os números de mês possam trabalhar naturalmente de 1 a 12 ao invés de 0 a 11 . Como o espaço não é escasso neste exemplo, esta forma é mais clara do que ajustar os índices.

Se um vetor bidimensional tiver que ser passado para uma função, a declaração de parâmetro na função deve incluir o número de colunas; o número de linhas é irrelevante, pois o que é passado é, como antes, um ponteiro para um vetor de linhas, onde cada linha é um vetor de 13 inteiros. Neste caso particular, ele é um ponteiro para objetos que são vetores de 13 inteiros. Assim, se o vetor *tabdia* tivesse que ser passado para uma função $f$ a declaração de $f$ seria

```
f (int tabdia[2][13]) {...}
```

Também poderia ser

```
f (int tabdia[][13]) {...}
```

pois o número de linhas é irrelevante, ou então

```
f (int (*tabdia)[13]) {...}
```

que diz que o parâmetro é um ponteiro para um vetor de 13 inteiros. Os parênteses são necessários porque os colchetes [] têm maior precedência do que $*$. Sem os parênteses, a declaração

```
int *tabdia[13]
```

seria um vetor de 13 ponteiros para inteiros. Generalizando, somente a primeira dimensão (subscrito) de um vetor é opcional; todas as outras precisam ser especificadas.

A Seção 5.12 explica melhor algumas declarações mais complicadas.

**Exercício 5.8** Não existe uma checagem de erro nas funções *dia_do_ano* e *dia_do_mes*. Conserte este defeito. ■

## 5.8 Inicialização de Vetores de Ponteiros

Considere o problema de escrever uma função *nome_mes* que retorne um ponteiro para uma string de caracteres contendo o nome do *n-ésimo* mês. Esta é uma aplicação ideal para um vetor *static* interno. *nome_mes* contém um vetor privado de cadeias de caracteres, e retorna um ponteiro para o nome apropriado quando chamado. Esta seção mostra como esse vetor de nomes é inicializado pela função.

A sintaxe é semelhante às inicializações anteriores:

Programa 5.18: Função nome_mes().

```
/* nome_mes: retorna nome do n-ésino mês */
char* nome_mes(int n)
{
    static char* nome[] = {
        "Mês ilegal",
        "janeiro", "fevereiro", "março",
        "abril", "maio", "junho",
        "julho", "agosto", "setembro",
        "outubro", "novembro", "dezembro"
    };
    return (n < 1 || n > 12) ? nome[0] : nome[n];
}
```

A declaração de nome, que é um vetor de ponteiros de caracteres, é o mesmo que *ptrlinha* no exemplo de ordenação. O inicializador é uma lista de cadeias de caracteres; cada uma é atribuída à posição correspondente no vetor. Os caracteres da *i-ésima* string são colocados em algum lugar, e um ponteiro para eles é armazenado em $nome[i]$. Como o tamanho do vetor nome não é especificado, o compilador conta os inicializadores e preenche o número correto.

## 5.9 Ponteiros Versus Vetores Multidimensionais

Os iniciantes em **C** ficam às vezes confusos com a diferença entre um vetor bidimensional e um vetor de ponteiros, como nome no exemplo anterior. Dadas as definições

```
int a[10][20];
int *b[10];
```

então $a[3][4]$ e $b[3][4]$ são sintaticamente referências legais a um único inteiro. Mas *a* é um verdadeiro vetor bidimensional: 200 locais do tamanho de um inteiro foram separados, e o cálculo convencional de subscrito retangular *20 * linha + col* é usado para encontrar o elemento *a[linha][col]*. Para *b*, no entanto, a definição somente aloca 10 ponteiros e não os inicializa; a inicialização deve ser feita explicitamente, isto é, estaticamente ou codificada. Supondo que cada elemento de *b* aponte para um vetor de **vinte elementos**, então haverá 200 inteiros separados, e mais **dez** células para os ponteiros. A importante vantagem do vetor de ponteiros é que as linhas do vetor podem ser de tamanhos diferentes, isto é, cada elemento de *b* não precisa apontar para um vetor de vinte elementos; alguns podem apontar para dois elementos, alguns para cinquenta, e alguns para nenhum sequer.

Embora tenhamos discutido este assunto em termos de inteiros, o uso mais frequente de vetores de ponteiros é armazenar cadeias de caractere de diversos tamanhos, como na função *nome_mes*. Compare a declaração e a figura de um vetor de ponteiros (Figura 5.9):

```
char *nome[] = { "Mês ilegal", "jan", "fev", "mar" };
```

com os de um vetor bidimensional:

```
char vnome[][15] = { "Mês ilegal", "jan", "fev", "mar" };
```

**Exercício 5.9** Reescreva as rotinas *dia_do_ano* e *dia_do_mes* com ponteiros no lugar da indexação. ■

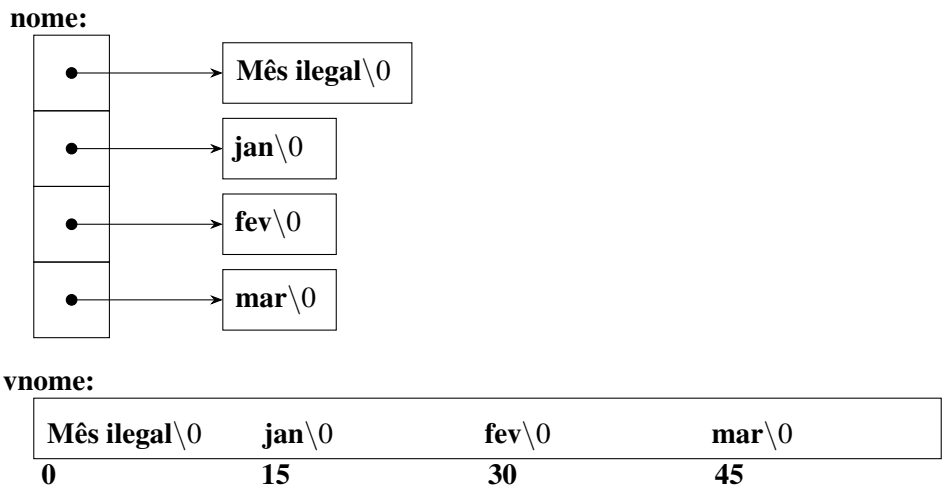



Figura 5.9: Vetor de ponteiros e vetor multidimensional.

## 5.10 Argumentos da Linha de Comando

Em ambientes que suportam **C**, há uma forma de passar argumentos ou parâmetros da linha de comando para um programa quando ele começa a ser executado. Quando *main* é chamado, ele recebe dois argumentos. O primeiro (por convenção chamado *argc*, de contador de argumentos) é o número de argumentos da linha de comando com que o programa foi chamado; o segundo (*argv*, de vetor de argumentos) é um ponteiro para um vetor de cadeias de caracteres que contêm os argumentos, um por string. Normalmente usamos múltiplos níveis de ponteiros para lidar com estas cadeias de caracteres.

A ilustração mais simples é o programa *eco*, que ecoa seus argumentos da linha de comando em uma única linha, separados por espaços em branco, isto é, o comando

```
eco primeiro programa
```

imprime a saída

```
primeiro programa
```

Por convenção, *argv[0]* é o nome pelo qual o programa foi chamado, de modo que *argc* é pelo menos 1. Se *argc* é 1, não existem argumentos na linha de comando após o nome do programa. No exemplo acima, *argc* é 3, e *argv[0]*, *argv[1]* e *argv[2]* são "eco", "primeiro" e "programa", respectivamente. O primeiro argumento opcional é *argv[1]* e o último *argv[argc-1]*; adicionalmente, o padrão requer que *argv[argc]* seja um ponteiro nulo.

A primeira versão de eco trata *argv* como um vetor de ponteiros do tipo caractere:

Programa 5.19: Função eco(), primeira versão.

```
#include <stdio.h>

/* ecoa argumentos da linha de comando; versão 1 */
int main(int argc, char* argv[])
{
    for (int i = 1; i < argc; i++) {
        printf("%s %s", argv[i], (i < argc - 1) ? " " : "");
    }
```

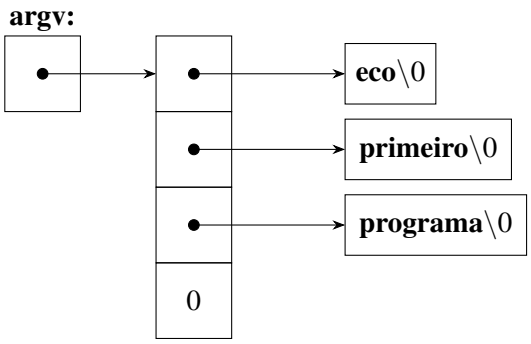



Figura 5.10: argv.

```
    printf ("\n");
    return 0;
}
```

Como *argv* é um ponteiro para um vetor de ponteiros, podemos manipular o ponteiro no lugar do índice do vetor. Esta próxima variação é baseada no incremento de *argv*, que é um ponteiro para um ponteiro para *char*, enquanto *argc* é decrementado:

Programa 5.20: Função eco(), segunda versão.

```
#include <stdio.h>

/* ecoa argumentos da linha de comando; versão 2 */
int main(int argc, char* argv[])
{
    while (--argc > 0) {
        printf ("%s%s", *++argv, (argc > 1) ? " " : "");
    }

    printf ("\n");
    return 0;
}
```

Como *argv* é um ponteiro para o início do vetor de cadeias do argumento, incrementá-lo de 1 ($++argv$) faz com que aponte para o $argv[1]$ original ao invés de $argv[0]$. Cada incremento sucessivo o move para o próximo argumento; **argv* é então um ponteiro para esse argumento. Ao mesmo tempo, *argc* é decrementado; quando ele se torna 0, não há mais argumentos para imprimir.

Como alternativa, poderíamos escrever o comando *printf* da seguinte forma:

```
printf((argc > 1) ? "%s " : "%s", *++argv);
```

Isto mostra que o argumento de formato de *printf* pode ser também uma expressão.

Como um segundo exemplo, vamos fazer algumas melhorias no programa de pesquisa de padrão da Seção 4.1. Se você se lembra, nós embutimos o padrão de pesquisa dentro do programa, uma solução obviamente não satisfatória. Apoiando-se na sintaxe do utilitário *grep* do **UNIX**, vamos alterar o programa para que o padrão a ser pesquisado seja especificado pelo primeiro argumento da linha de comando.

Programa 5.21: Programa de busca de padrão aprimorado.

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

```
#include <string.h>
#define MAXLIN 1000

int lelinha(char* linha, int max);

/* acha: imprime linhas combinando com padrão do primeiro arg.*/
int main(int argc, char* argv[])
{
    char linha[MAXLIN];
    int achadas = 0;

    if (argc != 2) {
        printf ("Uso: acha padrão\n");
    } else {
        while (lelinha(linha, MAXLIN) > 0)
            if (strstr(linha, argv[1]) != NULL) {
                printf ("%s", linha);
                achadas++;
            }

        return achadas;
    }
}
```

A função da **biblioteca-padrão** *strstr(a,t)* retorna um ponteiro para a primeira ocorrência da string *t* na string *s*, ou *NULL* se não houver. Ela é declarada em <*string.h* >.

O modelo pode agora ser elaborado para ilustrar outras construções de ponteiro. Suponha que queiramos permitir dois argumentos opcionais. Um diz "*imprima todas as linhas exceto aquelas que casam com o padrão*;" a segunda diz "*preceda cada linha impressa pelo seu número*".

Uma convenção comum para os programas em **C** nos sistemas **UNIX** é a de que um argumento começando com um sinal de menos introduz um sinalizador ou parâmetro opcionais. Se escolhemos $-x$ ("exceto") para sinalizar a inversão, e $-n$ ("numerar") para requisitar a numeração de linha, então o comando

```
acha -x -n padrão
```

imprimirá cada linha que não combine com o padrão, precedida pelo seu número de linha.

Argumentos opcionais devem ser permitidos em qualquer ordem, e o restante do programa deve ser independente do número de argumentos que estiverem presentes. Além do mais, é conveniente para os usuários se os argumentos das opções pudessem ser combinados, como em

```
acha -nx padrão
```

Aqui está o programa:

Programa 5.22: Programa de busca de padrão com parâmeros -x e - n.

```
#include <stdio.h>
#include <string.h>
#define MAXLIN 1000

int lelinha (char* linha, int max);

/* acha: imprime linhas combinando com padrão do primeiro arg. */
main(int argc, char* argv[])
{
    char linha[MAXLIN];
```

```
    long nrlinha = 0;
    int c, exceto = 0, numero = 0, achadas = 0;

    while (--argc > 0 && (*++argv)[0] == '-')
        while (c = *++argv[0])
            switch (c) {
            case 'x':
                exceto = 1;
                break;

            case 'n':
                numero = 1;
                break;

            default:
                printf("acha: opção ilegal %c\n", c);
                argc = 0;
                achadas = -1;
                break;
            }

    if (argc != 1) {
        printf("Uso: acha -x -n padrão\n");
    } else {
        while (lelinha(linha, MAXLIN) > 0) {
            nrlinha++;

            if ((strstr(linha, *argv) != NULL) != exceto) {
                if (numero) {
                    printf("%ld:", nrlinha);
                }

                printf("%s", linha);
                achadas++;
            }
        }
    }

    return achadas;
}
```

*argc* é decrementado e *argv* é incrementado antes de cada argumento opcional. Ao final do laço, se não houver erros, *argc* dirá quantos argumentos restam não processados e *argv* apontará para o primeiro deles. Assim, *argc* deve ser 1 e **argv* deve apontar para o padrão. Observe que `*++argv` é um ponteiro para uma string do argumento, de forma que `(*++ argv)[0]` é seu primeiro caractere. (Uma alternativa válida seria `**++argv`.) Como `[]` liga mais forte do que $*$ e $++$, os parênteses são necessários; sem eles, a expressão seria tomada como `*++(argv[0])`. De fato, foi isso o que usamos no laço interior, onde a tarefa foi passar pela string do argumento específico. No laço interior, a expressão `*++argv[0]` incrementa o ponteiro `argv[0]`!

Raramente alguém usará expressões com ponteiros mais difíceis do que estas; nesses casos, dividi-los em dois ou três passos será mais intuitivo.

**Exercício 5.10** Escreva o programa *expr*, que avalia uma expressão em notação polonesa reversa na linha de comando, onde cada operador ou operando é um argumento separado. Por

exemplo,

```
expr 3 4 + *
```

ava1ia 2 x (3 + 4). ■

**Exercício 5.11** Modifique os programas *tab* e *destab* (escritos como exercícios no **Capítulo 1**) para aceitarem uma lista de pontos de tabulação como argumentos. Use os pontos normais de tabulação, se não forem dados argumentos. ■

**Exercício 5.12** Estenda *tab* e *destab* para aceitarem a abreviação

```
tab -m +n
```

que define pontos de tabulação a cada *n* colunas a partir da coluna *m*. Escolha um comportamento conveniente (para o usuário) no caso de omissão dos argumentos. ■

**Exercício 5.13** Escreva o programa *cauda* que imprima as últimas *n* linhas de sua entrada. Caso não especificado, *n* assume valor 10, digamos, mas ele pode ser alterado por um argumento opcional de forma que

```
cauda -n
```

imprima as últimas *n-linhas*. O programa deve se comportar de forma racional independente da entrada e do valor de *n*. Escreva o programa de modo que ele faça o melhor uso da área de armazenamento disponível; linhas devem ser armazenadas como na função de ordenação da Seção 5.6, e não em um vetor bidimensional de tamanho fixo. ■

## 5.11 Ponteiros para Funções

Em **C**, uma função não é uma variável, mas é possível definir um ponteiro para uma função, o qual pode ser atribuído, colocado em vetores, passado para funções, retornado de funções, e assim por diante. Ilustraremos isso modificando o programa de ordenação escrito anteriormente neste capítulo para que, na presença de um argumento opcional $-n$, ele ordene as linhas de entrada numericamente e não lexicograficamente.

Um ordenador frequentemente consiste em três partes – **uma comparação** que determina a ordenação de qualquer par de objetos, **uma troca** que inverte sua ordem, e **um algoritmo de ordenação** que faz as comparações e trocas até que os objetos estejam na ordem. O algoritmo de ordenação é independente das operações de comparação e troca, de modo que, passando diferentes funções de comparação e troca a ele, podemos obter a ordenação por diferentes critérios. Este é o método tomado no novo ordenador.

A comparação lexicográfica de duas linhas é feita por meio de *strcmp*, como antes; também usaremos uma rotina *numcmp* que compare duas linhas com base no valor numérico e retorne o mesmo tipo de indicação que *strcmp* retorna. Estas funções são declaradas em *main* e um ponteiro para elas é passado para *quicksort*. Não incluímos o processamento de erros para os argumentos, para nos concentrarmos nas ideias principais.

Programa 5.23: Programa de ordenação de linhas de entrada.

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

```
#define MAXLIN 5000     // máximo de linhas para ordenar
char* ptrlinha[MAXLIN]; // ponteiros para próximas linhas

int lelinhas(char* ptrlinha[], int nlinhas);
void imprlinhas(char* ptrlinha[], int nlinhas);

void quicksort(void* ptrlinha[], int esq, int dir,
               int(*comp)(void*, void*));
int numcmp(char*, char*);

/* ordena linhas de entrada */
int main(int argc, char* argv[])
{
    int nlinhas;      // número de linhas lidas
    int numerico = 0; // 1 se ordenação numérica

    if (argc > 1 && strcmp(argv[1], "-n") == 0) {
        numerico = 1;
    }

    if ((nlinhas = lelinhas(ptrlinha, MAXLIN)) >= 0) {
        quicksort((void*) ptrlinha, 0, nlinhas - 1,
                  (int (*) (void*, void*)) (numerico ? numcmp : strcmp));
        imprlinhas(ptrlinha, nlinhas);
        return 0;
    } else {
        printf("Entrada muito longa\n");
        return 1;
    }
}
```

Na chamada a *quicksort*, *strcmp* e *numcmp* são endereços de funções. Como se sabe que são funções, o operador & não é necessário, da mesma forma como não é necessário antes de um nome de vetor.

Escrevemos *quicksort* de forma que possa processar qualquer tipo de dado, não apenas cadeias de caracteres. Conforme indicado pelo protótipo da função, *quicksort* espera um vetor de ponteiros, dois inteiros e uma função com dois argumentos do tipo ponteiro. O tipo ponteiro genérico *void∗* é usado para os argumentos do ponteiro. Qualquer ponteiro pode ser moldado para *void∗* e trazido de volta sem qualquer perda de informação, de forma que podemos chamar *quicksort* moldando argumentos para *void∗*. O molde elaborado do argumento da função molda os argumentos da função de comparação. Estes geralmente não terão efeito sobre a representação real, mas asseguram ao compilador de que tudo está bem.

Programa 5.24: Função quicksort().

```
/* quicksort: ordena v[esq]...v[dir] em ordem crscente */
void quicksort(void* v[], int esq, int dir,
               int (*comp)(void*, void*))
{
    int i, ultimo;
    void troca(void* v[], int, int);

    if (esq >= dir) {  // não faz nada se o vetor contém
        return;        // menos de 2 elementos
    }
```

```
    troca(v, esq, (esq + dir) / 2);
    ultimo = esq;

    for (i = esq + 1; i <= dir;  i++) {
        if ((*comp)(v[i], v[esq]) < 0) {
            troca(v, ++ultimo, i);
        }
    }

    troca(v, esq, ultimo);
    quicksort(v, esq, ultimo - 1, comp);
    quicksort(v, ultimo + 1, dir, comp);
}
```

As declarações devem ser estudadas com certo cuidado. O quarto parâmetro de *quicksort* é

```
int (*comp)(void *, void *)
```

que diz que *comp* é um ponteiro para uma função que tem dois argumentos *void∗* e retorna um inteiro.

O uso de *comp* na linha

```
if ((*comp)(v[i], v[esq]) < 0)
```

É coerente com a declaração; *comp* é um ponteiro para uma função, **comp* é a função, e

```
(*comp)(v[i], v[esq])
```

é a chamada a esta função. Os parênteses são necessários para que os componentes sejam corretamente associados; sem eles,

```
int *comp (void *, void *) /* ERRADO */
```

diz que *comp* é uma função retornando um ponteiro para um inteiro, o que é muito diferente.

Já mostramos *strcmp*, que compara duas cadeias. Aqui está *numcmp*, que compara duas cadeias baseando-se nos valores numéricos iniciais, calculados pela chamada a *atof*:

Programa 5.25: Função numcmp().

```
#include <stdlib.h>

/* numcmp:  compara s1 e s2 numericamente */
int numcmp(char* s1, char* s2)
{
    double v1, v2;
    v1 = atof(s1);
    v2 = atof(s2);

    if (v1 < v2) {
        return -1;
    } else if (v1 > v2) {
        return 1;
    } else {
        return 0;
    }
}
```

A função *troca*, que troca de lugar dois ponteiros, é idêntica à que apresentamos anteriormente neste capítulo, exceto que as declarações são alteradas para *void*∗.

Programa 5.26: Função troca() genérica.

```
void troca(void* v[], int i, int j)
{
    void* temp;
    temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}
```

Uma série de outras opções pode ser adicionada ao programa de ordenação: algumas compõem exercícios desafiadores.

**Exercício 5.14** Modifique o programa de ordenação para que utilize um sinalizador $-r$, que indica ordenação em ordem **reversa** (decrescente). Assegure-se que $-r$ funciona com $-n$. ■

**Exercício 5.15** Inclua a opção $-f$ para comparar maiúsculas e minúsculas juntamente, de forma que as distinções das mesmas letras não sejam feitas durante a ordenação: por exemplo, *a* e *A* devem ser comparados igualmente. ■

**Exercício 5.16** Inclua a opção $-d$ ("ordem do diretório"), que faz comparações somente em letras, números e espaços em branco. Assegure-se de que funciona em conjunto com $-f$. ■

**Exercício 5.17** Inclua uma capacidade de busca de campo, de modo que a ordenação possa ser feita em campos dentro das linhas, com cada campo ordenado segundo um conjunto independente de opções. (O índice do original deste livro foi ordenado com $-df$ para a categoria do índice e $-n$ para os números de página. ■

## 5.12 Declarações Complicadas

**C** às vezes é castigado pela sintaxe de suas declarações, particularmente aquelas que envolvem ponteiros para funções. A sintaxe é uma tentativa de fazer com que a declaração e o uso se combinem: ela funciona bem para casos simples, mas pode ser confusa para aqueles mais difíceis, pois as declarações não podem ser lidas da esquerda para a direita, e porque os parênteses são muito usados. A diferença entre

```
int *f(); // f:  função retornando ponteiro para int
```

e

```
int (*af)(); // af:  ponteiro para função retornando int
```

ilustra o problema: ∗ é um operador de prefixo e tem menor precedência que (), de forma que os parênteses são necessários para forçarem a associação correta.

Embora declarações verdadeiramente complicadas raramente surjam na prática, é importante saber como entendê-las, e, se necessário, como criá-las. Uma boa forma de sintetizar as declarações é em pequenos passos com *typedef*, que é discutido na Seção 6.7. Como alternativa, nesta seção apresentaremos um par de programas que convertem de **C** válido para uma descrição por palavras e novamente de volta. A descrição por palavras é lida da esquerda para a direita.

O primeiro, *dcl*, é o mais complexo. Ele converte uma declaração em **C** numa descrição por palavras, como nos exemplos a seguir:

```
char **argv
     argv:  ponteiro para ponteiro para char
int (*tabdia)[13]
     tabdia:  ponteiro para vetor[13] de int
int *tabdia[13]
     tabdia:  vetor[13] de ponteiro para int
void *comp()
     comp: função retornando ponteiro para void
void (*comp)()
     comp: ponteiro para função retornando void
char (*(*x())[])()
      x: função retornando ponteiro para vetor[] de
      ponteiro para função retornando char
char (*(*x[3])())[5]
      x: vetor[3] de ponteiro para função retornando
      ponteiro para vetor[5] de char
```

*dcl* é baseado na gramática que especifica um declarador, que é precisamente expressa no Apêndice A, Seção A.8.5; esta é uma forma simplificada:

```
dcl:       *'s opcionais dcl-direta
dcl-direta:  nome
                  (dcl)
                  dcl-direta()
                  dcl-direta[tamanho opcional]
```

Em palavras, *dcl* é uma *dcl-direta*, talvez precedida por $*$'s. Uma *dcl-direta* é um nome, ou uma *dcl* entre parênteses, ou uma *dcl-direta* seguida por parênteses, ou uma *dcl-direta* seguida por colchetes com um tamanho opcional.

Esta gramática pode ser usada para analisar declarações. Por exemplo, considere este declarador:

```
(*pfa[])()
```

*pfa* será definido como um nome e, portanto, como uma *dcl-direta*. Então *pfa[]* é também uma *dcl-direta*. Então **pfa[]* é uma forma reconhecida de *dcl*, de forma que *(*pfa[])* é uma *dcl-direta*. Então *(*pfa[])()* é uma *dcl-direta* e, portanto, uma *dcl*. Podemos também ilustrar esta análise com uma árvore como a que apresentamos na figura seguinte (onde *dcl-direta* foi abreviada para *dcl-dir*).

O núcleo do programa *dcl* é um par de funções, *dcl* e *dcldir*, que analisa uma declaração segundo esta gramática. Como a gramática é definida recursivamente, as funções chamam umas às outras recursivamente, à medida que reconhecem partes de uma declaração: o programa é chamado analisador recursivo de descendência.

Programa 5.27: Programa analisador recursivo de descêndencia.

```c
/* dcl: analisa um declarador */
void dcl(void)
{
    int ns;

    for (ns = 0; pegacodigo() == '*'; ) { // conta *
        ns++;
```

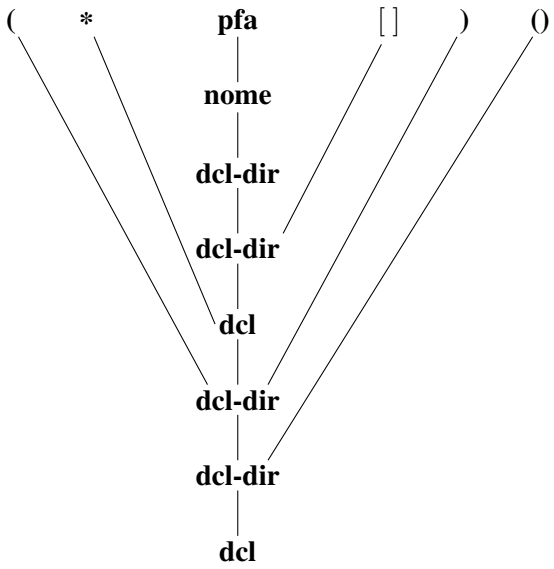



Figura 5.11: Análise de declaração.

```
    }

    dcldir ();

    while (ns-- > 0) {
        strcat (saida, " ponteiro para");
    }
}

/* dcldir: analisa um declarador direto */
void dcldir(void)
{
    int tipo;

    if (tipocodigo == '(') { // (dcl)
        dcl ();

        if (tipocodigo != ')') {
            printf ("Erro: falta )\n");
        }
    } else if (tipocodigo == NOME) { // nome de variável
        strcpy (nome, codigo);
    } else {
        printf ("Erro: esperado nome ou (dcl)\n");
    }

    while ((tipo = pegacodigo()) == PARENS || tipo == COLCHETES) {
        if (tipo == PARENS) {
            strcat (saida, " função retornando");
        } else {
            strcat(saida, " vetor");
            strcat(saida, codigo);
```

```
            strcat(saida, " de");
        }
    }
}
```

Como os programas pretendem ser ilustrativos, e não à prova de falhas, há restrições significativas sobre *dcl*. Ele só pode usar um tipo de dados simples como *char* ou *int*. Ele não lida com tipos de argumento em funções, ou qualificadores como *const*. Espaços embutidos o confundem. Ele não realiza muita recuperação de erro, de forma que declarações inválidas também o confundirão. Estas melhorias são deixadas como exercícios.

Aqui estão as variáveis globais e a rotina principal:

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

#define MAXCODIGO  100

enum { NOME, PARENS, COLCHETES };

void dcl(void);
void dcldir(void);

int pegacodigo(void);
int tipocodigo;           // tipo do último código
char codigo[MAXCODIGO];   // string do último código
char nome[MAXCODIGO];     // nome do identificador
char tipodado[MAXCODIGO]; // tipo dado = char, int, etc.
char saida[1000];

int main()  /* converte declaração em palavras */
{
    while (pegacodigo() != EOF) { // primeiro código na linha
        strcpy(tipodado, codigo); // é o tipo dado
        saida[0] = '\0';
        dcl();       // analisa restante da linha

        if (tipocodigo != '\n') {
            printf("erro de sintaxe\n");
        }

        printf("%s: %s %s\n", nome, saida, tipodado);
    }

    return 0;
}caption
```

A função *pegacodigo* salta espaços em branco e marcas de tabulação, e depois encontra o próximo código na entrada; um "código" é um nome, um par de parênteses, um par de colchetes talvez incluindo um número, ou qualquer outro caractere isolado.

Programa 5.28: Função pegacodigo().

```
int pegacodigo(void)  /* retorna próximo código */
{
    int c, getch(void);
    void ungetch(int);
```

```
    char* p = codigo;

    while ((c = getch()) == ' ' || c == '\t')
        ;

    if (c == '(') {
        if ((c = getch()) == ')') {
            strcpy(codigo, "()");
            return tipocodigo = PARENS;
        } else {
            ungetch(c);
            return tipocodigo = '(';
        }
    } else if (c == '[') {
        for (*p++ = c; (*p++ = getch()) != ']'; )
            ;

        *p = '\0';
        return tipocodigo = COLCHETES;
    } else if (isalpha(c)) {
        for (*p++ = c; isalnum(c = getch()); ) {
            *p++ = c;
        }

        *p = '\0';
        ungetch(c);
        return tipocodigo = NOME;
    } else {
        return tipocodigo = c;
    }
}
```

*getch* e *ungetch* foram discutidos no Capítulo 4.

Caminhar na outra direção é mais fácil, especialmente se não nos importamos em gerar parênteses redundantes. O programa *indcl* converte uma descrição com palavras do tipo "*x é uma função retornando um ponteiro para um vetor de ponteiros para funções retornando char*", que será expressa por

```
x () * [] * () char
```

para

```
char (*(*x())[])()
```

A sintaxe abreviada na entrada nos permite reutilizar a função *pegacodigo*, *indcl* também usa as mesmas variáveis externas que *dcl* usa.

Programa 5.29: Programa indcl.

```
/* indcl: converte descrição de palavras em declaração */
int main()
{
    int tipo;
    char temp[MAXCODIGO];

    while (pegacodigo() != EOF) {
        strcpy(saida, codigo);
```

```
        while ((tipo = pegacodigo()) != '\n') {
            if (tipo == PARENS || tipo == COLCHETES) {
                strcat(saida, codigo);
            } else if (tipo == '*') {
                sprintf(temp, "(*%s)", saida);
                strcpy(saida, temp);
            } else if (tipo == NOME) {
                sprintf(temp, "%s %s", codigo, saida);
                strcpy(saida, temp);
            } else {
                printf("Entrada inválida em %s\n", codigo);
            }
        }
    }

    return 0;
}
```

**Exercício 5.18** Faça com que *dcl* se recupere de erros na entrada. ■

**Exercício 5.19** Modifique *indcl* de forma a não incluir parênteses redundantes nas declarações. ■

**Exercício 5.20** Expanda *dcl* para que manipule declarações com tipos de argumentos de função, qualificadores como *const*, e assim por diante. ■

# 6. Estruturas

Uma estrutura é uma coleção de uma ou mais variáveis, possivelmente de tipos diferentes, colocadas juntas sob um único nome para manipulação conveniente. (Estruturas são chamadas "registros" em algumas linguagens, mais notadamente Pascal.) As estruturas ajudam a organizar dados complicados, particularmente em grandes programas, pois permitem que um grupo de variáveis relacionadas sejam tratadas como uma unidade ao invés de entidades separadas.

Um exemplo tradicional de estrutura é o registro de uma folha de pagamento: um empregado é descrito por um conjunto de atributos como nome, endereço, número do CPF, salário, etc. Alguns deles, por sua vez, poderiam ser estruturas: um nome tem vários componentes, assim como um endereço e até mesmo um salário. Um outro exemplo, mais típico para C, origina-se dos gráficos: um ponto é um par de coordenadas, um retângulo é um par de pontos, e assim por diante.

A principal mudança feita pelo padrão **ANSI** é definir a atribuição de estrutura – estruturas podem ser copiadas e atribuídas, passadas para funções, e retornadas de funções. Isso tem sido suportado por muitos compiladores há anos, mas as propriedades foram agora definidas precisamente. Agora as estruturas e vetores automáticos também podem ser inicializados.

## 6.1 Elementos Básicos

Vamos criar algumas poucas estruturas apropriadas para gráficos. O objeto básico é um ponto, que assumiremos ter uma coordenada $x$ e uma coordenada $y$, ambas inteiras.

Os dois componentes podem ser colocados em uma estrutura declarada da seguinte forma:

```
struct ponto {
    int x;
    int y;
};
```

A palavra-chave *struct* introduz uma declaração de estrutura, que é uma lista de declarações entre chaves. Um nome opcional chamado etiqueta de estrutura pode seguir a palavra *struct* (como a palavra *ponto* aqui). A etiqueta atribui um nome a este tipo de estrutura, e pode ser usada posteriormente como uma abreviação para a descrição detalhada entre chaves.

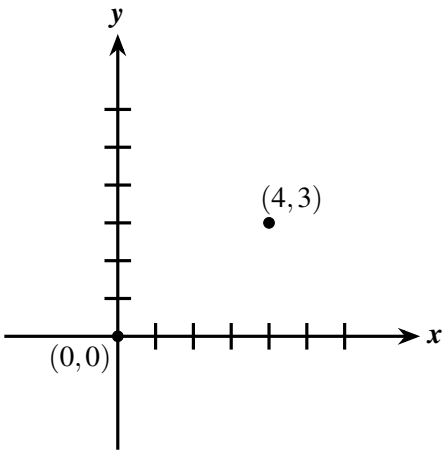



Figura 6.1: Coordenadas cartesianas.

As variáveis nomeadas em uma estrutura são chamadas *membros*. Um membro ou etiqueta de uma estrutura e uma variável comum (isto é, não-membro) podem ter o mesmo nome sem haver conflito, pois sempre poderão ser distinguidas pelo contexto. Além do mais, os mesmos nomes de membro podem ocorrer em diferentes estruturas, embora por questão de estilo, normalmente se usam os mesmos nomes apenas para objetos intimamente relacionados.

Uma declaração *struct* define um tipo. O **fecha-chave** que termina a lista de membros pode ser seguido por uma lista de variáveis, assim como para qualquer tipo básico. Isto é,

```
struct { ... } x, y, z;
```

é sintaticamente análogo a

```
int x, y, z;
```

no sentido de que cada comando declara *x*, *y* e *z* como variáveis do tipo nomeado e faz com que seja separada memória para elas.

Uma declaração de *struct* que não é seguida por uma lista de variáveis não reserva qualquer memória; ela simplesmente descreve um gabarito ou formato de uma estrutura. Se a declaração for etiquetada, entretanto, a etiqueta pode ser usada mais adiante em definições das ocorrências da estrutura. Por exemplo, dada a declaração de *ponto* acima,

```
struct ponto pt;
```

define uma variável *pt* que é uma estrutura do tipo *struct ponto*. Uma estrutura pode ser inicializada seguindo-se sua definição por uma lista de inicializadores, cada um sendo uma expressão constante, para os membros:

```
struct ponto ptmax = { 320, 260 };
```

Uma estrutura automática também pode ser inicializada por atribuição ou pela chamada de uma função que retorne uma estrutura do tipo correto.

Um membro de uma estrutura em particular é referenciado em uma expressão por meio de uma construção do tipo

*nome-estrutura.membro*

O operador de membro de estrutura "." conecta o nome da estrutura e o nome do membro. Para imprimir as coordenadas do ponto *pt*, por exemplo.

```
printf("%d,%d", pt.x, pt.y);
```

ou então, para calcular a distância da origem $(0,0)$ a *pt*, segundo o teorema de Pitágoras.

```
double dist, sqrt(double);

dist = sqrt((double)pt.x * pt.x + (double)pt.y * pt.y);
```

As estruturas podem ser encaixadas. Uma representação de um retângulo é um par de pontos que indica os cantos diagonalmente opostos:

```
struct retang {
    struct ponto pt1;
    struct ponto pt2;
};
```

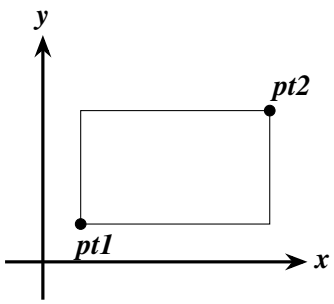



Figura 6.2: Retângulo.

A estrutura *retang* contém duas estruturas *ponto*. Se declararmos tela como

```
struct retang tela;
```

então

```
tela.pt1.x
```

refere-se à coordenada *x* do membro *pt1* de tela.

## 6.2 Estruturas e Funções

As únicas operações legais em uma estrutura são copiá-la e atribuí-la como uma unidade, tomando-se seu endereço com &, e acessando-se seus membros. Cópia e atribuição incluem a passagem de argumentos para função e também o retorno de valores de funções. As estruturas não podem ser comparadas. Uma estrutura pode ser inicializada por uma lista de valores constantes dos membros: uma estrutura automática também pode ser inicializada por uma atribuição.

Vamos investigar as estruturas escrevendo algumas funções para manipular pontos e retângulos. Há pelo menos três métodos possíveis: passar os componentes separadamente, passar uma estrutura inteira, ou passar um ponteiro para ela. Cada um tem seus pontos fortes e fracos.

A primeira função, *cria_ponto*, tomará dois inteiros e retornará uma *struct* do tipo *ponto*:

Programa 6.1: Função cria_ponto().

```
/* cria_ponto: cria um ponto dos componentes x e y */
struct ponto cria_ponto(int x, int y)
{
    struct ponto temp;
    temp.x = x;
    temp.y = y;
    return temp;
}
```

Observe que não há conflito entre o nome do argumento e o membro com o mesmo nome; na verdade, a reutilização dos nomes reforça a relação.

*cria_ponto* pode agora ser usado para inicializar qualquer estrutura dinamicamente, ou para fornecer argumentos da estrutura para uma função:

```
struct retang tela;
struct ponto meio;
struct ponto cria_ponto(int,int);

tela.pt1 = cria_ponto(0,0);
tela.pt2 = cria_ponto(XMAX, YMAX);
meio = cria_ponto ((tela.pt1.x + tela.pt2.x)/2,
                   (tela.pt1.y + tela.pt2.y)/2);
```

O próximo passo é um conjunto de funções para fazer a aritmética com pontos. Por exemplo,

Programa 6.2: Função soma_ponto().

```
/* soma_ponto: soma dois pontos */
struct ponto soma_ponto(struct ponto p1, struct ponto p2)
{
    p1.x += p2.x;
    p1.y += p2.y;
    return p1;
}
```

Aqui os argumentos e o valor de retorno são estruturas. Incrementamos os componentes em *p1* ao invés de usar uma variável temporária explícita para enfatizar que os parâmetros da estrutura são passados por valor como outros quaisquer.

Como outro exemplo, a função *pt_em_retangulo* testa se um ponto está dentro de um retângulo, onde adotamos a convenção de que um retângulo inclui seus lados esquerdo e do fundo, mas não seus lados de cima e da direita:

Programa 6.3: Função para verificar se ponto está dentro de retângulo.

```
/* retorna 1 se p em r, 0 caso contrário */
int pt_em_retangulo(struct ponto p, struct retang r)
{
    return p.x >= r.pt1.x
           && p.x < r.pt2.x
           && p.y >= r.pt1.y
           && p.y < r.pt2.y;
}
```

Isso assume que o retângulo é representado em um formato-padrão onde as coordenadas *pt1* são menores que as coordenadas *pt2*. A função a seguir retorna um retângulo garantidamente em forma canônica:

Programa 6.4: Função para tornar coordenadas canônicas.

```
#define min(a, b) ((a) < (b) ? (a) : (b))
#define max(a, b) ((a) > (b) ? (a) : (b))

/* canonret: torna canônicas as coordenadas do retângulo */
struct retang canonret(struct retang r)
{
    struct retang temp;
    temp.pt1.x = min(r.pt1.x, r.pt2.x);
    temp.pt1.y = min(r.pt1.y, r.pt2.y);
    temp.pt2.x = max(r.pt1.x, r.pt2.x);
    temp.pt2.y = max(r.pt1.y, r.pt2.y);
    return temp;
}
```

Se uma estrutura maior tiver que ser passada para uma função, é geralmente mais eficiente se passar um ponteiro do que copiar a estrutura inteira. Os ponteiros de estrutura são semelhantes a ponteiros para variáveis comuns. A declaração

```
struct ponto *pp;
```

diz que *pp* é um ponteiro para uma estrutura do tipo `struct ponto`. Se *pp* aponta para uma estrutura do tipo *ponto*, $*pp$ é a estrutura, e $(*pp).x$ e $(*pp).y$ são seus membros. Para usar *pp*, poderíamos escrever, por exemplo.

```
struct ponto origem, *pp;

pp = &origem;
printf("origem é (%d,%d)\n", (*pp).x, (*pp).y);
```

Os parênteses são necessários em $(*pp).x$ porque a precedência do operador de membro de estrutura é maior do que $*$. A expressão $*pp.x$ significa $*(pp.x)$, que é ilegal aqui porque $x$ não é um ponteiro.

Os ponteiros para estruturas são usados com tanta frequência que existe uma notação alternativa usada como abreviação. Se *p* é um ponteiro para uma estrutura, então

```
p->membro_da_estrutura
```

refere-se ao membro em particular. (O operador `->` é um sinal de menos seguido imediatamente por >.) Assim, poderíamos escrever, no lugar do *printf* acima.

```
printf("origem é (%d,%d)\n", pp->x, pp->y);
```

Tanto . quanto `->` associam da esquerda para a direita, de forma que se tivermos

```
struct retang r, *rp = &r;
```

então estas quatro expressões são equivalentes:

```
r.pt1.x
rp->pt1.x
(r.pt1).x
(rp->pt1).x
```

Os operadores de estrutura . e `->`, juntamente com *()* para chamadas de função e *[]* para subscritos, estão no topo da hierarquia de precedência e, sendo assim, ligam muito fortemente. Por exemplo, dada a declaração

```
struct {
  int tam;
  char *cad;
} *p;
```

então

```
++p->tam
```

incrementa *tam*, e não *p*, pois a parentetização implícita é `++(p->tam)`. Os parênteses podem ser usados para alterarem a ligação: `(++p)->tam` incrementa *p* antes de acessar *tam*, e `(p++)->tam` incrementa *p* após. (Este último conjunto de parênteses é desnecessário.)

Da mesma forma, `*p->cad` obtém qualquer objeto apontado por *cad*; `*p->cad++` incrementa *cad* após acessar algo que ele aponta (assim como `*s++`); `(*p->cad)++` incrementa qualquer objeto apontado por *cad*; e `*p++->cad` incrementa *p* após acessar qualquer objeto apontado por *cad*.

## 6.3 Vetores de Estruturas

Considere a escrita de um programa para contar as ocorrências de cada palavra-chave em **C**. Precisamos de um vetor de cadeias de caracteres para conter os nomes, e um vetor de inteiros para os contadores. Uma possibilidade é usar dois vetores paralelos, *palavrachave* e *contapalavra*, como em

```
char *palavrachave[NCHAVES];
int contapalavra[NCHAVES];
```

Mas o fato dos vetores estarem em paralelo sugere uma organização diferente: um vetor de estruturas. Cada entrada de palavra-chave é um par:

```
char *palavra;
int contador;
```

e existe um vetor de pares. A declaração de estrutura

```
struct chave {
  char *palavra;
  int contador;
} tab_chaves[NCHAVES];
```

declara uma estrutura do tipo *chave*, define um vetor *tab_chaves* de estruturas deste tipo, e separa memória para tudo isto. Cada elemento do vetor é uma estrutura. Isto também poderia ser escrito por

```
struct chave {
  char *palavra;
  int contador;
}
struct chave tab_chaves[NCHAVES];
```

Como a estrutura *tab_chaves* contém um conjunto constante de nomes, é fácil torná-la uma variável externa e inicializá-la de uma vez por todas ao ser definida. A inicialização da estrutura é semelhante aos casos anteriores – a definição é seguida por uma lista de inicializadores entre chaves:

```
struct chave {
  char *palavra;
  int contador;
```

```
} tab_chaves[] ={
"auto", 0,
"break", 0,
"case", 0,
"char", 0,
"const", 0,
"continue", 0,
"default", 0,
/* ... */
"unsigned", 0,
"void", 0,
"volatile", 0,
"while", 0
};
```

Os inicializadores são listados em pares correspondentes aos membros da estrutura. Seria mais preciso delimitar os inicializadores para cada "linha" ou estrutura entre chaves, como em

```
{ "auto", 0 },
{ "break", 0 },
{ "case", 0 },
...
```

mas as chaves internas não são necessárias quando os inicializadores são variáveis simples ou cadeias de caracteres e quanto todos estão presentes. Como sempre, o número de entradas no vetor *tab_chave* será calculado se os inicializadores estiverem presentes e `[]` for deixado vazio.

O programa de contagem de palavras-chave começa com a definição de *tab_chave*. A rotina principal lê a entrada repetidamente chamando uma função *le_palavra* que obtém uma palavra de cada vez. Cada palavra é pesquisada em *tab_chave* com uma versão da função de pesquisa binária escrita no Capítulo 3. A lista de palavras-chave deve estar em ordem crescente na tabela.

Programa 6.5: Programa de contagem de palavras-chave.

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

#define MAXPAL 100

struct chave {
    char* palavra;
    int contador;
} tab_chaves[] = {
    {"auto", 0},
    {"break", 0},
    {"case", 0},
    {"char", 0},
    {"const", 0},
    {"continue", 0},
    {"default", 0},
    /* ... complete as palavras aqui */
    {"unsigned", 0},
    {"void", 0},
    {"volatile", 0},
    {"while", 0}
};
#define NCHAVES sizeof tab_chaves / sizeof (tab_chaves[0])
```

```
int le_palavra(char*, int);
int pesq_bin(char*, struct chave*, int);

/* conta palavras-chave em C */
int main()
{
    int n;
    char palavra[MAXPAL];

    while (le_palavra(palavra, MAXPAL) != EOF) {
        if (isalpha(palavra[0])) {
            if ((n = pesq_bin(palavra, tab_chaves, NCHAVES)) >= 0) {
                tab_chaves[n].contador++;
            }
        }
    }

    for (n = 0; n < NCHAVES; n++) {
        if (tab_chaves[n].contador > 0) {
            printf("%4d %s\n", tab_chaves[n].contador, tab_chaves[n].palavra);
        }
    }

    return 0;
}

/* pesq_bin: encontra palavra em tab[0]...tab[n-1] */
int pesq_bin(char* palavra, struct chave tab[], int n)
{
    int cond;
    int inicio = 0;
    int fim = n - 1;

    while (inicio <= fim) {
        int meio = (inicio + fim) / 2;

        if ((cond = strcmp(palavra, tab[meio].palavra)) < 0) {
            fim = meio - 1;
        } else if (cond > 0) {
            inicio = meio + 1;
        } else {
            return meio;
        }
    }

    return -1;
}
```

Mostraremos a função *le_palavra* mais adiante; por enquanto é suficiente dizer que cada chamada a *le_palavra* encontra uma palavra, que é copiada para o vetor indicado pelo seu primeiro argumento.

A quantidade *NCHAVES* é o número de palavras-chave em *tab_chaves*. Embora possamos contar este valor à mão, é muito mais fácil e seguro fazê-lo pela máquina, especialmente se a lista estiver sujeita a mudanças. Uma possibilidade seria terminar a lista de inicializadores com um

ponteiro nulo, e depois passar pelo vetor *tab_chave* até ser encontrado o final.

Mas não precisamos de tudo isto, pois o tamanho do vetor é completamente determinado em tempo de compilação. O tamanho do vetor é o tamanho de uma entrada vezes o número de entradas, de forma que o número de entradas é apenas

***tamanho** de tab_chaves / **tamanho** de struct chave*

C provê um operador unário em tempo de compilação chamado *sizeof*, que pode ser usado para calcular o tamanho de qualquer objeto. As expressões

```
sizeof objeto
```

e

```
sizeof (nome do tipo)
```

geram um inteiro igual ao tamanho do objeto especificado ou tipo em bytes. (Estritamente falando, *sizeof* produz um valor inteiro não-sinalizado cujo tipo, *size_t*, é definido no cabeçalho <*stddef.h*>). Um objeto pode ser uma variável ou um vetor ou estrutura. Um nome de tipo pode ser o nome de um tipo básico, como *int* ou *double*, ou um tipo derivado, como uma estrutura ou um ponteiro.

No nosso caso, o número de palavras-chave é o tamanho do vetor dividido pelo tamanho de um elemento. Este cálculo é usado em um comando *#define* para definir o valor de *NCHAVES*:

```
#define NCHAVES (sizeof tab_chaves / sizeof (struct chave))}
```

Uma outra forma de escrever isto é dividir o tamanho do vetor pelo tamanho de um elemento específico:

```
#define NCHAVES sizeof tab_chaves / sizeof (tab_chaves[0])
```

Este tem a vantagem de não precisar ser alterado se o tipo for mudado.

Um *sizeof* não pode ser usado em uma linha *#if* pois o pré-processador não analisa nomes de tipo. Mas a expressão no *#define* não é avaliada pelo pré-processador, de forma que o código aqui é válido.

Agora passemos à função *le_palavra*. Escrevemos uma *le_palavra* mais geral do que o necessário para este programa, mas não é complicada. *le_palavra* obtém a próxima "palavra" da entrada, onde uma palavra é ou uma string de letras e dígitos começando com uma letra, ou uma única letra. O valor da função é o primeiro caractere da palavra, ou *EOF* para o fim de arquivo, ou o próprio caractere se este não é alfabético.

Programa 6.6: Função le_palavra().

```
/* le_palavra: obtém próxima palavra ou caractere da entrada */
int le_palavra(char* palavra, int lim)
{
    int c, getch(void);
    void ungetch(int);
    char* p = palavra;

    while (isspace(c = getch()))
        ;

    if (c != EOF) {
        *p++ = c;
    }

    if (!isalpha(c)) {
```

```
        *p = '\0';
        return c;
    }

    for ( ; --lim > 1; p++) {
        if (!isalnum(*p = getch())) {
            ungetch(*p);
            break;
        }
    }

    *p = '\0';
    return palavra[0];
}
```

*le_palavra* usa as funções *getch* e *ungetch* que escrevemos no Capítulo 4. Quando a leitura de um código alfanumérico termina, *le_palavra* terá obtido um caractere a mais do que devia. A chamada a *ungetch* coloca esse caractere de volta à entrada para a próxima chamada. *le_palavra* também usa *isspace* para saltar o espaço em branco, *isalpha* para identificar letras, e *isalnum* para identificar letras e dígitos. Todos vêm do cabeçalho-padrão <*ctype.h*> .

**Exercício 6.1** Nossa versão de *le_palavra* não trabalha adequadamente com caracteres de sublinhado, constantes de string, comentários ou linhas do controle do pré-processador. Escreva uma versão mais aprimorada. ■

## 6.4 Ponteiros para Estruturas

Para ilustrar algumas das considerações envolvidas com ponteiros e vetores de estruturas, vamos escrever o programa de contagem de palavras novamente, desta vez usando ponteiros no lugar de índices de vetor. A declaração externa de *tab_chaves* não precisa mudar, mas *main* e *pesqbin* precisam de modificação.

Programa 6.7: Programa de contagem de palavras-chave usando ponteiros.

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#define MAXPAL 100

int le_palavra(char*, int);
struct chave* pesq_bin(char*, struct chave*, int);

/* conta palavras-chave em C; versão com ponteiros */
int main()
{
    char palavra[MAXPAL];
    struct chave* p;

    while (le_palavra(palavra, MAXPAL) != EOF) {
        if (isalpha(palavra[0])) {
            if ((p = pesq_bin(palavra, tab_chaves, NCHAVES)) != NULL) {
                p->contador++;
            }
        }
    }
```

```
    for (p = tab_chaves; p < tab_chaves + NCHAVES; p++) {
        if (p->contador > 0) {
            printf("%4d %s\n", p->contador, p->palavra);
        }
    }

    return 0;
}

/* pesq_bin: encontra palavra em tab[0]...tab[n-1] */
struct chave* pesq_bin(char* palavra, struct chave* tab, int n)
{
    int cond;
    struct chave* inicio = &tab[0];
    struct chave* fim = &tab[n];

    while (inicio < fim) {
        struct chave* meio = inicio + (fim - inicio) / 2;

        if ((cond = strcmp(palavra, meio->palavra)) < 0) {
            fim = meio;
        } else if (cond > 0) {
            inicio = meio + 1;
        } else {
            return meio;
        }
    }

    return NULL;
}
```

Há diversas coisas a observar aqui. Primeiro, a declaração de *pesq_bin* deve indicar que retorna um ponteiro para *struct chave* ao invés de um inteiro; isto é declarado tanto no protótipo de função quanto em *pesq_bin*. Se *pesq_bin* achar a palavra, ela retornará um ponteiro para ela; se falhar, retornará *NULL*.

Segundo, os elementos de *tab_chave* são agora acessados por meio de ponteiros, isso requer mudanças significativas em *pesq_bin*.

Os inicializadores para *inicio* e *fim* são agora ponteiros para o início e um objeto além do fim da tabela.

O cálculo do elemento do meio não pode mais ser simplesmente

```
meio = (inicio + fim) / 2 // ERRADO
```

pois a soma de dois ponteiros é ilegal. A subtração é legal, entretanto, de modo que `fim-inicio` é o número de elementos, e assim

```
meio = inicio + (fim - inicio) / 2
```

define *meio* apontando para o elemento na metade entre *inicio* e *fim*.

A mudança mais importante é ajustar o algoritmo para assegurar que não gere um ponteiro ilegal ou tente acessar um elemento fora do vetor. O problema é que *&tab[-1]* e *&tab[n]* estão fora dos limites do vetor *tab*. O primeiro é estritamente ilegal, e é ilegal *desreferenciar* o segundo. A definição da linguagem, entretanto, garante que a aritmética de ponteiro que envolve o primeiro elemento além do final de um vetor (ou seja, *&tab[n]* não funcionará corretamente.

Em *main* escrevemos

```
for (p = tab_chaves; p < tab_chave + NCHAVES; p++)
```

Se *p* é um ponteiro para uma estrutura, a aritmética com *p* leva em consideração o tamanho da estrutura, de forma que *p++* incrementa *p* pela quantidade correta para obter o próximo elemento do vetor de estruturas, e o teste para o laço na hora certa.

Não assuma, no entanto, que o tamanho de uma estrutura é a soma dos tamanhos de seus membros. Devido aos requisitos de alinhamento para diferentes objetos, pode haver "buracos" não nomeados em uma estrutura. Assim, por exemplo, se um *char* ocupa um byte e um *int* quatro bytes, a estrutura

```
struct {
  char c;
  int i;
};
```

pode exigir **oito** bytes, e não **cinco**. O operador *sizeof* retorna o valor apropriado.

Finalmente, um lembrete sobre o formato do programa: quando uma função retorna um tipo complicado com um ponteiro de estrutura, como em

```
struct chave *pesq_bin(char *palavra, struct chave *tab, int n)
```

o nome da função pode ser difícil de ser visto e encontrado com um editor de texto. Por isso um estilo alternativo às vezes é usado:

```
struct key *
pesqbin(char *palavra, struct chave *tab, int n)
```

Isto é uma questão de gosto pessoal; escolha a forma que desejar e use-a todo o tempo.

## 6.5 Estruturas Auto-referenciadas

Suponha que queremos lidar com o problema mais geral de contar as ocorrências de todos as palavras em alguma entrada. Como a lista de palavras não é conhecida a priori, não podemos ordená-la de forma conveniente e usar a pesquisa binária. Além disso, não podemos fazer uma pesquisa linear para cada palavra assim que chegar, para ver se já foi encontrada; o programa ficaria muito grande. (Mais precisamente, seu tempo de execução talvez cresça exponencialmente com o número de palavras da entrada.) Como podemos organizar os dados para trabalhar de forma eficiente com uma lista de palavras arbitrárias?

Uma solução é manter o conjunto de palavras vistas até o momento ordenadas o tempo inteiro, colocando cada palavra na sua posição correta em ordem assim que ela chegar. Isso não poderia ser feito deslocando-se as palavras em um vetor linear, entretanto, isso também leva muito tempo. Ao invés disso, usaremos uma estrutura de dados chamada árvore binária.

A árvore contém um "nodo" para cada palavra distinta; cada nodo contém

- um ponteiro para o texto da palavra,
- um contador do número de ocorrências,
- um ponteiro para o nodo filho esquerdo,
- um ponteiro para o nodo filho direito.

Nenhum nodo pode ter mais de dois filhos; ele só pode ter zero ou um.

Os nodos são mantidos de forma que em qualquer nodo a subárvore esquerda tenha somente palavras que sejam lexicograficamente menores do que a palavra no nodo, e a subárvore direita contenha somente palavras maiores. Esta é árvore para a sentença "*now is the time for all good men to come to the aid of party*", montada inserindo-se cada palavra a medida que for encontrada:

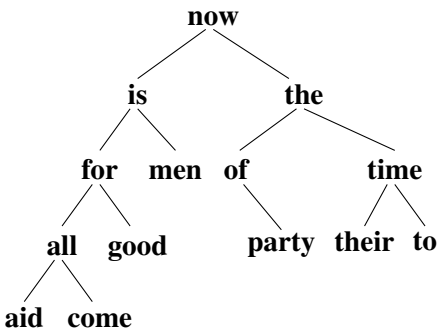



Figura 6.3: Árvore de palavras.

Para descobrir se uma nova palavra já está na árvore, **inicie** na raiz e **compare** a nova palavra com a palavra armazenada neste nodo. **Se** elas combinarem, a pergunta é respondida afirmativamente. **Se** a nova palavra **for menor que** a palavra na árvore, continue procurando no filho esquerdo, **caso contrário** no filho direito. **Se não** houver filho na direção solicitada, a nova palavra não está na árvore, e na verdade o campo vazio é o local correto para incluir a nova palavra. Este processo é recursivo, pois a pesquisa de qualquer nodo usa uma pesquisa a partir de um dos seus filhos. Portanto, as rotinas recursivas para inserção e impressão serão mais naturais.

Voltando à descrição de um nodo, ele é convenientemente representado por uma estrutura de quatro componentes

```
struct nodo_a {        // o nodo da árvore:
  char *palavra;       // aponta para o texto
  int contador;        // número da ocorrências
  struct nodo_a *esq; // filho esquerdo
  struct nodo_a *dir; // filho direito
};
```

Esta declaração recursiva de um nodo poderia parecer arriscada, mas está correta. Uma estrutura não pode conter um membro de si mesma, mas

```
struct nodo_a *esq;
```

declara *esq* como um ponteiro para uma estrutura *nodo_a*, e não o próprio *nodo_a*.

Ocasionalmente, precisa-se de uma variação das estruturas auto-referenciais: duas estruturas que referem-se uma à outra. O modo de se fazer isso é

```
struct t {
  ...
  struct s *p; /* p aponta para um s */
};
struct s {
  ...
  struct t *q; /* q aponta para um t */
};
```

O código para o programa inteiro é surpreendentemente pequeno, devido ao número de rotinas de suporte, como *le_palavra*, que já foram escritas. A rotina principal lê palavras com *le_palavra* e as instala na árvore com *arvore*.

Programa 6.8: Programa de busca de palavras com árvore binária.

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

#define MAXPAL 100

struct nodo_a {          // o nodo da árvore:
    char* palavra;       // aponta para o texto
    int contador;        // número da ocorrências
    struct nodo_a* esq; // filho esquerdo
    struct nodo_a* dir; // filho direito
};

struct nodo_a* arvore(struct nodo_a*, char*);
void impr_arv(struct nodo_a*);
int le_palavra(char*, int);

/* conta frequência de palavras */
int main()
{
    struct nodo_a* raiz = NULL;
    char palavra[MAXPAL];

    while (le_palavra(palavra, MAXPAL) != EOF) {
        if (isalpha(palavra[0])) {
            raiz = arvore(raiz, palavra);

            if(raiz == NULL) {
                printf("sem memória\n");
                return 1;
            }
        }
    }

    impr_arv(raiz);
    return 0;
}
```

A função *arvore* é recursiva. Uma palavra é apresentada por *main* no nível de cima (a raiz) da árvore. Em cada estágio, essa palavra é comparada à palavra já armazenada no nodo, e é pesquisada pela subárvore esquerda ou direita por meio de uma chamada recursiva a *arvore*. Eventualmente a palavra ou combina com algo já na árvore (quando o contador é incrementado), ou é encontrado um ponteiro nulo, indicando que um nodo deve ser criado e adicionado à árvore. Se um novo nodo for criado, *arvore* retorna um ponteiro para ele, que é instalado no nodo pai.

Programa 6.9: Função arvore().

```c
struct nodo_a* aloca(void);
char* dup_cad(char*);

/* arvore: inclui um nodo com w, em p ou abaixo */
struct nodo_a* arvore(struct nodo_a* p, char* w)
{
    int cond;

    if (p == NULL) { // uma nova palavra chegou
```

```
        p = aloca(); // cria um novo nodo
        p->palavra = dup_cad(w);
        p->contador = 1;
        p->esq = p->dir = NULL;
    } else if ((cond = strcmp(w, p->palavra)) == 0) {
        p->contador++;      // palavra repetida
    } else if (cond < 0) { // menor na subárvore esquerda
        p->esq = arvore(p->esq, w);
    } else {  // maior na subárvore direita
        p->dir = arvore(p->dir, w);
    }

    return p;
}
```

A memória para o novo nodo é obtida por uma rotina *aloca*, que retorna um ponteiro para um espaço livre adequado para manter um nodo da árvore, e a nova palavra é copiada para um local oculto por *dup_cad*. (Discutiremos estas rotinas mais adiante.) O contador é inicializado, e os dois filhos tornam-se nulos. Esta parte do código só é executada nas folhas das árvores, quando um novo código está sendo incluído. Omitimos a checagem de erro (inadmissível num programa de produção) nos valores retornados por *dup_cad* e *aloca*.

*impr_arv* imprime a árvore ordenada; em cada nodo, ela imprime a subárvore da esquerda (todas as palavras menores que esta palavra), depois a própria palavra, e depois a subárvore direita (todas as palavras maiores). Se você sente dúvidas sobre o funcionamento da recursão, simule *imp_arv* com as palavras mostradas na árvore do início desta seção.

Programa 6.10: Função impr_arv().

```
/* impr_arv: impressão ordenada da árvore p */
void impr_arv(struct nodo_a* p)
{
    if (p != NULL) {
        impr_arv(p->esq);
        printf("%4d %s\n", p->contador, p->palavra);
        impr_arv(p->dir);
    }
}
```

Uma nota prática: se a árvore se tomar "desbalanceada" devido às palavras não aparecerem em ordem randômica, o tempo de execução do programa pode crescer rapidamente. No pior caso, se as palavras já estiverem em ordem, este programa faz uma desgastante simulação da pesquisa linear. Há generalizações da árvore binária que não sofrem este comportamento no pior caso, mas não as descreveremos aqui.

Antes de deixamos este exemplo, é interessante fazermos um breve comentário sobre um problema relacionado com alocadores de memória. Evidentemente, é desejável que haja somente um alocador de memória no programa, muito embora ele aloque diferentes tipos de objetos. Mas se um alocador tem de processar pedidos, digamos, para ponteiros para *char* e ponteiros para *struct nodo*, duas questões surgem. Primeiro, como ele atende aos requisitos da maioria das máquinas reais de que objetos de certos tipos devam satisfazer restrições de alinhamento (por exemplo, inteiros normalmente devem ser localizados em endereços pares)? Segundo, que declarações podem lidar com o fato de que os alocadores devem necessariamente retornar diferentes tipos de ponteiros?

Os requisitos de alinhamento geralmente podem ser satisfeitos com facilidade, ao custo de algum espaço perdido, assegurando-se de que o alocador sempre retorne um ponteiro que satisfaça todas as restrições de alinhamento. O *aloca* do Capítulo 5 não garante qualquer alinhamento particular, de

modo que usaremos a função *malloc*, da biblioteca-padrão, que garante. No Capítulo 8 mostraremos uma forma de implementar *malloc*.

A questão da declaração de tipo para uma função como *malloc* é um problema para qualquer linguagem que leve a sério sua verificação de tipo. Nas versões de **C pré-ANSI**, o método adequado era declarar que *malloc* retorna um ponteiro para *void*, e depois forçar explicitamente o ponteiro para o tipo desejado com um molde. Nas versões posteriores de **C**, o molde é automático e fazer o molde explícito é potencialmente perigoso, pois poderia mascarar um erro, obrigando o compilador a fazer uma conversão errada. *malloc* e as rotinas relacionadas são declaradas no cabeçalho-padrão <*stdlib.h*>.

Assim, *aloca* pode ser escrita da seguinte forma:

Programa 6.11: Função aloca() para nó da árvore.

```
#include <stdlib.h>

/* aloca:  cria um nodo_a */
struct nodo_a* aloca(void)
{
    return (struct nodo_a*) malloc(sizeof(struct nodo_a));
}
```

**Adendo 6.5.1 — Conversão automática de void *.** Na versão **ANSI** e posteriores de **C**, o ponteiro *void * tipo* é um ponteiro genérico e pode apontar para qualquer tipo de dado. Como *void ** nunca aponta para nada por si próprio, o compilador sempre fará sua moldagem para algum tipo e desta forma a moldagem explícita não é estritamente necessária.

Desta forma, em código semelhante a

```
struct nodo_a *p = malloc(sizeof(struct nodo_a));
```

ou

```
return malloc(sizeof(struct nodo_a));
```

o compilador fará a moldagem silenciosamente e sem *warning*. É sempre melhor deixar o próprio compilador fazer a moldagem automática.

*dup_cad* simplesmente copia a string informada pelo seu argumento em um local seguro, obtido por uma chamada a *malloc*:

Programa 6.12: Função dup_cad().

```
char* dup_cad(char* s) /* cria uma duplicata de s */
{
    char* p = malloc(strlen(s) + 1); /* +1 para '\0' */

    if (p != NULL) {
        strcpy(p, s);
    }

    return p;
}
```

*malloc* retorna *NULL* se não houver espaço disponível; *dup_cad* passa este valor adiante, deixando a verificação de erro à função que o chamou.

A memória obtida chamando *malloc* pode ser liberada para reutilização chamando-se *free*; veja nos Capítulo 7 e Capítulo 8.

**Exercício 6.2** Escreva um programa que leia um programa em C e imprima em ordem alfabética cada grupo de nomes de variáveis idênticas nos seus seis primeiros caracteres, mas diferentes depois disso. Não conte palavras dentro de string e comentários. Faça com que seis seja um parâmetro que possa ser definido pela linha de comando. ■

**Exercício 6.3** Escreva um programa de referência cruzada que imprima uma lista de todas as palavras em um documento e, para cada palavra, uma lista de números de linhas onde ela ocorre. Remova palavras desnecessárias como "o", "e", e assim por diante. ■

**Exercício 6.4** Escreva um programa que imprima as palavras distintas na sua entrada em ordem decrescente por frequência de ocorrência. Preceda cada palavra pelo seu contador. ■

.

## 6.6 Pesquisa em Tabela

Nesta seção escreveremos os detalhes de um pacote de pesquisa de tabela, ilustrando mais aspectos das estruturas. Este código é tipico do que poderia ser encontrado nas rotinas de gerenciamento da tabela de símbolos de um macroprocessador ou compilador. Por exemplo, considere o comando *#define*. Quando uma linha como

```
#define DENTRO 1
```

é encontrada, o nome *DENTRO* e o texto substituto 1 são armazenados em uma tabela. Mais tarde, quando o nome *DENTRO* aparecer em um comando como

```
estado = DENTRO;
```

ele será substituído por 1.

Há rotinas que manipulam os nomes e textos substitutos. $instala(s,t)$ registra o nome $s$ e o texto substituto $t$ em uma tabela; $s$ e $t$ são apenas cadeias de caracteres. *pesquisa(s)* procura $s$ na tabela, e retorna um ponteiro para o local onde foi encontrada, ou *NULL* se não estava lá.

O algoritmo usado é uma pesquisa "*hash*" – o nome recebido é convertido num pequeno inteiro não negativo, que é depois usado para indexar um vetor de ponteiros. Um elemento do vetor aponta para o início de uma lista encadeada de blocos descrevendo nomes que possuem esse valor *hash*. Ele é *NULL* se nenhum nome gerou esse valor *hash*.

Um bloco na lista é uma estrutura contendo ponteiros para o nome, o texto substituto, e o próximo bloco na lista. Um próximo ponteiro nulo marca o fim da lista encadeada.

```
struct lista {          // entrada da tabela:
    struct lista* prox; // prox.  entrada na cadeia
    char* nome;         // nome definido
    char* defn;         // texto substituto
};
```

O vetor de ponteiros é somente

```
#define TAMHASH 101

static struct lista* tab_hash[TAMHASH]; // tabela de ponteiros
```

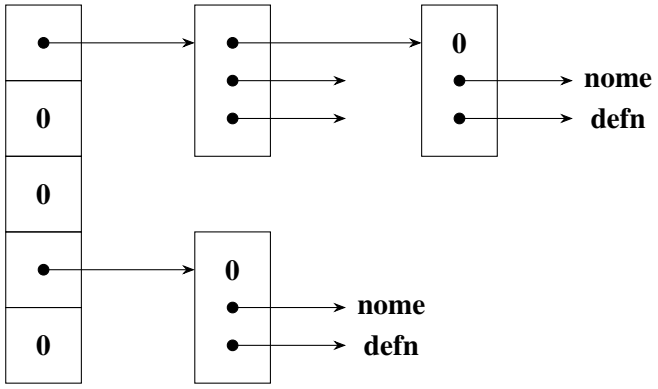



Figura 6.4: Estrutura dos blocos da pesquisa.

A função de "*hash*", que é usada por *pesquisa* e *instala*, inclui cada valor de caractere na cadeia a uma combinação misturada dos anteriores e retorna o módulo restante do tamanho do vetor. Esta não é a melhor função de *hash* possível, mas é curta e eficiente.

Programa 6.13: Função hash().

```
/* hash: forma valor de hash para a cadeia s */
unsigned hash(char* s)
{
    unsigned val_hash;

    for (val_hash = 0; *s != '\0'; s++) {
        val_hash = *s + 31 * val_hash;
    }

    return val_hash % TAMHASH;
}
```

A aritmética não-sinalizada assegura que o valor de *hash* seja não-negativo.

O processo de *hash* produz um índice inicial no vetor *tab_hash*: se a cadeia for encontrada em algum local, ela está na lista de bloco começando aí. A busca é executada por *pesquisa*. Se *pesquisa* descobrir que a entrada já está presente, ela retorna um ponteiro para ela; se não, retorna um *NULL*.

Programa 6.14: Função pesquisa() em lista encadeada.

```
/* pesquisa: procura s em tab_hash */
struct lista* pesquisa(char* s)
{
    struct lista* np;

    for (np = tab_hash[hash(s)]; np != NULL; np = np->prox) {
        if (strcmp(s, np -> nome) == 0) {
            return np; // achou
        }
    }

    return NULL; // não achou
}
```

O laço *for* em *pesquisa* é o idioma-padrão para se percorrer uma lista encadeada:

```
for (ptr = primeiro; ptr != NULL; ptr = ptr- >prox)
...
```

*instala* usa *pesquisa* para determinar se o nome sendo instalado já está presente: se estiver, a nova definição substituirá a antiga. Caso contrário, uma nova entrada é criada. *instala* retorna *NULL* se por qualquer motivo não há espaço para uma nova entrada.

Programa 6.15: Função instala().

```
struct lista* pesquisa(char*);
char* dup_cad(char*);

/* instala:  coloca (nome, defn) em tab_hash */
struct lista* instala(char* nome, char* defn)
{
    struct lista* np = NULL;
    unsigned val_hash = 0;

    if ((np = pesquisa(nome)) == NULL) { // não achou
        np = (struct lista*) malloc(sizeof(*np));

        if (np == NULL || (np->nome = dup_cad(nome)) == NULL) {
            return NULL;
        }

        val_hash = hash(nome);
        np->prox = tab_hash[val_hash];
        tab_hash[val_hash] = np;
    } else {  // já existe
        free((void*) np->defn); // libera defn anterior
    }

    if ((np->defn = dup_cad(defn)) == NULL) {
        return NULL;
    }

    return np;
}
```

**Exercício 6.5** Escreva uma função *indef* que removerá um nome e definição da tabela mantida por *pesquisa* e *instala*. ■

**Exercício 6.6** Implemente uma versão simples do processador *#define* (ou seja, sem argumentos) adaptável ao uso com programas em **C**, baseado nas rotinas desta seção. As funções *getch* e *ungetch* também podem ser úteis. ■

## 6.7 Typedef

**C** provê uma facilidade chamada *typedef* para criar novos nomes de tipo de dado. Por exemplo, a declaração

```
typedef int Tamanho;
```

torna o nome *Tamanho* um sinônimo para *int*. O tipo *Tamanho* pode ser usado em declarações, moldes etc., exatamente da mesma forma que o tipo *int*:

```
Tamanho tam, maximo;
Tamanho *tamanhos[];
```

Semelhantemente, a declaração

```
typedef char *String;
```

torna *String* um sinônimo de *char ** ou um ponteiro de caractere, que pode então ser usado nas declarações e moldes:

```
String p, ptrlinha[MAXLINHAS], aloca(int);
int strcmp(String, String);
p = (String)malloc(100);
```

Observe que o tipo sendo declarado em um *typedef* aparece na posição de um nome de variável, e não logo após a palavra *typedef*. Sintaticamente, *typedef* é como as classes de armazenamento *extern*, *static*, etc. Usamos nomes iniciados com letra maiúscula para *typedefs* a fim de que sejam destacados.

Como um exemplo mais complicado, poderíamos criar *typedefs* para os nodos de árvore mostrados anteriormente neste capítulo

```
typedef  struct nodo_t* Arvoreptr;

typedef struct nodo_a { // o nodo da árvore:
    char* palavra;      // aponta para o texto
    int contador;       // número de ocorrências
    Arvoreptr esq;      // filho esquerdo
    Arvoreptr dir;      // filho direito
} Nodoarvore;
```

Isso cria duas novas palavras-chave para tipo, chamadas *Nodoarvore* (uma estrutura) e *Arvoreptr* (um ponteiro para estrutura). Depois, a rotina *aloca* poderia tornar-se

```
Arvoreptr aloca(void)
{
  return (Arvoreptr) malloc(sizeof(Nodoarvore));
}
```

Deve ser enfatizado que uma declaração *typedef* não cria um novo tipo em qualquer sentido; ela simplesmente inclui um novo nome para algum tipo existente. E também não existe nova semântica: as variáveis declaradas desta forma têm exatamente as mesmas propriedades das variáveis cujas declarações são indicadas explicitamente. Com efeito, *typedef* é como *#define*, exceto que, por ser interpretada pelo compilador, pode lidar com substituições textuais além das capacidades do pré-processador. Por exemplo,

```
typedef int (*PPF)(char *, char *);
```

cria o tipo *PPF*, de "ponteiro para função (de dois argumentos *char**) retornando int", que pode ser usado em contextos como

```
PPF strcmp, numcmp;
```

no programa de ordenação do Capítulo 5.

Além de problemas puramente estéticos, há dois motivos principais para se usar *typedefs*. O primeiro é parametrizar um programa contra problemas de portabilidade. Se *typedefs* forem usados

para tipos de dados que possam ser dependentes da máquina, somente os *typedefs* precisam ser alterados quando o programa for movido. Uma situação comum é usar nomes de *typedef* para várias quantidades inteiras, e depois fazer um conjunto adequado de escolhas de *short*, *int* e *long* para cada máquina hospedeira. Os tipos como *size_t* e *ptrdiff_t* das bibliotecas-padrão são exemplos.

A segunda finalidade dos *typedefs* é dar uma melhor documentação para um programa – um tipo chamado *Arvoreptr* pode ser mais fácil de se entender do que um declarado apenas como um ponteiro para uma estrutura complicada.

## 6.8 Uniões

Uma união é uma variável que pode conter (em momentos diferentes) objetos de diferentes tipos e tamanhos, com o compilador tomando conta dos requisitos de tamanho e alinhamento. As uniões fornecem um meio de manipular diferentes tipos de dados em uma única área da memória, sem a necessidade de qualquer informação dependente de máquina no programa. Elas são semelhantes aos registros variantes do Pascal.

Como um exemplo do tipo que poderia ser encontrado em um gerenciador de tabela de simbolo de um compilador, suponha que uma constante possa ser *int*, *float* ou um ponteiro para caracteres. O valor de uma constante particular deve ser armazenado em uma variável do tipo apropriado, sendo mais conveniente para o gerenciamento de tabela que o valor ocupe a mesma quantidade de memória e seja guardado no mesmo lugar, independente do seu tipo. Esta é a finalidade de uma união – uma única variável que pode conter legitimamente qualquer um dentre diversos tipos. A sintaxe é baseada em estruturas:

```
union etiqueta{
  int vali;
  float valf;
  char *vals;
} u;
```

A variável *u* terá tamanho suficiente para conter o maior dos três tipos: o tamanho específico é dependente da implementação. Qualquer um desses tipos pode ser atribuído a *u* e depois usado em expressões, desde que o uso seja coerente: o tipo recuperado deve ser o tipo mais recentemente armazenado. O programador é responsável por cuidar de que tipo está atualmente armazenado em uma união; os resultados dependem da implementação caso algo seja armazenado como um tipo e extraído como outro.

Sintaticamente, os membros de uma união são acessados como

```
nome_união.membro
```

ou

```
ponteiro_união->membro
```

assim como em estruturas. Se a variável *tipo_u* é usada para manter o tipo corrente armazenado em *u*, então poderíamos ver um código como

```
if (tipo_u == INT)
    printf("%d\n", u.vali);
else if (tipo_u == FLOAT)
    printf("%f\n", u.valf);
else if (tipo_u == STRING)
    printf("%s\n", u.vals);
else
    printf("Tipo %d errado em tipo_u\n", tipo_u);
```

As uniões podem ocorrer dentro de estruturas e vetores, e vice-versa. A notação para acessar um membro de uma união em uma estrutura (ou vice-versa) é idêntica à de estruturas encaixadas. Por exemplo, no vetor de estruturas definido por

```
struct{
  char *nome;
  int sinalizadores;
  int tipo_u;
  union {
    int vali;
    float valf;
    char *vals;
  } u;
} tabsimb[NSIMB];
```

o membro *vali* é referenciado como segue

```
tabsimb[i].u.vali
```

e o primeiro caractere da string *vals* por um dentre

```
*tabsimb[i].u.vals
tabsimb[i].u.vals[0]
```

Isto é, uma união é uma estrutura em que todos os membros possuem deslocamento zero a partir da base, a estrutura é grande o suficiente para conter o membro "mais largo", e o alinhamento é adequado a todos os tipos na união. As mesmas operações são permitidas com uniões e estruturas: atribuição ou cópia como unidade, tomada de endereço e acesso a um membro.

Uma união só pode ser inicializada com um valor do tipo de seu primeiro membro; assim, a união *u* descrita anteriormente só pode ser inicializada com um valor inteiro.

O alocador de memória no Capítulo 8 mostra como uma união pode ser usada para forçar uma variável a ser alinhada em um tipo particular de limite de memória.

## 6.9 Campos de Bit

Quando a memória é escassa, pode ser necessário empacotar diversos objetos em uma única palavra de máquina; um uso comum seria um conjunto de sinalizadores de único bit em aplicações como tabelas de símbolo do compilador. Os formatos de dados externamente impostos, como interfaces para dispositivos de hardware, geralmente também exigem a capacidade de trabalhar com pedaços de uma palavra.

Imagine um fragmento de um compilador que manipule uma tabela de símbolos. Cada identificador em um programa possui certas informações associadas a ele, por exemplo, se é ou não uma palavra-chave, se é ou não externo e/ou estático, e assim por diante. A forma mais compacta de codificar tais informações está sob a forma de um conjunto de sinalizadores de 1 bit em um único *char* ou *int*.

A forma geral com que isto é feito é definir um conjunto de "máscaras" correspondentes às posições de bit relevantes, como em

```
#define PCHAVE 01
#define EXTERNO 02
#define ESTATICO 04
```

ou

```
enum {PCHAVE = 01, EXTERNO = 02, ESTATICO = 04};
```

Os números devem ser potências de dois. Depois disso, acessar os bits torna-se uma questão de "manuseio de bit" com os operadores de deslocamento, máscara e complemento, descritos no Capítulo 2.

Certos idiomas aparecem frequentemente:

```
flags |= EXTERNO | ESTATICO;
```

ativa os bits *EXTERNO* e *ESTATICO* nos flags sinalizadores enquanto

```
flags &= ~(EXTERNO | ESTATICO);
```

os desativa, e

```
if ((flags & (EXTERNO | ESTATICO)) == 0) ...
```

é verdadeiro se os dois bits estiverem desativados.

Embora estas construções sejam facilmente dominadas, como alternativa **C** oferece a capacidade de definir e acessar campos dentro de uma palavra diretamente ao invés de usar operadores lógicos bit-a-bit. Um campo de bit, ou campo para abreviar, é um conjunto de bits adjacentes dentro de uma única unidade de armazenamento definida pela implementação que chamaremos "palavra". A sintaxe da definição e acesso dos campos é baseada em estruturas. Por exemplo, a tabela de símbolo *#define* acima poderia ser substituída pela definição de três campos:

```
struct {
  unsigned int e_pchave : 1;
  unsigned int e_externo : 1;
  unsigned int e_estatico : 1;
} flags;
```

Isto define uma variável chamada *flags* que contém três campos de 1 bit. O número após os dois pontos representa a largura do campo em bits. Os campos são declarados `unsigned int` para se ter certeza de serem quantidades não sinalizadas.

Os campos individuais são referenciados da mesma forma que outros membros de estrutura: *flags.e_pchave*, *flags.e_externo*, etc. Os campos comportam-se como pequenos inteiros, e podem participar em expressões aritméticas assim como outros inteiros. Assim, os exemplos anteriores podem ser escritos mais naturalmente por

```
flags.e_externo = flags.e_estatico = 1;
```

para ativar os bits;

```
flags.e_externo = flags.e_estatico = 0;
```

para desativá-los; e

```
if (flags.e_externo == 0 && flags.e_estatico == 0)
...
```

para testá-los.

Quase tudo a respeito de campos depende da implementação. Se um campo pode ultrapassar um limite de palavra, é definido pela implementação. Os campos não precisam ser nomeados; campos não nomeados (somente os dois pontos e um tamanho) são usados para preenchimento. O tamanho especial 0 pode ser usado para forçar o alinhamento no próximo limite de palavra.

Os campos são atribuídos da esquerda para a direita em algumas máquinas e da direita para a esquerda em outras. Isso significa que, embora os campos sejam úteis para manterem estruturas de dados definidas internamente, a questão de qual extremo vem primeiro deve ser cuidadosamente considerada quando se apanham dados definidos externamente: os programas que dependem dessas

coisas não são portáteis. Os campos só podem ser declarados como *int*: por questão de portabilidade, especifique *signed* ou *unsigned* explicitamente. Eles não são vetores, e não possuem endereços, de forma que o operador **&** não pode ser aplicado aos mesmos.

# 7. Entrada e Saída

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

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

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

## 7.1 Entrada e Saída-Padrão

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

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

```
int getchar (void)
```

*getchar* retorna o próximo caractere da entrada toda vez que é chamado, ou *EOF* quando encontrar o fim do arquivo. A constante simbólica *EOF* é definida em <*stdio.h*>. O valor é normalmente $-1$,

mas os testes devem ser escritos em termos de *EOF* para que se tornem independentes do valor específico.

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

```
prog < arquivo
```

faz com que *prog* leia caracteres do arquivo. A troca da entrada é feita de tal forma que o próprio *prog* ignore a mudança; em particular, a string "< *arquivo*" não é incluída nos argumentos da linha de comando em *argv*. A troca da entrada é também invisível se a entrada vier de um outro programa por meio de um mecanismo de duto (*pipes*): em alguns sistemas, a linha de comando

```
outroprog | prog
```

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

A função

```
int putchar(int)
```

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

```
prog > arquivo_saida;
```

escreverá a saída-padrão em *arquivo_saida*. Se as canalizações forem suportadas,

```
prog | outroprog
```

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

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

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

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

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

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

Programa 7.1: Programa minusculo().

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

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

    while ((c = getchar()) != EOF) {
```

```
        putchar(tolower(c));
    }

    return 0;
}
```

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

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

## 7.2 Saída Formatada – printf

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

```
int printf(char  *formato, arq1, arq2, ...)
```

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

A cadeia de formato contém dois tipos de objetos: caracteres comuns, que são copiados no fluxo de saída, e especificações de conversão, cada uma delas causando uma conversão e impressão do próximo argumento sucessivo a *printf*.

Cada especificação de conversão começa com um % e termina com um caractere de conversão. Entre o % e o caractere de conversão pode haver, em ordem:

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

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

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

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

A maioria das conversões de formato já foi ilustrada em capítulos anteriores. Uma exceção é a precisão relacionada a cadeias de caracteres. A tabela a seguir mostra o efeito de uma série de especificações para "*hello, world*" (12 caracteres). Colocamos sinais de dois pontos ao redor de cada campo para que você possa ver sua extensão.

| Caractere | Tipo de Argumento, Impresso como |
|---|---|
| d,i | *int*; número decimal. |
| o | *unsigned int*; número octal não sinalizado (sem um zero inicial). |
| x,X | *unsigned int*; número hexadecimal não sinalizado (sem um $0x$ ou $0X$ inicial), usando *abcdef* ou *ABCDEF* para $10, ...., 15$. |
| u | *unsigned int*; número decimal não sinalizado. |
| c | *int*; único caractere. |
| s | *char **; imprime caracteres da cadeia até um '\0' ou número de caracteres indicado na precisão. |
| f | *double*; $[-]m.dddddd$, onde o número de *ds* é dado pela precisão (default 6). |
| e,E | *double*; $[-]m.ddddde+/-xx$ ou $[-]m.ddddddE+/-xx$, onde número de *ds* é dado pela precisão (default 6). |
| g,G | *double*; usa *%e* ou *%E* se o expoente for menor que $-4$ ou maior ou igual à precisão; caso contrário, usa *%f*. Zeros adicionais e um ponto decimal final não são impressos. |
| p | *void **; ponteiro (representação dependente da implementação). |
| % | nenhum argumento é convertido; imprime um %. |

Tabela 7.1: Conversões Básicas de *printf*

```
:%s:          :hello, world:
:%10s:        :hello, world:
:%.10s:       :hello, wor:
:%-10s:       :hello, world:
:%.15s:       :hello, world:
:%-15s:       :hello, world   :
:%15.10s:     :     hello, wor:
:%-15.10s:    :hello, wor     :
```

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

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

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

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

*sprintf* formata os argumentos em *arg1*, *arg2*, etc., de acordo com o formato, mas coloca o resultado na cadeia ao invés de enviá-lo à saída-padrão; a string deve ter tamanho suficiente para receber o resultado.

**Exercício 7.2** Escreva um programa que imprima uma entrada qualquer de forma compreensível. No mínimo, ele deve imprimir caracteres não-visíveis em octal ou hexadecimal, segundo a preferência local, e quebrar linhas longas. ■

## 7.3 Listas de Argumentos de Tamanho Variável

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

A declaração apropriada para *printf* é

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

onde a declaração ... significa que o número e tipos destes argumentos pode variar. A declaração só pode aparecer ao final de uma lista de argumentos. Nosso *minprintf* é declarado da seguinte forma:

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

não retornaremos o contador de caracteres, como faz *printf*.

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

O tipo *va_list* é usado para declarar uma variável que referir-se-á a cada argumento por sua vez; em *minprintf*, esta variável é chamada *pa*, de "**ponteiro de argumento**" em português. A macro *va_start* inicializa *pa* de forma que aponta para o primeiro argumento não nomeado. Ela deve ser chamada uma vez antes que *pa* seja usado. Deve haver pelo menos um argumento nomeado; o argumento nomeado final é usado por *va_start* para dar início.

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

Estas propriedades formam a base do nosso *printf* simplificado:

Programa 7.2: Programa minprintf().

```
#include <stdarg.h>

/* minprintf: printf mínimo com lista de argumentos variável */
void minprintf(char* fmt, ...)
{
    va_list pa; // aponta para cada argumento não nomeado
    char* p, *valc;
    int vali;
    double vald;
    va_start(pa, fmt); // faz pa apontar para primeiro arg.

    for (p = fmt; *p; p++) {
        if (*p != '%') {
            putchar(*p);
            continue;
        }

        switch (*++p) {
        case 'd':
            vali = va_arg(pa, int);
            printf("%d", vali);
            break;

        case 'f':
            vald = va_arg(pa, double);
```

```
            printf("%f", vald);
            break;

        case 's':
            for (valc = va_arg(pa, char*); *valc; valc++) {
                putchar(*valc);
            }

            break;

        default:
            putchar(*p);
            break;
        }
    }

    va_end(pa); // finaliza
}
```

**Exercício 7.3** Modifique *minprintf* para lidar com outras facilidades de *printf*. ■

## 7.4 Entrada Formatada - Scanf

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

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

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

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

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

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

Ela analisa a cadeia de acordo com o formato, e armazena os valores resultantes em *arg1*, *arg2*, etc. Estes argumentos devem ser ponteiros.

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

- Espaço ou tabulações. Se um espaço ou tabulação ocorre em alguma posição no formato, qualquer espaço ou tabulação na entrada correspondente será ignorado.
- Caracteres comuns (não %), que devem combinar com o próximo caractere não espaço do fluxo de entrada.

- Especificações de conversão, consistindo no caractere %, um caractere * opcional de supressão de atribuição, um número opcional especificando um tamanho máximo de campo, um *h*, *l* ou *L* opcional indicando o tamanho do destino, e um caractere de conversão.

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

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

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

Tabela 7.2: Conversões Básicas de *scanf*

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

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

Programa 7.3: Programa de calculadora elementar.

```
#include <stdio.h>

int main()  /* calculadora elementar */
{
    double soma, v;
    soma = 0;

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

```
    }

    return 0;
}
```

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

```
25 dez 2020
```

O comando *scanf* seria

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

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

Nenhum & é usado com nomes, pois um nome de vetor já é um ponteiro. Os caracteres literais podem aparecer na cadeia de formato de *scanf*; eles devem combinar com os mesmos caracteres na entrada. Assim, poderíamos ler datas do formato dd/mm/aa com este comando *scanf*:

```
int dia, mes, ano;

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

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

```
while (lelinha(linha, sizeof(linha)) > 0) {
  if (sscanf(linha, "%d %s %d", &dia, nomemes, &ano) == 3)
      printf("válido: %s\n", linha); // forma 25 Dez 2020
  else if (sscanf(linha, "%d/%d/%d", &mes, &dia, &ano) == 3)
      printf("válido: %s\n", linha); // forma mm/dd/aa
  else
       printf("inválido: %s\n", linha); // forma inválida
}
```

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

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

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

ao invés de

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

Este erro geralmente não é detectado em tempo de compilação.

**Exercício 7.4** Escreva uma versão particular de *scanf* semelhante a *minprintf* da seção anterior. ■

**Exercício 7.5** Reescreva a calculadora pós-fixada do Capítulo 4 para usar *scanf* e *sscanf* para realizar a entrada e conversão de números. ■

## 7.5 Acesso a Arquivos

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

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

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

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

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

As regras são simples. Antes que possa ser lido ou gravado, um arquivo deve ser aberto pela função de biblioteca *fopen*. *fopen* usa um nome externo como *x.c* ou *y.c*, faz alguma tarefa inicial e negociação com o sistema operacional (sendo que não estamos interessados nos detalhes), e retorna um ponteiro a ser usado em leituras ou escritas subsequentes no arquivo.

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

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

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

A chamada a *fopen* em um programa é

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

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

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

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

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

*getc* retorna o próximo caractere do fluxo referenciado por *fp*; ele retorna *EOF* para fim de arquivo ou erro. *putc* é uma função de saída:

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

*putc* escreve *c* no arquivo *fp* e retorna o caractere escrito; ou *EOF* se houver erro. Assim como *getchar* e *putchar*, *getc* e *putc* podem ser macros no lugar de funções.

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

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

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

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

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

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

Programa 7.4: Programa cat.

```
#include <stdio.h>

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

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

    return 0;
}

/* copia_arq:  copia arquivo ifp para arquivo ofp */
```

```
void copia_arq(FILE* ifp, FILE* ofp)
{
    int c;

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

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

A função

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

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

## 7.6 Tratamento de Erro – Stderr e Exit

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

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

Programa 7.5: Programa cat com saída de erro.

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

/* cat:  concatena arquivos, versão 2 */
int main(int argc, char* argv[])
{
    FILE* fp;
    void copia_arq(FILE*, FILE*);
    char* prog = argv[0];  // nome de programa p/ erros

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

```
            } else {
                copia_arq(fp, stdout);
                fclose(fp);
            }
        }

    if (ferror(stdout)) {
        fprintf(stderr, "%s: erro escrevendo em stdout\n", prog);
        exit(2);
    }

    exit(0);
}
```

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

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

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

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

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

Embora os erros de saída sejam raros, eles acontecem (por exemplo, se um disco estiver cheio), de forma que um programa em produção deve checar isto também. A função *feof(FILE *)* é semelhante a *ferror*; ela retorna não-zero se houve um fim de arquivo no arquivo especificado.

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

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

## 7.7 Entrada e Saída de Linhas

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

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

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

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

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

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

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

C99 A função *gets()* é considerada perigosa pois não tem nenhuma proteção contra *overflow* da entrada. Na versão **C99**, a função *gets()* foi considerada obsoleta e você receberá uma mensagem de alarme do compilador desaconselhando fortemente seu uso.
Na versão **C11** *gets()* foi completamente eliminada. Prefira *fgets()*.

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

Programa 7.6: Funções fgets e fputs.

```
/* fgets:  obtém no máximo n-1 caracteres, mais um null de iop */
char* fgets(char* s, int n, FILE* iop)
{
    register int c;
    register char* cs;
    cs = s;

    while (--n > 0 && (c = getc(iop)) != EOF)
        if ((*cs++ = c) == '\n') {
            break;
        }

    *cs = '\0';
    return (c == EOF && cs == s) ? NULL : s;
}

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

    while (c = *s++) {
        putc(c, iop);
    }

    return ferror(iop) ? EOF : 0;
}
```

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

Não é difícil a implementação de nosso *lelinha* a partir de *fgets*:

Programa 7.7: Função lelinha() com fgets()..

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

**Exercício 7.6** Escreva um programa para comparar dois arquivos, imprimindo a primeira linha onde eles diferem. ■

**Exercício 7.7** Modifique o programa de pesquisa de padrão do Capítulo 5 para aceitar sua entrada a partir de um conjunto de arquivos nomeados ou, se nenhum arquivo for nomeado como argumento, da entrada-padrão. Seria bom imprimir o nome do arquivo quando uma linha como padrão especificado fosse encontrada? ■

**Exercício 7.8** Escreva um programa que imprima um conjunto de arquivos, iniciando cada um com uma nova página, com um título e um contador de página corrente para cada arquivo. ■

## 7.8 Algumas Funções Diversas

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

### 7.8.1 Operações de Cadeia

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

| | |
|---|---|
| *strcat(s,t)* | concatena *t* ao final de *s* |
| *strncat(s,t,n)* | concatena *n* caracteres de *t* ao final de *s* |
| strcmp(s,t) | retorna negativo, zero, ou positivo para $s < t$, $s == t$, ou $s > t$ |
| *strncmp(s,t,n)* | o mesmo que *strcmp* mas somente com os primeiros *n* caracteres |
| *strcpy(s,t)* | copia *t* para *s* |
| *strncp(s,t,n)* | copia no máximo *n* caracteres de *t* para *s* |
| *strIen(s)* | retorna o tamanho de *s* |
| strchr(s,c) | retorna ponteiro para primeiro *c* em *s*, ou *NULL* se não for achado |
| *strrchr(s,c)* | retorna ponteiro para último *c* em *s*, ou *NULL* se não for achado |

### 7.8.2 Teste e Conversão de Classe de Caracteres

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

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

### 7.8.3 Ungetc

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

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

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

### 7.8.4 Execução de Comando

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

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

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

### 7.8.5 Gerenciamento de Memória

As funções *malloc* e *calloc* obtêm blocos de memória dinamicamente.

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

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

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

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

O ponteiro retornado por *malloc* ou *calloc* tem o alinhamento adequado ao objeto em questão, e não necessita ser moldado para o tipo apropriado. A moldagem é automática, como em

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

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

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

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

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

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

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

### 7.8.6 Funções Matemáticas

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

| | |
|---|---|
| *sin(x)* | seno de $x$, $x$ em radianos |
| *cos(x)* | cosseno de $x$, $x$ em radianos |
| *atn2(y,z)* | arco-tangente de $y/x$, em radianos |
| *exp(x)* | função exponencial $e^x$ |
| *log(x)* | logaritmo natural (base $e$) de $x$ ($x > 0$) |
| *log10(x)* | logaritmo comum (base 10) de $x$ ($x > 0$) |
| *pow(x,y)* | $x^y$ |
| *sqrt(x)* | raiz quadrada de $x$ ($x \geq 0$) |
| *fabs(x)* | valor absoluto de $x$ |

### 7.8.7 Geração de Número Randômico

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

```
#define frand() ((double) rand() / (RAND_MAX+1.0))
```

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

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

**Exercício 7.9** Funções tais como *isupper* podem ser implementadas a fim de economizar espaço ou tempo. Estude estas possibilidades. ■

# 8. A Interface com o Sistema Unix

O sistema operacional **UNIX** fornece seus serviços por meio de um conjunto de chamadas do sistema, que são funções dentro do sistema operacional que que podem ser chamadas por programas usuários. Este capítulo descreve como usar algumas das chamadas de sistema mais importantes para programas em **C**. Se você usa o **UNIX**, isso será diretamente útil, pois às vezes será necessário empregar chamadas do sistema para a obtenção de uma máxima eficiência, ou acessar alguma facilidade que não esteja na biblioteca. No entanto, mesmo que você use **C** em um sistema operacional diferente, você terá um maior conhecimento de C ao estudar este exemplo; muito embora os detalhes possam variar, um código semelhante poderá ser usado em qualquer sistema. Como a biblioteca de **C** da **ANSI** em muitos casos baseia-se nas facilidades do **UNIX**, este código pode ajudar no seu conhecimento da própria biblioteca.

O capítulo está dividido em três partes principais: entrada/saída, sistema de arquivo e alocação de memória. As duas primeiras partes pressupõem uma certa familiaridade com as características externas dos sistemas **UNIX**.

O Capítulo 7 lidou com uma interface de entrada/saída uniforme numa variedade de sistemas operacionais. Em qualquer sistema particular as rotinas da biblioteca-padrão devem ser escritas em termos das facilidades fornecidas pelo sistema hospedeiro. Nas próximas seções, descreveremos as chamadas do sistema **UNIX** para entrada e saída e também mostraremos como partes da biblioteca-padrão podem ser implementadas com elas.

## 8.1 Descritores de Arquivos

No sistema operacional UNIX, toda a entrada e saída é realizada pela leitura ou gravação de arquivos, pois todos os dispositivos periféricos, incluindo o teclado e o vídeo, são arquivos para o sistema de arquivo. Isso significa que uma única interface comum lida com todas as comunicações entre um programa e dispositivos periféricos.

No caso mais geral, antes de ler ou gravar um arquivo, você deve informar ao sistema a sua intenção de fazer isto, um processo chamado abrir um arquivo. Se você vai gravar em um arquivo, pode ser necessário criá-lo ou desconsiderar seu conteúdo anterior. O sistema verifica seu direito de fazer isto (O arquivo existe? Você tem permissão para acessá-lo?), e se tudo estiver bem, ele

retorna ao programa um pequeno inteiro não negativo chamado descritor de arquivo. Sempre que uma entrada ou saída for feita no arquivo, o descritor de arquivo é usado ao invés do nome para identificá-lo. (Um descritor de arquivo é semelhante ao apontador de arquivo usado pela biblioteca-padrão, ou ao manipulador de arquivo do MS-DOS.) Toda a informação sobre um arquivo aberto é mantida pelo sistema; o programa usuário só faz referência ao arquivo por meio do seu descritor.

Como a entrada e saída envolvendo o teclado e o vídeo é tão comum, existem formas especiais para tomar isto conveniente. Quando o interpretador de comandos (o *shell*) roda um programa, três arquivos são abertos com os descritores de arquivo 0, 1 e 2, chamados **entrada-padrão**, **saída-padrão** e **saída-padrão de erro**. Se um programa lê 0 e grava l e 2, ele pode realizar a entrada e saída sem se preocupar em abrir arquivos.

0 usuário de um programa pode redirecionar a E/S de e para arquivos com < e >:

```
prog <arqentra >arqsai
```

Neste caso, o *shell* muda as atribuições *default* para os descritores de arquivo 0 e 1 para os arquivos nomeados. Normalmente, o descritor de arquivo 2 permanece sempre ligado à tela, de forma que as mensagens de erro possam ir para lá. Observações semelhantes permanecem para entrada ou saída associadas a um duto. Em todos os casos, as atribuições de arquivo são alteradas pelo *shell*, e não pelo programa. O programa não sabe de onde vem sua entrada ou para onde vai a saída, desde que use o arquivo 0 para entrada e l e 2 para a saída.

## 8.2 Entrada e Saída de Baixo Nível - Read e Write

A entrada e saída usam as chamadas do sistema *read* e *write*. que são acessadas pelos programas em C por meio de duas funções chamadas *read* e *write*. Para ambas, o primeiro argumento é um descritor de arquivo. O segundo argumento é um vetor de caracteres no seu programa onde os dados vão ou vem. O terceiro argumento é o número de bytes a ser transferido.

```
int n_lidos = read(int fd, char *buf, int n);
int n_gravados = write(int fd, char *buf, int n);
```

Cada chamada retorna um contador do número de bytes realmente transferidos. Na leitura, o número de bytes retornados pode ser menor que o número solicitado. Um valor de retorno de **zero** bytes indica um fim de arquivo, e $-1$ indica um erro de algum tipo. Para a gravação, o valor de retorno é o número de bytes gravados; um erro acontece se este não for igual ao número solicitado.

Qualquer número de bytes pode ser lido ou gravado em uma chamada. Os valores mais comuns são 1, que significa um caractere de cada vez (sem *buffer*), e um número como 1024 ou 4096, que corresponde a um tamanho físico de bloco em um dispositivo periférico. Tamanhos maiores serão mais eficientes porque menos chamadas ao sistema serão feitas.

Juntando esses fatos, podemos escrever um programa simples para copiar sua entrada na saída, o equivalente do programa de cópia de arquivo escrito para o Capítulo 1. O programa copiará qualquer coisa para qualquer coisa, pois a entrada e saída podem ser redirecionadas a qualquer arquivo ou dispositivo.

Programa 8.1: Programa copia.

```
#include <stdio.h>    /* BUFSIZ */
#include <unistd.h>   /* read, write */

int main()  /* copia entrada para saída */
{
  char buf[BUFSIZ];
  int n;
```

```
  while ((n = read(0, buf, BUFSIZ)) > 0)
  write(1, buf, n);
  return 0;
}
```

O parâmetro **BUFSIZ** está definido em *stdio.h*; seu valor é um tamanho adequado ao sistema local. Se o tamanho do arquivo não for um múltiplo de **BUFSIZ**, algum *read* retornará um número de bytes menor a ser gravado por *write*; a próxima chamada a *read* retornará **zero**.

É instrutivo ver como *read* e *write* podem ser usados para construírem rotinas de mais alto nível como *getchar*, *putchar* etc. Por exemplo, segue uma versão de *getchar* que faz entrada sem *buffer*, lendo um caractere de cada vez na entrada-padrão.

Programa 8.2: Função getchar() sem buffer.

```
#include <stdio.h>    /* EOF */
#include <unistd.h>   /* read, write */

/* getchar:  entrada de único caractere sem buffer */
int getchar(void)
{
    char c;
    return (read(0, &c, 1) == 1) ? (unsigned char) c : EOF;
}
```

*c* deve ser um *char*, pois *read* precisa de um ponteiro para caracteres. Moldar *c* para *unsigned char* no comando *return* elimina qualquer problema de extensão de sinal.

A segunda versão de *getchar* realiza entrada com grandes blocos, e envia os caracteres um de cada vez.

Programa 8.3: Função getchar() com buffer.

```
#include <stdio.h>    /* BUFSIZ, EOF */
#include <unistd.h>   /* read, write */

/* getchar:  versão simplificada com buffer */
int getchar(void)
{
    static char buf[BUFSIZ];
    static char* bufp = buf;
    static int n = 0;

    if (n == 0) {  /* buffer vazio */
        n = read(0, buf, sizeof buf);
        bufp = buf;
    }

    return (--n >= 0) ? (unsigned char) * bufp++ : EOF;
}
```

Se estas versões de *getchar* fossem feitas para compilar com <*stdio.h*> incluído em que *getchar* tenha sido implementado como uma macro, seria necessário *#undef* o nome *getchar*.

## 8.3 Open, Creat, Close, Unlink

Para ler ou gravar outros arquivos além da entrada, saída e saída-padrão de erro, você deve abrir explicitamente os arquivos para que possa ler ou gravar. Há duas chamadas do sistema para isto, *open* e *creat* [sic].

*open* é um pouco parecido com *fopen* discutido no Capítulo 7, exceto que ao invés de retornar um ponteiro de arquivo, ela retorna um descritor de arquivo, que é apenas um *int*. *open* retorna $-1$ se houver algum erro.

```
#include <fcntl.h>

int fd;
int open(char *nome, int flags, int perms);

fd = open(name, flags, perms);
```

Assim como em *fopen*, o nome do argumento é uma cadeia de caracteres contendo o nome de arquivo. O segundo argumento *flags*, é um inteiro que especifica como o arquivo deve ser aberto; os principais valores são

**O_RDONLY** abre somente para leitura
**O_WRONLY** abre para gravação
**O_RDWR** abre para leitura e gravação

Estas constantes são definidas em <*fcntl.h*> nos sistemas **System V UNIX**, e em <*sys/fiIe.h*> nas versões Berkeley (**BSD**).

Para abrir um arquivo existente para leitura.

```
fd = open(nome, O_RDONLY, 0);
```

O argumento *perms* é sempre 0 para o uso de *open* conforme discutido aqui.

É um erro tentar abrir um arquivo que não exista. A chamada do sistema *creat* serve para criar novos arquivos, ou para gravar sobre antigos.

```
int creat(char *nome, int perms);

fd = creat(nome, perms);
```

retorna um descritor de arquivo se a função foi capaz de criar o arquivo, e $-1$, caso contrário. Se o arquivo já existe, *creat* o passará para tamanho 0, desprezando assim seu conteúdo anterior; não é um erro criar um arquivo que já existe.

Se o arquivo ainda não existe, *creat* o cria com permissões especificadas pelo argumento *perms*. No sistema de arquivo do **UNIX**, há **nove bits** de informação de permissão associados a um arquivo que controla o acesso de **leitura**, **gravação** e **execução** para o **dono** do arquivo, para o **grupo** do dono e para todos os **outros** usuários. Assim, um número octal de três dígitos é conveniente para especificar as permissões. Por exemplo, 0755 especifica permissão de leitura, escrita e execução para o proprietário, e permissão de leitura e execução para o grupo e todos os demais.

Para ilustrar, segue uma versão simplificada do programa *cp* do **UNIX**, que copia um arquivo para outro. Nossa versão copia somente um arquivo, não permite que o segundo argumento seja um diretório, e cria permissões ao invés de copiá-las.

Programa 8.4: Programa cp.

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h> /* read, write */
#define PERMS 0666  /* lê/grava para dono, grupo, outros */

void erro(char*,  ...);

/* cp:  copia f1 para f2 */
int main(int argc, char* argv[])
{
```

```
    int f1, f2, n;
    char buf[BUFSIZ];

    if (argc != 3) {
        erro("Uso: cp arquivo_origem arquivo_destino");
    }

    if ((f1 = open(argv[1], O_RDONLY, 0)) == -1) {
        erro("cp: não posso abrir %s", argv[1]);
    }

    if ((f2 = creat(argv[2], PERMS)) == -1)
        erro("cp: não posso criar %s, modo %03o",
             argv[2], PERMS);

    while ((n = read(f1, buf, BUFSIZ)) > 0)
        if (write(f2, buf, n) != n) {
            erro("cp: erro ao gravar arquivo %s", argv[2]);
        }

    return 0;
}
```

Este programa cria o arquivo de saída com permissões fixas de 0666. Com a chamada do sistema *stat*, descrita na Seção 8.6, podemos determinar o modo de um arquivo existente e dar o mesmo modo a sua cópia.

Observe que a função erro é chamada com listas de argumento variáveis, assim como *printf*. A implementação de erro ilustra como usar um outro membro da família *printf*. A função da biblioteca-padrão *vprintf* funciona como *printf*, exceto que a lista variável de argumentos é substituída por um único argumento que foi inicializado chamando-se a macro *va_start*. Semelhantemente, *vfprintf* e *vsprintf* correspondem a *fprintf* e *sprintf*.

Programa 8.5: Programa cp.

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

/* erro:  imprime uma mensagem de erro e termina */
void erro(char* fmt, ...)
{
    va_list args;
    va_start(args, fmt);
    fprintf(stderr, "erro: ");
    vfprintf(stderr, fmt, args);
    fprintf(stderr, "\n");
    va_end(args);
    exit(1);
}
```

Existe um limite (normalmente 20) para o número de arquivos que um programa pode manter abertos simultaneamente. Em consequência, qualquer programa que pretende processar muitos arquivos deve estar preparado para reutilizar descritores de arquivo. A função *close(int fd)* termina a conexão entre um descritor de arquivo e um arquivo aberto, liberando o descritor de arquivo para uso com algum outro arquivo; ela corresponde a *fclose* na biblioteca-padrão, exceto que não existe um *buffer* para esvaziar. O término do programa por meio de *exit* ou o retorno do programa principal fecha todos os arquivos abertos.

A função *unlink(char *nome)* remove o arquivo *nome* do sistema de arquivos. Ela corresponde à função *remove* da biblioteca-padrão.

**Exercício 8.1** Reescreva o programa *cat* do Capítulo 7 usando *read*, *write*, *open* e *close* no lugar dos seus equivalentes na biblioteca-padrão. Execute testes para determinar as velocidades relativas das duas versões. ■

## 8.4 Acesso Randômico - Lseek

A entrada e saída de arquivos é normalmente sequencial: cada *read* ou *write* acessa a posição seguinte à anterior. Quando necessário, entretanto, um arquivo pode ser lido ou gravado em qualquer ordem arbitrária. A chamada do sistema *lseek* fornece uma maneira de se posicionar dentro de um arquivo sem leitura ou gravação:

```
long lseek(int da, long deslocamento, int origem);
```

força a posição corrente do arquivo cujo descritor é *da*, a se mover para a posição *deslocamento*, o qual é relativo à posição especificada por *origem*. Leituras ou gravações subsequentes começarão nesta posição. *origem* pode ser 0, 1, ou 2 para especificar que *deslocamento* deve ser medido desde o **início**, desde a posição **corrente** ou desde o **final** do arquivo, respectivamente. Por exemplo, para anexar a um arquivo (a redireção >> no *shell* do **UNIX**, ou "*a*" para *fopen*), posicione-se ao final antes de gravar no arquivo:

```
lseek(da, 0L, 2);
```

Para voltar ao início ("rebobinar").

```
lseek(da, 0L, 0);
```

Observe o argumento 0*L*; ele também poderia ser escrito por `(long) 0` ou apenas 0 se *lseek* fosse declarado apropriadamente.

Com *lseek*, é possvel tratar os arquivos mais ou menos como grandes vetores, ao custo de um acesso mais lento. Por exemplo, a seguinte função lê qualquer número de bytes a partir de qualquer lugar arbitrário em um arquivo. Ela retorna o número de bytes lidos, ou $-1$ se houver erro.

Programa 8.6: Programa get.

```
#include <unistd.h>   /* read, write */

/*get:  lê n bytes a partir da posição pos */
int get(int da, long pos, char* buf, int n)
{
    if (lseek(da, pos, 0) >= 0) { /* posiciona-se */
        return read(da, buf, n);
    } else {
        return -1;
    }
}
```

O valor de retorno de *lseek* é um *long* que indica a nova posição no arquivo, ou $-1$ se houver um erro. A função da biblioteca-padrão *fseek* é semelhante a *lseek*, exceto que o primeiro argumento é um *FILE* * e o retorno é diferente de zero, se houver um erro.

## 8.5 Exemplo - Uma Implementação de Fopen e Getc

Vamos ilustrar como algumas dessas peças se juntam mostrando uma implementação das rotinas da biblioteca-padrão *fopen* e *getc*.

Lembre-se que arquivos na biblioteca-padrão são descritos por ponteiros ao invés de descritores de arquivo. Um ponteiro de arquivo é um ponteiro para uma estrutura que contém diversas informações sobre um arquivo: um ponteiro para um *buffer*, de forma que o arquivo possa ser lido em grandes partes; um contador do número de caracteres deixados no *buffer*; um ponteiro para a próxima posição de caractere no *buffer*: o descritor de arquivo; e sinalizadores descrevendo o modo de leitura/gravação, estado de erro, etc.

A estrutura de dados que descreve um arquivo está contida em <*stdio.h*>, que deve ser incluído (por *#include*) em qualquer arquivo-fonte que use rotinas da biblioteca de **entrada/saída-padrão**. Ela também é incluída em funções nessa biblioteca. No seguinte trecho de um <*stdio.h*> típico, os nomes que devem ser usados apenas por funções da biblioteca começam com um sinal de sublinhado, a fim de que não entrem em conflito com nomes no programa de um usuário.

Esta convenção é usada por todas as rotinas da biblioteca-padrão.

Programa 8.7: Trecho de um stdio.h típico.

```
#define NULL      0
#define EOF       (-1)
#define BUFSIZ    1024
#define FOPEN_MAX  20    /* número max de arquivos abertos simultaneamente */

typedef struct _iobuf {
    int  cnt;       /* caracteres deixados */
    char* ptr;      /* posição prox. caractere */
    char* base;     /* local do buffer */
    int  flag;      /* modo de acesso do arquivo */
    int  fd;        /* descritor do arquivo */
} FILE;
extern FILE _iob[FOPEN_MAX];

#define stdin   (&_iob[0])
#define stdout  (&_iob[1])
#define stderr  (&_iob[2])

enum _flags {
    _READ   = 01,   /* arquivo aberto para leitura */
    _WRITE  = 02,   /* arquivo aberto para escrita */
    _UNBUF  = 04,   /* arquivo sem buffer */
    _EOF    = 010,  /* EOF ocorreu neste arquivo */
    _ERR    = 020   /* erro ocorreu neste arquivo */
};

int _fillbuf(FILE*);
int _flushbuf(int, FILE*);

#define feof(p)     ((p)->flag & _EOF) != 0)
#define ferror(p)   ((p)->flag & _ERR) != 0)
#define fileno(p)   ((p)->fd)

#define getc(p)   (--(p)->cnt >= 0 \
                  ? (unsigned char) *(p)->ptr++ : _fillbuf(p))
#define putc(x,p) (--(p)->cnt >= 0 \
                  ? *(p)->ptr++ = (x) : _flushbuf((x),p))

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

A macro *getc* normalmente decrementa o contador, avança o apontador, e retorna o caractere. (Lembre-se de que um *#define* longo é continuado com uma **contrabarra**.) Se o contador ficar negativo, entretanto, *getc* chama a função *_fillbuf* para preencher o *buffer*, reinicializar o conteúdo da estrutura e retornar um caractere. Os caracteres são retornados *unsigned*, fazendo com que nunca possam ser negativos.

Embora não discutamos quaisquer detalhes, incluímos a definição de *putc* para mostrar que ela opera de forma muito semelhante a *getc*, chamando uma função *_flushbuf* quando seu *buffer* estiver cheio. Também incluímos macros para acessar o estado de erro e fim-de-arquivo e o descritor de arquivo.

A função *fopen* pode agora ser escrita. A maior parte de *fopen* tem a ver com a abertura de arquivo e com o posicionamento no local correto, e com a atualização dos sinalizadores para indicar o estado correto. *fopen* não aloca espaço em *buffer*; isso é feito por *_fillbuf* quando o arquivo é lido inicialmente.

Programa 8.8: Função fopen().

```
#include <fcntl.h>
#include <unistd.h>   /* read, write */
#define PERMS 0666    /* lê/grava para dono, grupo e outros */

/* fopen: abre arquivo, retorna apontador de arquivo */
FILE* fopen(char* nome, char* modo)
{
    int fd;
    FILE* fp;

    if (*modo != 'r' && *modo != 'w' && *modo != 'a') {
        return NULL;
    }

    for (fp = _iob; fp < _iob + FOPEN_MAX; fp++)
        if ((fp->flag & (_READ | _WRITE)) == 0) {
            break;    /* encontrou slot vazio */
        }

    if (fp >= _iob + FOPEN_MAX) { /* sem slot vazio  */
        return NULL;
    }

    if (*modo == 'w') {
        fd = creat(nome, PERMS);
    } else if (*modo == 'a') {
        if ((fd = open(nome, O_WRONLY, 0)) == -1) {
            fd = creat(nome, PERMS);
        }

        lseek(fd, 0L, 2);
    } else {
        fd = open(nome, O_RDONLY, 0);
    }

    if (fd == -1) {       /* sem acesso a nome */
        return NULL;
    }
```

```
    fp->fd = fd;
    fp->cnt = 0;
    fp->base = NULL;
    fp->flag = (*modo == 'r') ? _READ : _WRITE;
    return fp;
}
```

Esta versão de *fopen* não lida com todas as possibilidades de modo de acesso do padrão, embora sua inclusão não ocupe muito código. Em particular, nosso *fopen* não reconhece o "*b*" que sinaliza acesso binário, pois isso não tem significado para os sistemas **UNIX**, e nem o "+" que permite tanto leitura quanto gravação em arquivo.

A primeira chamada a *getc* para um arquivo particular encontra um contador zero, que força uma chamada a *_fillbuf*. Se *_fillbuf* descobre que o arquivo não está aberto para leitura, ele retorna *EOF* imediatamente. Caso contrário, ele tenta alocar um *buffer* (se a leitura tiver que ser *bufferizada*).

Uma vez estabelecido o *buffer*, *_fillbuf* chama *read* para preenchê-lo, atualiza contador e ponteiros, e retorna o caractere no início do *buffer*. As chamadas subsequentes a *_fillbuf* encontrarão um *buffer* alocado.

Programa 8.9: Função _fillbuf().

```
#include <stdlib.h>

/* _fillbuf:  aloca e preenche buffer da entrada */
int _fillbuf(FILE* fp)
{
    int bufsize;

    if ((fp->flag & (_READ | _EOF | _ERR)) != _READ) {
        return EOF;
    }

    bufsize = (fp->flag & _UNBUF) ? 1 : BUFSIZ;

    if (fp->base == NULL)     /* nenhum buffer ainda */
        if ((fp->base = (char*) malloc(bufsize)) == NULL) {
            return EOF;    /* não pode obter buffer */
        }

    fp->ptr = fp->base;
    fp->cnt = read(fp->fd, fp->ptr, bufsize);

    if (--fp->cnt < 0) {
        if (fp->cnt == -1) {
            fp->flag |= _EOF;
        } else {
            fp->flag |= _ERR;
        }

        fp->cnt = 0;
        return EOF;
    }

    return (unsigned char) * fp->ptr++;
}
```

O que falta agora é saber como tudo se inicia. O vetor *_iob* deve ser definido e inicializado para stdin, stdout e stderr:

Programa 8.10: Função _iob().

```
FILE _iob[FOPEN_MAX] = {    /* stdin, stdout, stderr */
    { 0, (char*) 0, (char*) 0, _READ, 0 },
    { 0, (char*) 0, (char*) 0, _WRITE, 1 },
    { 0, (char*) 0, (char*) 0, _WRITE | _UNBUF, 2 }
};
```

A inicialização dos sinalizadores da estrutura mostra que *stdin* deve ser lido, *stdout* deve ser gravado e *stderr* deve ser gravado sem *buffer*.

**Exercício 8.2** Reescreva *fopen* e *_fillbuf* com campos ao invés de operações explícitas com bits. Compare o tamanho do código e velocidade de execução. ■

**Exercício 8.3** Projete e escreva as rotinas *_flushbuf*, *fflush* e *fclose*. ■

**Exercício 8.4** A função da biblioteca-padrão

```
int fseak (fiLE *fp, long deslocamento, int origem)
```

é idêntica a *lseek*, exceto que *fp* é um apontador de arquivo ao invés de um descritor de arquivo, e o valor de retorno é um estado *int*, e não uma posição. Escreva *fseek*. Assegure-se de que seu *fseek* esteja coordenado com o tipo de *buffer* (ou ausência de tal) usado pelas outras funções da biblioteca. ■

## 8.6 Exemplo - Listagem de Diretórios

Às vezes, um tipo diferente de interação com o sistema de arquivos é necessário – determinando a informação sobre o arquivo, e não o que ele contém. Um programa de listagem de diretório como o comando *ls* do **UNIX** é um exemplo – ele imprime os nomes dos arquivos em um diretório e, opcionalmente, outras informações, como tamanhos, permissões, e assim por diante. O comando *dir* do **MS-DOS** é semelhante.

Como um diretório no **UNIX** é apenas um arquivo, *ls* só precisa ler esse arquivo para obtemos nomes dos arquivos. Mas é necessário usar uma chamada do sistema para acessar outras informações sobre o arquivo, como seu tamanho. Em outros sistemas, pode-se precisar de uma chamada do sistema até mesmo para acessar os nomes dos arquivos; este é o caso, por exemplo, no **MS-DOS**. O que queremos é dar acesso à informação de uma forma relativamente independente do sistema, muito embora a implementação possa ser muito dependente do sistema.

Ilustraremos parte disto escrevendo um programa chamado *tamarq*. *tamarq* é uma forma especial de *ls* que imprime os tamanhos de todos os arquivos indicados pelos argumentos na linha de comando. Se um desses arquivos for um diretório, *tamarq* aplica-se recursivamente a si mesmo para este diretório. Se não houver argumentos, ele processa o diretório corrente.

Para iniciar, uma breve revisão da estrutura do sistema de arquivos do **UNIX**. Um diretório é um arquivo que contém uma lista de nomes-de-arquivo e alguma indicação de onde estão localizados. O "*local*" é um índice para uma outra tabela chamada "*tabela de inodes*". O *inode* para um arquivo é onde toda a informação sobre o arquivo (exceto seu nome) é mantida. Uma entrada de diretório geralmente consiste em somente dois itens, o nome do arquivo e um número de *inode*.

Infelizmente, o formato e conteúdo exato de um diretório não são idênticos para todas as versões do sistema. Assim, dividiremos a tarefa em duas partes para tentar isolar as partes não

portatéis. O nível mais externo define uma estrutura chamada *Dirent* e três rotinas *opendir*, *readdir* e *closedir* para dar acesso independente do sistema ao nome e número de *inode* em uma entrada de diretório. Escreveremos *tamarq* com esta interface. Depois mostraremos como implementar estes em sistemas que usam a mesma estrutura de diretório da **Versão 7** e **System V UNIX**; as variantes são deixadas como exercícios.

A estrututra *Dirent* contém o número de *inode* e o nome. O tamanho máximo de um nome-de-arquivo componente é *NAME_MAX*, que é um valor dependente do sistema. *opendir* retorna um apontador para uma estrutura chamada *DIR*, semelhante a *FILE*, que é usada por *readdir* e closedir. Esta informação é coletada em um arquivo chamado *dirent.h*.

Programa 8.11: Arquivo dirent.h.

```
#define NAME_MAX   14 /* maior nome de arquivo; */
                      /* dependente do sistema */

typedef struct {             /* entrada portátil do diretório */
    long ino;                /* número de inode */
    char name[NAME_MAX + 1]; /* nome + término '\0' */
} Dirent;

typedef struct {      /* DIR mínimo: sem buffer, etc. */
    int fd;           /* descritor de arquivo p/ diretório */
    Dirent d;         /* entrada de diretório */
} DIR;

DIR* opendir(char* nomedir);
Dirent* readdir(DIR* dfd);
void closedir(DIR* dfd);
```

A chamada do sistema *stat* obtém um nome de arquivo e retorna toda a informação no *inode* para esse arquivo, ou $-1$ se houver um erro. Ou seja,

```
char *nome;
struct stat stbuf;
int stat(char *, struct stat *);

stat(nome, &stbuf);
```

preenche a estrutura *stat* com a informação de *inode* para o nome do arquivo. A estrutura descrevendo o valor retornado por *stat* está em <*sys/stat.h*>, e normalmente se parece assim:

```
struct stat   /* informação do inode retornada por stat */
{
  dev_t     st_dev;      /* dispositivo do inode */
  ino_t     st_ino;      /* número do inode */
  short     st_mode;     /* bits de modo */
  short     st_nlink;    /* número de ligações ao arquivo */
  short     st_uid;      /* id. do usuário do dono */
  short     st_gid;      /* id. do grupo do dono */
  dev_t     st_rdev;     /* para arquivos especiais */
  off_t     st_size;     /* tamanho arq. em caracteres */
  time_t    st_atime;    /* hora do último acesso */
  time_t    st_mtime;    /* hora da última modificação */
  time_t    st_ctime;    /* thora da criação original */
};
```

A maioria desses valores são explicados pelos campos de comentário. Os tipos como *dev_t* e *ino_t* são definidos em <*sys/types.h*>, que também deve ser incluído.

A entrada *st_mode* contém um conjunto de sinalizadores descrevendo o arquivo. As definições de sinalizador também estão incluídas em <*sys/types.h*>; precisamos somente da parte que lida com o tipo de arquivo:

```
#define S_IFMT    0160000  /* tipo de arquivo:  */
#define S_IFDIR   0040000  /* diretório */
#define S_IFCHR   0020000  /* caractere especial */
#define S_IFBLK   0060000  /* bloco especial */
#define S_IFREG   0010000  /* normal */
/* ... */
```

Agora estamos prontos para escrever o programa *tamarq*. Se o modo obtido por *stat* indicar que um arquivo não é um diretório, então o tamanho está disponível e pode ser impresso diretamente. Se o arquivo for um diretório, entretanto, então temos que processar este diretório com um arquivo de cada vez; ele pode, por sua vez, conter subdiretórios, de forma que o processo é recursivo.

A rotina principal lida com os argumentos da linha de comando; ela lida com cada argumento para a função *tamarq*.

Programa 8.12: Rotina principal do programa tamarq.

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>      /* flags para ler e gravar */
#include <sys/types.h>  /* typedefs */
#include <sys/stat.h>   /* estrutura retornada por stat */
#include "dirent.h"

void tamarq(char*);

/* imprime tamanhos de arquivos */
int main(int argc, char** argv)
{
    if (argc == 1) { /* default: diretório corrente */
        tamarq(".");
    } else
        while (--argc > 0) {
            tamarq(*++argv);
        }

    return 0;
}
```

**Adendo 8.6.1 — Como compilar tamarq..** O programa *tamarq* implementado aqui deve ser compilado sem a opção **-std=c99** ou **-std=c11**. Algumas macros não estão disponíveis nas versões mais avançadas de **C**, e assim o programa não vai compilar sem erros nessas versões. Um bom exercício é modificar o programa *tamarq* para funcionar com as opções de **c99** e **c11** habilitadas.

A função *tamarq* imprime o tamanho do arquivo. Se o arquivo for um diretório, *tamarq* primeiro chama diretório para manipular todos os arquivos dentro dele. Observe como os nomes

de sinalizadores *S_IFMT* e *S_IFDIR* de <*sys/stat.h*> são usados para decidir se o arquivo é um diretório. Os parênteses são importantes, pois a precedência de & é menor que a de ==.

Programa 8.13: Função tamarq().

```
void diretorio(char*, void (*fcn)(char*));

/* tamarq:  imprime tamanho de arquivo "nome" */
void tamarq(char* nome)
{
    struct stat stbuf;

    if (stat(nome, &stbuf) == -1) {
        fprintf(stderr, "tamarq: não posso acessar %s\n", nome);
        return;
    }

    if ((stbuf.st_mode & S_IFMT) == S_IFDIR) {
        diretorio(nome, tamarq);
    }

    printf("%8ld %s\n", stbuf.st_size, nome);
}
```

A função *diretorio* é uma rotina geral que aplica uma função para cada arquivo em um diretório. Ela abre o diretório, percorre seus arquivos em um laço, chamando a função para cada um, e depois fecha o diretório e retorna. Como *tamarq* chama *diretorio* em cada novo diretório, as funções chamam uma à outra recursivamente.

Programa 8.14: Função diretorio().

```
#define MAX_PATH 1024

/* diretorio:  aplica fcn a todos os arquivos no dir */
void diretorio(char* dir, void (*fcn)(char*))
{
    char nome[MAX_PATH];
    Dirent* dp;
    DIR* dfd;

    if ((dfd = opendir(dir)) == NULL) {
        fprintf(stderr, "diretorio: não posso abrir %s\n", dir);
        return;
    }

    while ((dp = readdir(dfd)) != NULL) {
        if (strcmp(dp->nome, ".") == 0
                || strcmp(dp->nome, "..")) {
            continue;    /* salta si mesmo e pai */
        }

        if (strlen(dir) + strlen(dp->nome) + 2 > sizeof(nome))
            fprintf(stderr, "diretorio: nome %s %s muito longo\n",
                    dir, dp->nome);
        else {
            sprintf(nome, "%s/%s", dir, dp->nome);
            (*fcn)(nome);
```

```
        }
    }

    closedir(dfd);
}
```

Cada chamada a *readdir* retorna um apontador para a informação para o próximo arquivo, ou *NULL* quando não houver mais arquivos. Cada diretório sempre contém entradas para si mesmo, chamada "." e para seu pai, "..", estas devem ser saltadas, ou então o programa entrará em um laço sem fim.

Neste nível, o código é independente de como os diretórios são formatados. O próximo passo é apresentar versões mínimas de *opendir*, *readdir* e *closedir* para um sistema específico. As rotinas a seguir são para a **Versão 7** e o **System V UNIX**; elas usam a informação de diretório no arquivo cabeçalho <*sys/dir.h*>, que é a seguinte:

Programa 8.15: Função diretorio().

```
#ifndef DIRSIZ
#define DIRSIZ  14
#endif
struct direct {           /* entrada de diretório */
    ino_t d_ino;          /* número de inode */
    char  d_name[DIRSIZ]; /* nome longo name não tem '\0' */
};
```

Algumas versões do sistema permitem nomes muito maiores e possuem uma estrutura de diretório mais complicada.

O tipo *ino_t* é um *typedef* que descreve o índice para a tabela de *inodes*. No sistema que usamos regularmente, ele é *unsigned short*, mas este não é o tipo de informação a ser embutida em um programa: ele pode ser diferente em sistemas diferentes, assim, o *typedef* é melhor. Um conjunto completo de tipos do "sistema" é encontrado em <*sys/types.h*>.

*opendir* abre o diretório, verifica se o arquivo é um diretório (desta vez pela chamada do sistema *fstat*, que se parece com *stat*, mas aplica-se a um descritor de arquivo), aloca uma estrutura de diretório e grava a informação:

Programa 8.16: Função opendir().

```
#include <stdlib.h>
int fstat(int fd, struct stat*);

/* opendir:  abre um diretório para chamadas a reddir */
DIR* opendir(char* nomedir)
{
    int fd;
    struct stat stbuf;
    DIR* dp;

    if ((fd = open(nomedir, O_RDONLY, 0)) == -1
            || fstat(fd, &stbuf) == -1
            || (stbuf.st_mode & S_IFMT) != S_IFDIR
            || (dp = (DIR*) malloc(sizeof(DIR))) == NULL) {
        return NULL;
    }

    dp->fd = fd;
    return dp;
```

```
}
```

*closedir* fecha o arquivo de diretório e libera o espaço:

Programa 8.17: Função closedir().

```
/* closedir:  fecha diretório aberto por opendir */
void closedir(DIR* dp)
{
    if (dp) {
        close(dp->fd);
        free(dp);
    }
}
```

Finalmente, *readdir* usa *read* para ler cada entrada de diretório. Se um local do diretório não estiver atualmente em uso (porque um arquivo foi removido), o número de *inode* é zero, e esta posição é saltada. Caso contrário, o número e nome de *inode* são colocados em uma estrutura *static* e um apontador para isso é retornado ao usuário. Cada chamada substitui a informação da chamada anterior.

Programa 8.18: Função readdir().

```
/* readdir:  lê entradas do diretório em sequência */
Dirent* readdir(DIR* dp)
{
    struct direct dirbuf;  /* estrutura de diretório local */
    static Dirent  d;      /* retorna: estrutura portátil */

    while (read(dp->fd, (char*) &dirbuf, sizeof(dirbuf))
            == sizeof(dirbuf)) {
        if (dirbuf.d_ino == 0) { /* entrada não usada */
            continue;
        }

        d.ino = dirbuf.d_ino;
        strncpy(d.nome, dirbuf.d_name, DIRSIZ);
        d.nome[DIRSIZ] = '\0';  /* assegura término */
        return &d;
    }

    return NULL;
}
```

Embora o programa *tamarq* seja especializado, ele ilustra algumas ideias importantes. Primeiro, muitos programas não são "programas do sistema"; ele simplesmente usa a informação mantida pelo sistema operacional. Para esses programas, é muito importante que a representação da informação apareça somente nos *cabeçalhos-padrão*, e que os programas incluam esses arquivos ao invés de embutirem as declarações neles mesmos. A segunda observação é que, com cuidado, é possível criar uma interface para objetos dependentes do sistema que seja por si mesma relativamente independente do sistema. As funções da biblioteca-padrão são bons exemplos.

**Exercício 8.5** Modifique o programa *tamarq* para imprimir as outras informações contidas na entrada do *inode*. ■

## 8.7 Exemplo - Um Alocador de Memória

No **Capítulo 5**, apresentamos um alocador de memória muito limitado orientado para a pilha. A versão que escreveremos agora não tem restrição. As chamadas *aloca* e *libera* podem ocorrer em qualquer ordem; *aloca* chama o sistema operacional para obter mais memória quando for necessário. Estas rotinas ilustram algumas das considerações envolvidas na escrita de código dependente da máquina em uma forma relativamente independente da máquina, e também mostram aplicações da vida real com estruturas, uniões e *typedef*.

Ao invés de alocar espaço a partir de um vetor com tamanho fixado em tempo de compilação, *aloca* requisitará espaço ao sistema operacional quando for necessário. Como outras atividades no programa também podem solicitar espaço sem chamar este alocador, o espaço gerenciado por *aloca* **pode não ser contíguo**. Assim, sua memória livre é mantida sob a forma de uma lista de blocos livres. Cada bloco contém um tamanho, um apontador para o próximo bloco, e o próprio espaço. Os blocos são mantidos em ordem crescente de endereço, e o último bloco (o endereço mais alto) aponta para o primeiro.

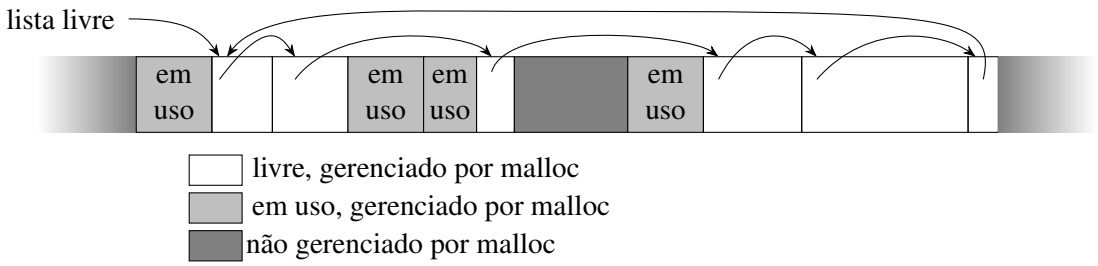



Figura 8.1: Gerenciamento de memória com *malloc*.

Quando é feito um pedido, a lista livre é analisada até que um bloco com tamanho suficiente seja encontrado. Este algoritmo é chamado "primeiro que serve", ao contrário de "melhor que serve", que procura o menor bloco que satisfaça o pedido. Se o bloco for exatamente do tamanho solicitado, ele é desligado da lista e retornado ao usuário. Se o bloco for muito grande, ele é dividido, e a quantidade exata é retornada ao usuário enquanto o resíduo permanece na lista livre. Se nenhum bloco de tamanho suficiente for encontrado, outro grande pedaço é obtido pelo sistema operacional e ligado à lista livre.

A liberação também gera uma busca da lista livre, a fim de achar o local apropriado para inserir o bloco sendo liberado. Se o bloco sendo liberado for adjacente a um bloco livre de qualquer lado, ele é unido a ele formando um único grande bloco, de forma que o armazenamento não fique muito fragmentado. Determinar a adjacência é fácil, porque a lista livre é mantida em ordem crescente de endereços.

Um problema, do qual falamos no **Capítulo 5**, é assegurar que a memória retornada por *aloca* esteja alinhada corretamente para os objetos que serão armazenados nela. Embora as máquinas variem, para cada uma existe um tipo mais restritivo: se o tipo mais restritivo puder ser armazenado em um endereço particular, todos os outros tipos também podem ser. Em algumas máquinas, o tipo mais restritivo é *double*; em outras, *int* ou *long* é suficiente.

Um bloco livre contém um ponteiro para o próximo bloco na cadeia, um registro do tamanho do bloco, e em seguida o próprio espaço livre; a informação de controle no início é chamada "cabeçalho". Para simplificar o alinhamento, todos os blocos são múltiplos do tamanho do cabeçalho, sendo que este é alinhado corretamente. Isto é feito por uma união que contém a estrutura de cabeçalho desejada e uma ocorrência do tipo com alinhamento mais restritivo, que escolhemos arbitrariamente como sendo *long*:

Programa 8.19: Cabeçalho para alinhamento da alocação de memória.

```
// long é escolhido como uma instância de um tipo de alinhamento mais restritivo
typedef long Alinha; // para alinhamento dos limites

union cabecalho {    // cabeçalho do bloco:
    struct {
        union cabecalho* prox; // prox.  bloco na lista livre
        unsigned tamanho;      // tamanho deste bloco
    } s;

    Alinha x;                  // força alinhamento dos blocos
};
typedef union cabecalho Cabecalho;
```

O campo *Alinha* nunca é usado; ele só força cada cabeçalho a ser alinhado em um limite do pior dos casos.

Em *aloca*, o tamanho pedido em caracteres é arredondado para cima para unidades de tamanho do cabeçalho; o bloco que será alocado contém uma unidade a mais que o calculado, para o próprio cabeçalho, e este é o valor registrado no campo de tamanho do cabeçalho. O ponteiro retornado por *aloca* aponta para o espaço livre, e não para o próprio cabeçalho. O usuário pode fazer qualquer coisa com o espaço solicitado, mas se algo for escrito fora do espaço alocado, a lista provavelmente ficará confusa.

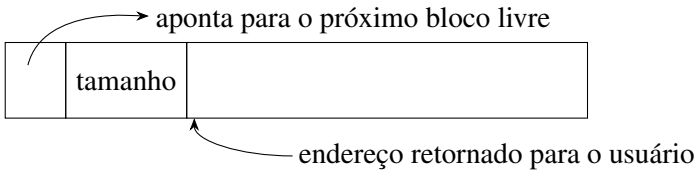



Figura 8.2: Um bloco retornado por *malloc*.

O campo de *tamanho* é necessário porque os blocos controlados por *aloca* não precisam ser contíguos – não podemos calcular tamanhos pela aritmética com ponteiros.

A variável *base* é usada para dar início. Se *plivre* for *NULL*, como na primeira chamada a *aloca*, então é criada uma lista livre degenerada; ela contém um bloco de tamanho zero, e aponta para si mesma. De qualquer forma, a lista livre é então pesquisada. A procura pelo bloco livre de tamanho adequado começa no ponto (*plivre*) onde o último bloco foi achado; esta estratégia ajuda a manter a lista homogênea. Se um bloco muito grande for achado, sua parte final é retornada ao usuário; dessa forma, o cabeçalho original só precisa ter o tamanho ajustado. Em todos os casos, o ponteiro retornado ao usuário aponta para o espaço livre dentro do bloco, que começa uma unidade além do cabeçalho.

Programa 8.20: Função aloca().

```
static Cabecalho base = {0};      // lista vazia para iniciar
static Cabecalho* plivre = NULL;  // início da lista livre
static Cabecalho* maismem(unsigned nblocks);

void* aloca(unsigned nbytes)
{
    Cabecalho* p_atual;
    Cabecalho* p_ant;
```

```
    unsigned n_unidades;
    n_unidades = ((nbytes + sizeof(Cabecalho) - 1) / sizeof(Cabecalho)) + 1;

    if (plivre == NULL) { // nenhuma lista livre ainda.
        base.s.prox = &base;
        base.s.tamanho = 0;
        plivre = &base;
    }

    p_ant = plivre;
    p_atual = p_ant->s.prox;

    for (; ; p_ant = p_atual, p_atual = p_atual->s.prox) {
        if (p_atual->s.tamanho >= n_unidades) { // tem tamanho
            if (p_atual->s.tamanho == n_unidades) { // exato
                p_ant->s.prox = p_atual->s.prox;
            } else { // aloca parte final
                p_atual->s.tamanho -= n_unidades;
                p_atual += p_atual->s.tamanho;
                p_atual->s.tamanho = n_unidades;
            }

            plivre = p_ant;
            return (void*) (p_atual + 1);
        }

        if (p_atual == plivre) { // voltou ao inicio da lista
            if ((p_atual = maismem(n_unidades)) == NULL) {
                return NULL; // sem memória para alocar
            }
        }
    }
}
```

A função *maismem* obtém memória do sistema operacional. Os detalhes de como isso é feito variam de um sistema para outro. Como pedir memória ao sistema é uma operação comparativamente dispendiosa, não desejamos fazê-la em toda chamada a *aloca*, e, portanto, *maismem* solicita pelo menos *NALOCA* unidades; este bloco maior será dividido conforme a necessidade. Após definir o campo de tamanho, *maismem* insere a memória adicional em cena chamando *libera*.

O chamada do sistema **UNIX**, *sbrk(n)* retorna um ponteiro para *n* mais bytes de memória. *ebrk* retorna $-1$ se não há espaço, muito embora *NULL* tivesse sido uma escolha melhor. O valor $-1$ deve ser convertido para *char* * para que possa ser comparado com o valor de retorno. Novamente, moldes tomam a função relativamente imune aos detalhes da representação de ponteiro em diferentes máquinas. Há ainda uma suposição, entretanto, de que os ponteiros para diferentes blocos retornados por *sbrk* podem ser comparados de maneira significativa. Isso não é garantido pelo padrão, que só permite comparações de ponteiros dentro de um vetor. Assim, esta versão de *aloca* só é portátil entre máquinas para as quais a comparação geral de ponteiro é significativa.

Programa 8.21: Função maismem().

```
#define ALOCAN 1024  // unidades mínimas a alocar

static Cabecalho* maismem(unsigned n_unidades)
{
    void* memlivre;     // endereço na memória recém-criada
    Cabecalho* p_inserido;
```

```
    if (n_unidades < ALOCAN) {
        n_unidades = ALOCAN;
    }

    memlivre = sbrk(n_unidades * sizeof(Cabecalho));

    if (memlivre == (void*) - 1) {
        return NULL; // impossível alocar, sbrk retorna -1
    }

    p_inserido = (Cabecalho*) memlivre;
    p_inserido->s.tamanho = n_unidades;
    libera((void*) (p_inserido + 1)); // insere na lista livre
    return plivre;
}
```

*libera* é a última parte. Ela simplesmente pesquisa uma lista livre, partindo de *plivre*, procurando o local para inserir o bloco livre. Isto acontecerá entre dois blocos existentes ou em uma das extremidades da lista. De qualquer forma, se o bloco a ser liberado é adjacente a um dos vizinhos, os blocos adjacentes são combinados. O único problema é o de manter os ponteiros apontando para as coisas certas e manter os tamanhos corretos.

Programa 8.22: Função libera().

```
/* libera:  coloca o bloco ptr na lista livre */
void libera(void* ptr)
{
    Cabecalho* p_inserido, *p_atual;
    p_inserido = ((Cabecalho*) ptr) - 1; // aponta para o cabeçalho

    for (p_atual = plivre; !((p_atual < p_inserido)
                             && (p_inserido < p_atual->s.prox));
            p_atual = p_atual->s.prox) {
        if ((p_atual >= p_atual->s.prox) && ((p_atual < p_inserido)
                                             || (p_inserido < p_atual->s.prox)))
    {
            break; // liberou de um lado ou de outro
        }
    }

    if ((p_inserido + p_inserido->s.tamanho) == p_atual->s.prox) {
        p_inserido->s.tamanho += p_atual->s.prox->s.tamanho;
        p_inserido->s.prox = p_atual->s.prox->s.prox;
    } else {
        p_inserido->s.prox = p_atual->s.prox;
    }

    if ((p_atual + p_atual->s.tamanho) == p_inserido) {
        p_atual->s.tamanho += p_inserido->s.tamanho;
        p_atual->s.prox = p_inserido->s.prox;
    } else {
        p_atual->s.prox = p_inserido;
    }

    plivre = p_atual;
}
```

Embora a alocação de memória seja intrinsecamente dependente de máquina, o código mostrado aqui ilustra como as dependências da máquina podem ser controladas e confinadas a uma parte muito pequena do programa. O uso de *typedef* e *union* controla o alinhamento (dado que *sbrk* fornece um ponteiro apropriado). Moldes explicam as conversões de ponteiros e até resolvem o problema de uma interface de sistema mal projetada. Embora os detalhes aqui relatados sejam relacionados com a alocação de memória, o enfoque geral é aplicável a outras situações também.

**Exercício 8.6** A função *calloc(n, tam)* da biblioteca-padrão retorna um apontador para *n* objetos de tamanho *tam*, inicializados com zero. Escreva *calloc* usando *aloca* como modelo ou como uma função a ser chamada. ■

**Exercício 8.7** *aloca* aceita um pedido sem verificar a validade do tamanho pedido; *libera* acredita que o bloco a ser liberado tem um campo de tamanho válido. Melhore estas rotinas para que façam uma verificação de erros mais completa. ■

**Exercício 8.8** Escreva uma rotina *blivre(p,n)* que libere um bloco arbitrário *p* de *n* caracteres para a lista livre mantida por *aloca* e *libera*. Usando *blivre*, um usuário pode acrescentar um vetor estático ou externo à lista livre em qualquer instante. ■

# A. Manual de Referência

## A.1 Introdução

Este manual descreve a linguagem **C** especificada pelo "*Draft Proposed American National Standard for Information Systems - Programming Language C*" , documento número **X3.159/1989**, datado de 31 de outubro de 1988. Ele é uma interpretação do rascunho proposto pelo padrão, e não o padrão em si, embora tenhamos tido o cuidado de torná-lo um guia confiável para a linguagem.

Na sua maior parte, este manual segue o esboço principal do Esboço do Padrão, que por sua vez segue o trabalho da primeira edição deste livro, embora sua organização seja diferente em certos detalhes. Exceto por renomear algumas produções e não formalizar as definições dos códigos léxicos ou do pré-processador, a gramática mostrada aqui para a linguagem é equivalente à do esboço atual.

Durante este manual, os comentários são indentados e escritos em tipo menor, como este. Na maioria, eles destacam as formas em que o padrão **ANSI** para o **C** diferem da linguagem definida pela primeira edição deste livro, ou as melhorias subsequentemente introduzidas em vários compiladores.

C99 As mudanças implementadas posteriormente por C99 estão indicadas no texto.

## A.2 Convenções Léxicas

Um programa consiste em uma ou mais unidades de tradução armazenadas em arquivos. Ele é traduzido em diversas fases, descritas no parágrafo A.12. As primeiras fases realizam as transformações léxicas de baixo-nível, executam diretivas introduzidas por linhas começando com o caractere **#**, e realizam a definição e expansão de macros. Quando o pré-processamento do parágrafo A.12 estiver completo, o programa estará reduzido a uma sequência de códigos.

### A.2.1 Códigos

Existem seis classes de códigos: **identificadores**, **palavras-chave**, **constantes**, **literais de cadeia**, **operadores** e **outros separadores**. Espaços em branco, tabulações horizontal e vertical, nova-linha,

alimentação de formulário e comentários descritos a seguir (agrupados como "*espaço em branco*") são ignorados, exceto por separar códigos. Algum espaço em branco é necessário para separar identificadores, palavras-chave e constantes que, de outra forma, ficariam adjacentes.

Se o fluxo de entrada fosse separado em códigos até um determinado caractere, o próximo código seria a cadeia de caracteres mais longa que poderia constituir um código.

### A.2.2 Comentários

Os caracteres **/*** introduzem um comentário, que termina com os caracteres ***/**. Os comentários não podem ser encaixados, e não ocorrem dentro de strings ou literais de caracteres.

C99 Comentários também podem ser indicados por duas barras paralelas //, neste caso o comentário começa logo após estes caracteres e vai até o final da linha.

### A.2.3 Identificadores

Um **identificador** é uma sequência de letras e dígitos. O primeiro caractere deve ser uma letra; o caractere de sublinhado _ é contado como uma letra. As letras maiúsculas e minúsculas são diferentes. Identificadores podem ter qualquer tamanho, e para identificadores internos, pelo menos os primeiros **31 caracteres são significativos**; algumas implementações podem utilizar um conjunto maior de caracteres significativos. Os identificadores internos incluem nomes de macro do pré-processador e todos os outros nomes que não possuem ligação externa (A.11.2). Identificadores com ligação externa são mais restritos: as implementações podem considerar apenas seis caracteres como significativos, e podem ignorar as distinções entre maiúsculas e minúsculas.

C99 Em C99, os primeiros **63 caracteres são significativos** para identificadores internos. Para identificadores com ligação externa, os primeiros **31 caracteres** são relevantes e letras maiúsculas são diferentes de minúsculas.

### A.2.4 Palavras-Chave

Os identificadores a seguir são reservados para uso como palavras-chave, e não podem ser usados de outra forma:

| | | | |
|---|---|---|---|
| auto | double | int | struct |
| break | else | long | switch |
| case | enum | register | typedef |
| char | extern | return | union |
| const | float | short | unsigned |
| continue | for | signed | void |
| default | goto | sizeof | volatile |
| do | if | static | while |

Tabela A.1: Palavras-chave

Algumas implementações também reservam as palavras **fortran** e **asm**.

As palavras-chave **const**, **signed** e **volatile** são introduzidas pelo padrão **ANSI**; **enum** e **void** são novas desde a primeira edição, mas com uso comum; **entry**, inicialmente reservada mas nunca usada, não é mais reservada.

C99 5 novas paravras-chave em C99: *inline*, *restrict*, *_Bool*, *_Complex*, e *_Imaginary*.

### A.2.5 Constantes

Existem diversos tipos de constantes. Cada uma tem seu tipo de dado; o Parágrafo A.4.2 discute os tipos básicos.

*constante:*
*constante-inteira*
*constante-caractere*
*constante-flutuante*
*constante-enumeração*

#### A.2.5.1 Constantes Inteiras

Uma constante inteira contendo uma sequência de dígitos é considerada **octal** se começar com **0** (dígito zero), **decimal** em caso contrário. As constantes octais não contêm os dígitos 8 e 9. Uma sequência de dígitos precedida por 0x ou 0X (dígito zero) é considerada um inteiro hexadecimal. Os dígitos hexadecimais incluem *a* ou *A* até *f* ou *F* com valores de 10 a 15.

Uma constante inteira pode ter o sufixo **u** ou **U**, especificando que é não sinalizada. Ela também pode receber o sufixo **l** ou **L** para indicar que é longa.

O tipo de uma constante inteira depende de sua forma, valor e sufixo. (Veja §A.4 para uma discussão sobre tipos.) Se ela for **decimal** e sem sufixo, ela possui o primeiro tipo com o qual seu valor pode ser representado: *int*, *long int*, *unsigned long int*. Se for **octal** ou **hexadecimal** sem sufixo, ela possui o primeiro tipo possível dentre: *int*, *unsigned int*, *long int*, *unsigned long int*. Se tiver o sufixo **u** ou **U**, então *unsigned int*, *unsigned long int*. Se tiver o sufixo **l** ou **L**, então *long int*, *unsigned long int*. Se a constante inteira tiver o sufixo **UL**, ela é *unsigned long*.

A elaboração dos tipos de constantes inteiras desenvolveu-se bastante desde a primeira edição, que simplesmente afirmava que grandes constantes inteiras eram *long*. Os sufixos **U** são novos.

#### A.2.5.2 Constantes de Caracteres

Uma constante de caractere é uma sequência de um ou mais caracteres entre apóstrofos, como em *'x'*. O valor de uma constante de caractere com somente um caractere é o valor numérico do caractere no conjunto de caracteres da máquina definido em tempo de execução. O valor de uma constante com múltiplos caracteres é definido pela implementação.

As constantes de caractere não contêm o caractere ' ou de nova-linha; para representá-los, e alguns outros caracteres, as sequências de escape a seguir podem ser usadas.

| | | | | | |
|---|---|---|---|---|---|
| nova-linha | NL (LF) | \n | contrabarra | \ | \\ |
| tab horizontal | HT | \t | interrogação | ? | \? |
| tab vertical | VT | \v | apóstrofo | ' | \' |
| retrocesso | BS | \b | aspas | " | \" |
| retorno carro | CR | \r | núm. octal | ooo | \ooo |
| formulário | FF | \f | núm. hexa | hh | \xhh |
| sinal sonoro | BEL | \a | | | |

Tabela A.2: Sequências de escape.

A sequência de escape *ooo* consiste na **contrabarra** seguida por 1, 2 ou 3 dígitos octais, que especificam o valor do caractere desejado. Um exemplo comum desta construção é \0 (não seguido por um dígito), que especifica o caractere **NULO**. A sequência de escape *xhh* consiste na **contrabarra**, seguida por um **x**, seguido por dígitos hexadecimais, que especificam o valor do caractere desejado. Não há limite para o número de dígitos, mas o comportamento é indefinido se o valor do caractere resultante exceder ao valor do maior caractere. Para os caracteres de escape

**octal** ou **hexadecimal**, se a implementação tratar os tipo *char* como sinalizado, o valor é estendido por sinal como se fosse modificado para o tipo *char*. Se o caractere após o \ não for um dos que foram indicados aqui, o comportamento é indefinido.

Em algumas implementações, existe um conjunto estendido de caracteres que não pode ser representado no tipo *char*. Uma constante neste conjunto estendido é escrita precedendo-se um **L**, como em **L'x'**, e é chamada constante de caractere largo. Tal constante possui o tipo **wchar_t**, um tipo integral definido no cabeçalho-padrão <*stddef.h*>. Assim como em constantes comuns de caracteres, os escapes octal ou hexadecimal podem ser usados; o efeito é indefinido se o valor especificado exceder o máximo permissível para **wchar_t**.

Algumas dessas sequências de escape são novas, em particular a representação de caracteres em hexadecimal. Os caracteres estendidos também são novos. Os conjuntos de caracteres normalmente usados nas Américas e Europa ocidental podem ser codificados para que possam ser representados pelo tipo *char*; a principal finalidade do acréscimo de **wchar_t** foi acomodar as linguagens asiáticas.

#### A.2.5.3 Constantes de Ponto flutuante

Uma constante de ponto flutuante consiste em uma parte inteira, um ponto decimal, uma parte fracionária, um *e* ou *E*, e opcionalmente um expoente inteiro sinalizado e um sufixo de tipo opcional, que pode ser *f*, *F*, *l* ou *L*. As partes inteira e fracionária consistem em sequências de dígitos. Uma dessas partes pode estar ausente (não as duas); o ponto decimal ou o *e* com o expoente (não os dois) pode estar ausente. O tipo é determinado pelo sufixo; *F* ou *f* a torna *float*, *L* ou *l* a toma *long double*; caso contrário, o tipo é *double*.

Os sufixos para constantes de ponto flutuante são novos.

#### A.2.5.4 Constantes de Enumeração

Os identificadores declarados como enumeradores (veja §A.8.4) são constantes do tipo *int*.

### A.2.6 Literais de String

Um literal de **string**, também chamado **constante de string**, é uma sequência de caracteres rodeada por aspas, como em "...". Uma string tem o tipo "*vetor de caracteres*" e classe de armazenamento *static* (veja §A.3 a seguir), sendo inicializada com os caracteres indicados. Se os literais de string idênticos são distintos um do outro depende da implementação, e o comportamento de um programa que tenta alterar um literal de string é indefinido.

Literais de string adjacentes são concatenados em uma única string. Após qualquer concatenação, um byte nulo \0 é anexado à string para que o programa que a analisa possa encontrar seu fim. Os literais de string não contêm caracteres de nova-linha ou aspas; para representá-los as mesmas sequências de escape das constantes de caracteres estão disponíveis.

Assim como em constantes de caracteres, os **literais de string** em um conjunto estendido de caracteres são escritos com um *L* prefixado, como em **L". . ."**. Literais de string com caracteres estendidos possuem o tipo "*vetor de wchar_t*". A concatenação de literais de comuns com estendidos é indefinida.

A especificação de que os literais de string não precisam ser distintos, e a proibição contra modificá-los, são novos no padrão ANSI, assim como a concatenação de literais de string adjacentes. Os literais de string com caracteres estendidos são novos.

## A.3 Notação Sintática

Na notação sintática usada neste manual, as categorias sintáticas são indicadas por tipo *itálico*, e as palavras literais e caracteres em estilo **destacado**. As categorias alternativas são geralmente listadas em linhas separadas; em alguns poucos casos, um longo conjunto de pequenas alternativas é

apresentado em uma linha, marcados pela frase "*um dentre*". Um símbolo terminal ou não terminal possui o subscrito "**opt**", de forma que, por exemplo,

```
{ expressão_opt }
```

significa uma expressão opcional, delimitada por chaves. A sintaxe é resumida no §A.13.

Diferente da gramática apresentada na primeira edição deste livro, a que mostramos aqui torna explícitas a precedência e associatividade dos operadores da expressão.

## A.4 Significado dos Identificadores

*Identificadores*, ou *nomes*, referem-se a uma variedade de coisas: funções; etiquetas de estruturas, uniões e enumerações; membros de estruturas ou uniões; constantes de enumeração; rótulos; nomes de *typedef* e objetos. Um objeto, às vezes chamado variável, é um local na memória, e sua interpretação depende de dois atributos principais: sua classe de memória e seu tipo. A classe de memória determina o tempo de vida da memória associada ao objeto identificado; o tipo determina o significado dos valores achados no objeto identificado. Um nome também tem um escopo, que é a região do programa em que ele é conhecido, e uma ligação, que determina se o mesmo nome em outro escopo refere-se ao mesmo objeto ou função. O escopo e ligação são discutidos em A.11.

### A.4.1 Classe de Memória

Há duas classes de memória: **automática** e **estática**. Diversas palavras-chave, juntamente com o contexto da declaração de um objeto, especificam sua classe de memória. Os objetos automáticos são locais a um bloco (§A.9.3), e são descartados na saída do bloco. As declarações dentro do bloco criam objetos automáticos se nenhuma especificação de classe de memória for mencionada, ou se for usado o especificador *auto*. Os objetos declarados como *register* são automáticos, e (se possível) são armazenados em rápidos registradores internos da máquina.

Os objetos estáticos podem ser locais a um bloco ou externos a todos os blocos, mas de qualquer forma retêm seus valores na saída e reentrada para funções e blocos. Dentro de um bloco, incluindo um bloco que fornece o código para uma função, os objetos estáticos são declarados com a palavra-chave *static*. Os objetos declarados foram de todos os blocos, ao mesmo nível das definições de função, são sempre estáticos. Eles podem se tomar locais a uma determinada unidade de tradução pelo uso da palavra-chave *static*; isso lhes dá ligação interna. Eles tornam-se globais a um programa inteiro omitindo-se uma classe de memória explícita, ou por meio da palavra-chave *extern*; isso lhes dá ligação externa.

### A.4.2 Tipos Básicos

Existem diversos tipos fundamentais. O cabeçalho-padrão <*limits.h*> descrito no **Apêndice B** define os maiores máximo e mínimo de cada tipo na implementação local. Os números dados no **Apêndice B** mostram as menores ordens de grandeza aceitáveis.

Objetos declarados como caracteres (*char*) têm tamanho suficiente para armazenarem qualquer membro do conjunto de caracteres em execução. Se um caractere genuíno desse conjunto for armazenado em um objeto *char*, seu valor é equivalente ao código inteiro para o caractere, e não é negativo. Outras quantidades podem ser armazenadas em variáveis *char*, mas a faixa de valores disponíveis, e especialmente se o valor é sinalizado ou não, dependem da implementação.

Os caracteres não sinalizados declarados como *unsigned char* consomem a mesma quantidade de espaço que os caracteres comuns, mas sempre aparecem não-negativos; os caracteres explicitamente sinalizados declarados como *signed char* também ocupam o mesmo espaço dos caracteres comuns.

O tipo *unsigned char* não aparece na primeira edição deste livro, mas é comumente usado. *signed char* é novo.

Além dos tipos *char*, os três tamanhos de inteiro, declarados por *short int*, *int* e *long int*, estão disponíveis. Objetos puramente *int* possuem o tamanho natural sugerido pela arquitetura da máquina utilizada; os outros tamanhos servem para atender as necessidades especiais. Os inteiros mais longos fornecem pelo menos a quantidade de armazenamento dos mais curtos, mas a implementação pode tornar os inteiros comuns equivalentes aos inteiros curtos ou longos. Todos os tipos *int* representam valores sinalizados, a menos que seja especificado de outra forma.

Os inteiros não sinalizados , declarados usando-se a palavra-chave *unsigned*, obedecem às leis da aritmética de módulo $2^n$, onde $n$ é o número de bits na representação, e assim a aritmética em quantidades não sinalizadas nunca pode causar estouro. O conjunto de valores não-negativos que pode ser armazenado em um objeto sinalizado é um subconjunto dos valores que podem ser armazenados em um objeto não sinalizado correspondente, e a representação para valores superpostos é a mesma.

Qualquer um dentre ponto flutuante de precisão simples (*float*), ponto flutuante de dupla precisão (*double*) e ponto flutuante de precisão extra (*long double*) podem ser sinônimos, mas os últimos na lista devem ter pelo menos a precisão dos anteriores.

*long double* é novo. A primeira edição tornou *long float* equivalente a *double*; esta terminologia foi retirada.

**Enumerações** são tipos únicos que possuem valores inteiros; associado a cada enumeração está um conjunto de constantes nomeadas (§A.8.4). Enumerações comportam-se como inteiros, mas é comum a emissão de um aviso pelo compilador quando um objeto de determinado tipo de enumeração for atribuído a algo diferente de uma de suas constantes, ou uma expressão com seu tipo.

Como objetos desses tipos podem ser interpretados como números, eles serão referenciados como tipos aritméticos. Os tipos *char* e *int* de todos os tamanhos, cada um deles com ou sem sinal, e também os tipos de enumeração, serão chamados coletivamente tipos integrais. Os tipos *float*, *double* e *long double* serão chamados tipos de ponto flutuante.

O tipo *void* especifica um conjunto vazio de valores. Ele é usado como o tipo retornado por funções que não geram valor.

### A.4.3 Tipos Derivados

Além dos tipos básicos, existe uma classe conceitualmente infinita de tipos derivados construídos a partir dos tipos fundamentais e arrumados das seguintes formas:

vetores *de objetos de um certo tipo*;
funções *retornando objetos de um certo tipo*;
ponteiros *para objetos de um certo tipo*;
estruturas *contendo uma sequência de objetos de vários tipos*;
uniões *capazes de conter qualquer um dentre diversos objetos de vários tipos*.

Em geral, estes métodos de construção de objetos podem ser aplicados recursivamente.

### A.4.4 Qualificadores de Tipo

O tipo de um objeto pode ter qualificadores adicionais. Declarar um objeto *const* anuncia que seu valor não será alterado; declará-lo *volatile* anuncia que ele possui propriedades especiais relevantes à otimização. Nenhum qualificador afeta a faixa de valores ou as propriedades aritméticas do objeto. Os qualificadores são discutidos no §A.8.2.

## A.5 Objetos e Lvalues

Um objeto é um local nomeado ou apontado da memória; um **lvalue** é uma expressão referindo-se a um objeto. Um exemplo óbvio de uma expressão **lvalue** é um identificador com o tipo e classe de memória adequados. Existem operadores que geram **lvalues**: por exemplo se E for uma expressão do tipo ponteiro, então *E é uma expressão **lvalue** referindo-se ao objeto para o qual E aponta. O nome "*lvalue*" vem da expressão de atribuição *E1 = E2*, em que o operando esquerdo E1 deve ser uma expressão **lvalue**. A discussão de cada operador especifica se ele espera operandos **lvalue** e se produz um **lvalue**.

## A.6 Conversões

Alguns operadores podem, dependendo de seus operandos, causar a conversão do valor de um operando de um tipo para outro. Esta seção explica o resultado a ser esperado nessas conversões. O §A.6.5 resume as conversões executadas pela maioria dos operadores comuns; o resumo será suplementado pela discussão de cada operador quando for necessário.

### A.6.1 Promoção Integral

Um caractere inteiro curto, ou um campo de bit inteiro, todos sinalizados ou não, ou um objeto do tipo enumerado podem ser usados em uma expressão sempre que um inteiro puder ser usado. Se um *int* pode representar todos os valores do tipo original, então o valor é convertido para *int*; caso contrário o valor é convertido para *unsigned int*. Este processo é chamado promoção integral.

### A.6.2 Conversões Integrais

Qualquer inteiro é convertido para um certo tipo não sinalizado encontrando-se o menor valor não-negativo congruente a esse inteiro, módulo um a mais que o maior valor que pode ser representado no tipo não-sinalizado. Na representação por complemento de dois, isto é equivalente a truncar à esquerda se o padrão de bits do tipo não-sinalizado for mais estreito, e preencher com zeros valores não-sinalizados e estender por sinal os valores sinalizados se o tipo não-sinalizado for mais largo.

Quando qualquer inteiro é convertido a um tipo sinalizado, o valor é inalterado se ele puder ser representado no novo tipo, e, em caso contrário, o valor é definido pela implementação.

### A.6.3 Inteiro e Ponto flutuante

Quando um valor do tipo ponto flutuante é convertido para um tipo inteiro, a parte fracionária é eliminada; se o valor resultante não puder ser representado no tipo integral, o comportamento é indefinido. Em particular, o resultado da conversão de valores de ponto flutuante negativos para tipos integrais não-sinalizados não é especificado.

Quando o valor de tipo integral é convertido para ponto flutuante, e o valor está na faixa representável mas não é representável exatamente, então o resultado pode ser o próximo valor representável mais baixo ou mais alto. Se o resultado estiver fora da faixa, o comportamento é indefinido.

### A.6.4 Tipos de Ponto flutuante

Quando um valor de ponto flutuante menos preciso é convertido para um tipo de ponto flutuante de precisão igual ou maior, o valor é inalterado. Quando um valor mais preciso é convertido para um tipo menos preciso, ambos em ponto flutuante, e o valor está dentro da faixa representável, o resultado pode ser o próximo valor representável maior ou menor. Se o resultado estiver fora da faixa, o comportamento é indefinido.

### A.6.5 Conversões Aritméticas

Muitos operadores causam conversões e produzem tipos resultantes de forma semelhante. O efeito é levar os operandos a um tipo comum, que é também o tipo do resultado. Este padrão é chamado conversões aritméticas comuns.

- Primeiro, se um dos operandos é *long double*, o outro é convertido para *long double*.
- Caso contrário, se um dos operandos é *double*, o outro é convertido para *double*.
- Caso contrário, se um dos operandos é *float*, o outro é convertido para *float*.
- Caso contrário, as promoções integrais são executadas nos dois operandos; então, se um deles for *unsigned long int*, o outro é convertido para *unsigned long int*.
- Caso contrário, se um operando é *long int* e o outro é *unsigned int*, o efeito depende se um *long int* pode ou não representar todos os valores de um unsigned int; se puder o operando *unsigned int* é convertido para *long int*; senão, ambos são convertidos para *unsigned long int*.
- Caso contrário, se um operando é *long int*, o outro é convertido para *long int*.
- Caso contrário, se um dos operandos é *unsigned int*, o outro é convertido para *unsigned int*.
- Caso contrário, os dois operandos têm o tipo *int*.

Há duas mudanças aqui. Primeiro, a aritmética em operandos *float* pode ser feita em precisão simples, ao invés de dupla; a primeira edição especificava que toda a aritmética de ponto flutuante era executada em dupla precisão. Segundo, tipos não sinalizados mais curtos, quando combinados com um tipo sinalizado mais longo, não propagam a propriedade não-sinalizada ao tipo resultante; na primeira edição, os não-sinalizados sempre dominavam. As novas regras são um pouco mais complicadas, mas reduzem as surpresas que podem ocorrer quando uma quantidade não-sinalizada encontra uma sinalizada. Resultados inesperados ainda podem ocorrer quando uma expressão não-sinalizada é comparada com uma expressão sinalizada do mesmo tamanho.

### A.6.6 Ponteiros e Inteiros

Uma expressão de tipo integral pode ser **somada ou subtraída** a um ponteiro; nesse caso, a expressão integral é convertida conforme especifica a discussão do operador de adição (§A.7.7).

Dois ponteiros para objetos do mesmo tipo, no mesmo vetor, podem ser **subtraídos**; o resultado é convertido para um inteiro conforme especifica a discussão do operador de subtração (§A.7.7).

Uma expressão constante integral com o valor 0, ou uma expressão modificada para o tipo *void* *, pode ser convertida, por meio de um molde, por atribuição, ou por comparação, a um ponteiro de qualquer tipo. Isto produz um ponteiro nulo que é igual a um outro ponteiro nulo do mesmo tipo, mas diferente de qualquer ponteiro para uma função ou objeto.

Outras conversões envolvendo ponteiros são permitidas, mas possuem aspectos dependentes da implementação. Elas devem ser especificadas por meio de um operador explícito de conversão de tipo, ou por um molde (§A.7.5 e §A.8.8).

Um ponteiro pode ser convertido a um tipo integral com tamanho suficiente para contê-lo; o tamanho exigido depende da implementação. A função de mapeamento também depende da implementação.

Um objeto de tipo integral pode ser explicitamente convertido a um ponteiro. O mapeamento sempre executa um inteiro suficientemente largo convertido de um ponteiro para o mesmo ponteiro, mas de outra forma é dependente da implementação.

Um ponteiro para um tipo pode ser convertido para um ponteiro para outro tipo. O ponteiro resultante pode causar exceções de endereçamento se o ponteiro sujeito não se referir a um objeto apropriadamente alinhado na memória. Garantidamente, um ponteiro para um objeto pode ser convertido a um ponteiro para um objeto cujo tipo requer alinhamento de memória com restrição igual ou menor e de volta sem mudança; a noção de "*alinhamento*" depende da implementação, mas os objetos dos tipos *char* possuem requisitos de alinhamento, mesmo estritos. Como descrevemos no §A.6.8, um ponteiro também pode ser convertido para o tipo *void* *, e de volta, sem qualquer

mudança.

Finalmente, um ponteiro para uma função pode ser convertido a um ponteiro para outro tipo de função. A chamada da função especificada por um ponteiro convertido depende da implementação; entretanto, se o ponteiro convertido for reconvertido para seu tipo original, o resultado é idêntico ao ponteiro original.

Um ponteiro pode ser convertido para outro ponteiro cujo tipo seja o mesmo, exceto pelo acréscimo ou remoção de qualificadores (§§A.4.4 e A.8.2) do tipo objeto ao qual o ponteiro se refere. Se forem incluídos qualificadores, o novo ponteiro é equivalente ao antigo exceto pelas restrições geradas pelos novos qualificadores. Se os qualificadores forem removidos, as operações com o objeto básico permanecem sujeitas aos qualificadores na sua declaração real.

### A.6.7 Void

O valor (não-existente) de um objeto *void* não pode ser usado de qualquer forma, e nenhuma conversão explícita ou implícita a qualquer tipo *não-void* pode ser aplicada. Como uma expressão *void* indica um valor não existente, tal expressão só pode ser usada onde o valor não for exigido, por exemplo como um comando de expressão (§A.9.2) ou como o operando esquerdo de um operador vírgula (§ efgeracaodeponteiro8).

Uma expressão pode ser convertida para o tipo *void* por meio de um molde. Por exemplo, um molde *void* documenta a eliminação do valor de uma chamada de função usada como um comando de expressão.

*void* não apareceu na primeira edição deste livro, mas tem se tornado comum desde então.

### A.6.8 Ponteiros para Void

Qualquer ponteiro para um objeto pode ser convertido para o tipo *void* * sem perda de informação. Se o resultado é convertido de volta ao tipo original do ponteiro, o ponteiro original é recuperado. Diferente das conversões de **ponteiro-para-ponteiro** discutidas no §A.6.6, que exigem um molde explícito, os ponteiros podem ser atribuídos *para* e *de* ponteiros do tipo *void* *, e podem ser comparados com eles.

Esta interpretação de ponteiros *void* * é nova; anteriormente, ponteiros *char* * desempenhavam o papel de ponteiro genérico. O padrão **ANSI** louva o uso de ponteiros *void* * com ponteiros de objeto em atribuições e relações, enquanto exige moldes explícitos para outras misturas de ponteiro.

## A.7 Expressões

A precedência dos operadores de expressão é a mesma que a ordem das principais subseções desta seção, com a mais alta precedência em primeiro lugar. Assim, por exemplo, as expressões referenciadas como operandos de + (§A.7.7) são aquelas expressões definidas nos §§ efgeracaodeponteiro-A.7.6. Dentro de cada subseção, os operadores têm a mesma precedência. A associatividade esquerda ou direita é especificada em cada subseção para os operadores discutidos lá. A gramática incorpora a precedência e associatividade dos operadores de expressão; ela é resumida no §A.13.

A precedência e associatividade dos operadores é totalmente especificada, mas a ordem de avaliação das expressões, com certas exceções, é indefinida, mesmo que as subexpressões envolvam efeitos colaterais, ou seja, a menos que a definição de um operador garanta que seus operandos são avaliados em uma ordem particular, a implementação é livre para avaliar os operandos e em qualquer ordem, ou até mesmo intercalar sua avaliação. No entanto, cada operador combina os valores produzidos por seus operandos de maneira compatível com a análise da expressão em que aparece.

O comitê **ANSI** decidiu, mais adiante nos seus procedimentos, restringir a liberdade anterior de reordenar as expressões envolvendo operadores matematicamente comutativos e associativos,

mas que possam falhar ao serem associados computacionalmente. Na prática, a mudança só afeta cálculos de ponto flutuante próximos aos limites de sua precisão, e situações onde o estouro é possível.

A manipulação de estouro, checagem de divisão e outras exceções na avaliação de expressão não são definidas pela linguagem. A maioria das implementações existentes de **C** ignora o estouro na avaliação de expressões integrais sinalizadas e atribuições, mas este comportamento não é garantido. O tratamento da divisão por zero, e todas as exceções de ponto flutuante, varia segundo a implementação; às vezes isso é ajustável por meio de uma função de biblioteca não-padrão.

### A.7.1 Geração de Ponteiro

Se o tipo de uma expressão ou subexpressão for "*vetor de T*", para algum tipo *T*, então o valor da expressão é um ponteiro para o primeiro objeto no vetor, e o tipo da expressão é alterado para "*ponteiro para T*". Esta conversão não ocorre se a expressão for o operando do operador unário *& sizeof*. Semelhantemente, uma expressão do tipo "*função retornando T*", exceto quando usada como operando do operador *&*, é convertida para "*ponteiro para função retornando T*". Uma expressão que sofreu uma dessas conversões não é um **lvalue**.

### A.7.2 Expressões Primárias

Expressões primárias são identificadores, constantes, cadeias ou expressões entre parênteses.

expressão-primária:
identificador
constante
string
(expressão)

Um identificador é uma expressão primária, desde que tenha sido corretamente declarado conforme discutimos a seguir. Seu tipo é especificado por sua declaração. Um identificador é um **lvalue** se se refere a um objeto (§A.5) e se seu tipo for aritmético, estrutura, união ou ponteiro.

Uma constante é uma expressão primária. Seu tipo depende do seu formato, como discutimos no §A.2.5.

Uma literal de *string* é uma expressão primária. Seu tipo é originalmente um "*vetor de char*" (para cadeias de caracteres estendidos, "*vetor de wchar_t*"), mas seguindo a regra dada no §A.7.1, isso geralmente é modificado para "*ponteiro para char*"(ou *wchar_t*) e o resultado é um ponteiro para o primeiro caractere na *string*. A conversão também não ocorre com certos inicializadores; veja §A.8.7.

Uma expressão entre parênteses é uma expressão primária cujo tipo e valor são idênticos aos de uma expressão comum sem enfeites. A presença de parênteses não afeta se a expressão é um **lvalue**.

### A.7.3 Expressões Pós-fixadas

Os operadores em expressões pós-fixadas associam-se da **esquerda para a direita**.

*expressão-pós-fixada*:
expressão-primária
expressão-pós-fixada [expressão]
expressão-pós-fixada(lista-argumento$_{opc}$)
expressão-pós-fixada.identificador
expressão-pós-fixada->identificador
expressão-pós-fixada $++$
expressão-pós-fixada $--$

lista-argumento:

expressão-atribuição
lista-argumento , expressão-atribuição

### A.7.3.1 Referências de Vetor

Uma expressão **pós-fixada** seguida por uma expressão entre colchetes é uma expressão pós-fixada denotando uma referência de vetor subscritado. Uma das duas expressões deve ter o tipo "*ponteiro para T*", onde *T* é algum tipo, e a outra deve ter um tipo integral; o tipo da expressão subscrita é *T*. A expressão *E1[E2]* é idêntica (por definição) a **((E1)+(E2))*. Veja §A.8.6.2 para maiores explicações.

### A.7.3.2 Chamadas de Função

Uma chamada de função é uma expressão pós-fixada, chamada designadora de função, seguida de parênteses contendo uma lista, possivelmente vazia, de expressões de atribuição separadas por vírgula (§A.7.17), que constituem os argumentos para a função. Se a expressão pós-fixada consistir em um identificador para o qual não existe declaração no escopo corrente, o identificador é implicitamente declarado como se a declaração.

```
extern int identificador( );
```

tivesse sido dada no bloco mais interno contendo a chamada de função. A expressão pós-fixada (após possível declaração implícita e geração de ponteiro, § efgeracaodeponteiro) deve ser do tipo "*ponteiro para função retornando T*", para algum tipo *T*, e o valor da chamada de função tem o tipo *T*.

Na primeira edição, o tipo foi restrito à "*função*" e um operador * explícito foi necessário para chamadas por meio de ponteiros para funções. O padrão **ANSI** louva a prática de alguns compiladores existentes, permitindo a mesma sintaxe para chamadas a funções e para funções especificadas por ponteiros. A antiga sintaxe ainda é usada.

O termo argumento é usado para uma expressão passada por uma chamada de função; o termo parâmetro é usado para um objeto de entrada (ou seu identificador) recebido por uma definição de função, ou descrito em uma declaração de função. Os termos "*argumento (parâmetro) real*" e "*argumento (parâmetro) formal*" respectivamente são usados para se fazer a mesma distinção.

Na preparação para a chamada a uma função, é feita uma cópia para cada argumento; toda passagem de argumento **é estritamente por valor**. Uma função pode mudar os valores de seus objetos de parâmetro, que são cópias das expressões do argumento, mas estas mudanças não podem afetar os valores dos argumentos. Entretanto é possível passar um ponteiro entendendo-se que a função pode alterar o valor do objeto para o qual o ponteiro indica.

Existem dois estilos em que as funções podem ser declaradas. No novo estilo, os tipos de parâmetros são explícitos e fazem parte do tipo da função; tal declaração é também chamada protótipo de função. No estilo antigo, os tipos de parâmetro não são especificados. A declaração de função é discutida nos §§A.8.6.3 e A.10.1.

Se a declaração de função no escopo de uma chamada estiver no estilo antigo, então a promoção *default* de argumento é aplicada a cada argumento da seguinte forma: a promoção integral (§A.6.1) é executada em cada argumento de tipo integral, e cada argumento *float* é convertido para *double*. O efeito da chamada é indefinido se o número de argumentos discordar com o número de parâmetros na definição da função, ou se o tipo de um argumento após a promoção discordar com o do parâmetro correspondente. A concordância de tipo depende do estilo (novo ou antigo) da definição da função. Se estiver no estilo antigo, então a comparação e feita entre o tipo promovido do argumento da chamada e o tipo promovido do parâmetro; se a definição estiver no estilo novo, o tipo promovido do argumento deve ser o do próprio parâmetro, sem promoção.

Se a declaração de função no escopo de uma função estiver no estilo novo, então os argumentos são convertidos , como se fosse uma atribuição, para os tipos dos parâmetros correspondentes

do protótipo da função. O número de argumentos deve ser o mesmo do número de parâmetros descritos explicitamente, a menos que a lista de parâmetros da declaração termine com a notação de reticências (, ...). Neste caso, o número de argumento deve ser igual ou maior que o número de parâmetros; argumentos restantes após os parâmetros indicados explicitamente sofrem a promoção *default* do argumento, descrita no parágrafo anterior. Se a definição da função estiver no estilo antigo, então o tipo de cada parâmetro no protótipo visível em cada chamada deve concordar com o parâmetro correspondente na definição, após o tipo de parâmetro da definição tiver sofrido a promoção de argumento.

Estas regras são especialmente complicadas porque devem fornecer detalhes para uma mistura de estilos novo e antigo de funções. As misturas devem ser evitadas sempre que for possível.

A ordem de avaliação dos argumentos não é especificada; observe que vários compiladores trabalham de formas diferentes. Contudo os argumentos e o designador da função são avaliados totalmente, incluindo todos os efeitos colaterais, antes da função ser entrada. As chamadas recursivas a qualquer função são permitidas.

#### A.7.3.3 Referências de Estrutura

A expressão pós-fixada seguida por um ponto e um identificador é uma expressão pós-fixada. O primeiro operando da expressão deve ser uma estrutura ou união. O valor é o membro nomeado da estrutura ou união, e seu tipo é o tipo do membro. A expressão é um **lvalue** se a primeira expressão for um **lvalue**, e se o tipo da segunda expressão não for um tipo vetor.

Uma expressão pós-fixada seguida por uma seta (montada de $-$ e $>$) seguida por um identificador é uma expressão pós-fixada. O primeiro operando da expressão deve ser um ponteiro para uma estrutura ou união, e o identificador deve nomear um membro da estrutura ou união. O resultado refere-se ao membro nomeado da estrutura ou união a qual a expressão apontadora indica, e o tipo é o tipo do membro; o resultado é um **lvalue** se o tipo não for um tipo vetor.

Assim, a expressão `E1->MOS` é o mesmo que `(*E1).MOS`. As estruturas e uniões são discutidas no §A.8.3.

Na primeira edição deste livro, a regra era de que um nome de membro em uma expressão desse tipo tinha que pertencer à estrutura ou união mencionada na expressão pós-fixada; contudo, uma nota admitia que esta regra não era rigidamente forçada. Os compiladores recentes, e o **ANSI**, a forçam.

#### A.7.3.4 Incrementação Pós-fixada

Uma expressão pós-fixada seguida por um operador $++$ ou $--$ é uma expressão pós-fixada. O valor da expressão é o valor do operando. Após ser registrado o valor, o operando é incrementado ($++$) ou decrementado ($--$) de 1. O operando deve ser um **lvalue**; veja a discussão sobre operadores aditivos (§A.7.7) e atribuição (§A.7.17) para conhecer mais restrições sobre o operando e detalhes da operação. O resultado não é um **lvalue**.

### A.7.4 - Operadores unários

As expressões com operadores unários se associam da direita para a esquerda.

expressão-unária:
- expressão-pós-fixada
- expressão-unária $++$
- expressão-unária $--$
- operador-unário expressão-molde
- sizeof expressão-unária
- sizeof (nome-tipo)

operador-unário: um dentre
- $\& * + - \sim |$

### A.7.4.1 Operadores de Incrementação Prefixado

Uma expressão unária precedida por um operador $++$ ou $--$ é uma expressão unária. O operando é incrementado $(++)$ ou decrementado $(--)$ de 1 . O valor da expressão é o valor após a incrementação (decrementação). O operando deve ser um **lvalue**; veja a discussão sobre operadores aditivos (§A.7.7) e atribuição (§A.7.17) para mais restrições sobre o operando e detalhes de operação. O resultado não é um **lvalue**.

### A.7.4.2 Operador de Endereço

O operador unário & toma o endereço de seu operando. O operando deve ser um **lvalue** que não se refira a um campo de bit e nem a um objeto declarado como *register*, ou então deve ser do tipo função. O resultado é um ponteiro para o objeto ou função referenciado pelo **lvalue**. Se o tipo do operando é *T*, o tipo do resultado é "*ponteiro para T*".

### A.7.4.3 Operador de Indireção

O operador unário * indica indireção, e retorna o objeto ou função para o qual seu operando aponta. Ele é um **lvalue** se o operando for um ponteiro para um objeto de tipo aritmético, estrutura, união ou ponteiro. Se o tipo da expressão é "*ponteiro para T*", o tipo do resultado é *T*.

### A.7.4.4 Operador Unário Mais

O operando do operador unário $+$ deve ter tipo aritmético, e o resultado é o valor do operando. Um operando integral passa por promoção integral. O tipo do resultado é o tipo do operando promovido.

O operador unário $+$ é novo com o padrão **ANSI**. Ele foi incluído por simetria com o operador unário $-$.

### A.7.4.5 Operador Unário Menos

O operando do operador unário $-$ deve ter um tipo aritmético, e o resultado é o negativo do seu operando. Um operando integral passa por promoção integral. O negativo de uma quantidade não sinalizada é calculado subtraindo-se o valor promovido do maior valor do tipo promovido e somando-se um; mas o negativo de zero é zero. O tipo do resultado é o tipo do operando promovido.

### A.7.4.6 Operador de Complemento a Um

O operando de um operador ~ deve ter tipo integral, e o resultado é o **complemento a um** do seu operando. As promoções integrais são realizadas. Se o operando não for sinalizado, o resultado é calculado subtraindo-se o valor do maior valor do tipo promovido. Se o operando for sinalizado, o resultado é calculado convertendo-se o operando promovido para o tipo não-sinalizado correspondente, aplicando-se , e convertendo-se de volta para o tipo sinalizado. O tipo de resultado é o tipo do operando promovido.

### A.7.4.7 Operador de Negação Lógica

O operando do operador ! deve ter tipo aritmético ou ser um ponteiro, e o resultado é *1* se o valor do seu operando for igual a 0, e 0 caso contrário. O tipo do resultado é *int*.

### A.7.4.8 Operador Sizeof

O operador *sizeof* produz o número de bytes exigidos para armazenar um objeto do tipo de seu operando. O operando pode ser uma expressão, que não é avaliada, ou um nome de tipo entre parênteses. Quando *sizeof* é aplicado a um *char*, o resultado é 1; quando aplicado a um vetor, o resultado é o número total de bytes no vetor. Quando aplicado a uma estrutura ou união, o resultado é o número de bytes no objeto, incluindo qualquer preenchimento necessário para tornar o objeto em vetor: o tamanho de um vetor de *n* elementos é *n* vezes o tamanho de um elemento. O operador não pode ser aplicado a um operando do tipo função, ou de tipo incompleto ou a um campo de bit. O resultado é uma constante integral não sinalizada; o tipo particular é definido pela implementação. O cabeçalho-padrão <*stddef.n*> (veja Apêndice B) define este tipo como *size_t*.

### A.7.5 Moldes

Uma expressão unária precedida por um nome entre parênteses de um tipo causa a conversão do valor da expressão para o tipo indicado.

expressão-molde

expressão-unária
( nome-tipo ) expressão-molde

Esta construção é chamada **molde**. Os nomes de tipos são descritos no §A.8.8, e os efeitos das conversões no §A.6. Uma expressão com um molde não é um **lvalue**.

### A.7.6 Operadores Multiplicativos

Os operadores multiplicativos *, /, e % associam-se da esquerda para a direita.

expressão-multiplicativa:
expressão-molde
expressão-multiplicativa $*$ expressão-molde
expressão-multiplicativa / expressão-molde
expressão-multiplicativa % expressão-molde

Os operandos de $*$ e / devem ter tipo aritmético; os operandos de % devem ter tipo integral. As conversões aritméticas normais são executadas sobre os operandos, e predizem o tipo do resultado.

O operador binário $*$ indica multiplicação.

O operador binário / produz o quociente, e o operador % o resto da divisão do primeiro operando pelo segundo; se o segundo operando for 0, o resultado não é definido. Caso contrário, sempre é verdade que $(a/b)*b+a\%b$ é igual a $a$. Se os dois operandos forem não-negativos, então o resto é não-negativo e menor que o divisor; se não, só se pode garantir que o valor absoluto do resto é menor que o valor absoluto do divisor.

### A.7.7 Operadores Aditivos

Os operadores aditivos $+$ e $-$ associam-se da esquerda para a direita. Se os operandos têm tipo aritmético, as conversões aritméticas comuns são executadas. Existem algumas possibilidades de tipo adicionais para cada operador.

expressão-aditiva:
expressão-multiplicativa
expressão-aditiva $+$ expressão-multiplicativa
expressão-aditiva $-$ expressão-multiplicativa

O resultado do operador $+$ é a soma dos operandos. Um ponteiro para um objeto em um vetor e um valor de qualquer tipo integral pode ser somado. O segundo é convertido para um deslocamento de endereço multiplicando-o pelo tamanho do objeto para o qual o ponteiro aponta. A soma é um ponteiro do mesmo tipo que o ponteiro original, apontando para outro objeto no mesmo vetor, deslocado apropriadamente do objeto original. Assim, se *P* for um ponteiro para um objeto num vetor, a expressão *P + 1* é um ponteiro para o próximo objeto no vetor. Se a soma apontar para fora dos limites do vetor, exceto no primeiro local além do seu fim, o resultado é indefinido.

A provisão para ponteiros logo após o final de um vetor é nova. Ela está de acordo com um idioma comum para laços que passam para fora dos elementos de um vetor.

O resultado do operador $-$ é a diferença dos operandos. Um valor de qualquer tipo integral pode ser subtraído de um ponteiro, e então aplicam-se as mesmas conversões e condições que a adição.

Se dois ponteiros para objetos do mesmo tipo são subtraídos, o resultado é um valor integral sinalizado representando o deslocamento entre os objetos apontados; os ponteiros para objetos sucessivos têm 1 como resultado da subtração. O tipo do resultado depende da implementação,

mas está definido como *ptrdiff_t* no cabeçalho-padrão <*stddef.h*>. O valor é indefinido, a menos que os ponteiros apontem para objetos dentro do mesmo vetor; contudo, se $P$ aponta para o último membro de um vetor, então *(P+1)-P* tem o valor 1.

### A.7.8 Operadores de Deslocamento

Os operadores de deslocamento $<<$ e $>>$ associam-se da esquerda para a direita. Para ambos os operadores, cada operando deve ser integral , e está sujeito às promoções integrais. O tipo do resultado é o do operando esquerdo promovido. O resultado é indefinido se o operando direito for negativo, ou maior ou igual ao número de bits no tipo da expressão esquerda.

expressão-deslocamento:
expressão-aditiva
expressão-deslocamento $<<$ expressão-aditiva
expressão-deslocamento $>>$ expressão-aditiva

O valor de $E1 << E2$ é $E1$ (interpretado como padrão de bit) deslocando à esquerda de $E2$ bits; na ausência de estouro, isto é equivalente à multiplicação por $2^{E2}$. O valor de $E1 >> E2$ é $E1$ deslocado à direita de $E2$ posições de bit. O deslocamento à direita é equivalente à divisão por $2^{E2}$ se $E1$ for não-sinalizado ou se tiver um valor não-negativo; caso contrário o resultado é definido pela implementação.

### A.7.9 Operadores Relacionais .

Os operadores relacionais associam-se da esquerda para a direita, mas este fato não é útil; $a < b < c$ é analisado como $(a < b) < c$, e $a < b$ é avaliado como 0 ou 1.

expressão-relacional:
expressão-deslocamento
expressão-relacional $<$ expressão-deslocamento
expressão-relacional $>$ expressão-deslocamento
expressão-relacional $<=$ expressão-deslocamento
expressão-relacional $>=$ expressão-deslocamento

Os operadores $<$ (menor que), $>$ (maior que), $<=$ (menor ou igual a) e $>=$ (maior ou igual a) produzem 0 se a relação especificada for falsa, e 1 se for verdadeira. O tipo do resultado é *int*. As conversões aritméticas comuns são executadas em operandos aritméticos. Ponteiros para objetos do mesmo tipo (ignorando quaisquer qualificadores) podem ser comparados; o resultado depende dos locais relativos no espaço de endereçamento dos objetos apontados. A comparação de ponteiro é definida somente para partes do mesmo objeto; se dois ponteiros apontam para o mesmo objeto simples, eles são comparados como iguais; se os ponteiros forem para membros da mesma estrutura, os ponteiros para objetos declarados mais adiante na estrutura são maiores; se os ponteiros forem para membros da mesma união, eles são considerados iguais; se os ponteiros referem-se a membros de um vetor, a comparação é equivalente à comparação dos subscritos correspondentes. Se $P$ aponta para o último membro de um vetor, então $P+1$ é considerado maior do que $P$, mesmo que $P+1$ aponte para fora do vetor. Caso contrário, a comparação de ponteiros é indefinida.

Estas regras liberam ligeiramente as restrições afirmadas na primeira edição, permitindo a comparação de ponteiros para diferentes membros de uma estrutura ou união. Elas também legalizam a comparação com um ponteiro logo após o fim de um vetor.

### A.7.10 Operadores de Igualdade

expressão-igualdade:
expressão-relacional
expressão-igualdade $==$ expressão-relacional
expressão-igualdade $!=$ expressão-relacional

Os operadores == (**igual a**) e ! = (**não igual a**) são análogos aos operadores relacionais, exceto por sua precedência menor. (Assim, $a < b == e < d$ é 1 sempre que $a < b$ e $c < d$ tiverem o mesmo valor-verdade.) Operadortes de igualdade para ponteiros também podem ser aplicados em ponteiros de objetos diferentes.

Os operadores de igualdade seguem as mesmas regras dos operadores relacionais, mas permitem outras possibilidades: um ponteiro pode ser comparado a uma expressão integral constante com o valor 0, ou a um ponteiro para o *void*. Veja §A.6.6.

### A.7.11 Operador E Bit-a-Bit

expressão-E:

expressão-igualdade
expressão-E & expressão-igualdade

As conversões aritméticas normais são executadas; o resultado é a função **E** (**AND**) **bit-a-bit** dos operandos. O operador aplica-se somente a operandos integrais.

### A.7.12 Operador OU Exclusivo Bit-a-Bit

expressão-OU-exclusivo:
expressão-E
expressão-OU-exclusivo ^ expressão-E

As conversões aritméticas normais são executadas; o resultado é a função **OU** (**OR**) **exclusivo bit-a-bit** dos operandos. O operador aplica-se somente a operandos integrais.

### A.7.13 Operador OU Inclusivo Bit-a-Bit

expressão-OU-inclusiva:
expressão-OU-exclusivo
expressão-OU-inclusiva | expressão-OU-exclusivo

As conversões aritméticas normais são executadas; o resultado é a função **OU** (**OR**) **inclusiva bit-a-bit** de seus operandos. O operador aplica-se somente a operandos integrais.

### A.7.14 Operador E Lógico

expressão-E-lógico:
expressão-OU-inclusiva
expressão-E-lógico && expressão-OU-inclusiva

O operador && associa-se da esquerda para a direita. Ele retorna 1 se os dois operandos forem diferentes de zero, 0, caso contrário. Diferente de &, && garante a avaliação da esquerda para a direita: o primeiro operando é avaliado, incluindo todos os efeitos colaterais; se for igual a 0, o valor da expressão é 0. Caso contrário, o operando direito é avaliado, e se for igual a 0, o valor da expressão é 0, caso contrário 1.

Os operandos não precisam ter o mesmo tipo, mas cada um deve ter um tipo aritmético ou ser um ponteiro. O resultado é int.

### A.7.15 Operador OU Lógico

expressão-OU-lógico:
expressão-E-lógico
expressão-OU-lógico || expressão-E-lógico

O operador || associa-se da esquerda para a direita. Ele retorna 1 se um dos seus operandos for diferente de 0, e 0 caso contrário. Diferente de |, || garante avaliação da esquerda para a direita: o

primeiro operando é avaliado, incluindo todos os efeitos colaterais; se for diferente de 0, o valor da expressão é 1. Caso contrário, o operando da direita é avaliado, e se for diferente de 0, o valor da expressão é 1, caso contrário 0.

Os operandos não precisam ter o mesmo tipo, mas cada um deve ter um tipo aritmético ou ser um ponteiro. O resultado é *int*.

### A.7.16 Operador Condicional

expressão-condicional:

expressão-OU-lógico
expressão-OU-lógico ? expressão : expressão-condicional

A primeira expressão é avaliada, incluindo todos os efeitos colaterais; se for diferente de 0, o resultado é o valor da segunda expressão, caso contrário o da terceira. Somente um dentre o segundo e terceiro operandos é avaliado. Se o segundo e terceiro operandos forem aritméticos, as conversões aritméticas normais são executadas para que tenham um tipo comum, e esse é o tipo do resultado. Se ambos forem *void*, ou estruturas ou uniões do mesmo tipo, ou ponteiros para objetos do mesmo tipo, o resultado tem o tipo comum. Se um for um ponteiro e outro a constante 0, o 0 é convertido para o tipo ponteiro, e o resultado tem esse tipo. Se um for um ponteiro para *void* e o outro for outro ponteiro, o outro ponteiro é convertido para um ponteiro para *void*, e esse é o tipo do resultado.

Na comparação de tipo para ponteiros, quaisquer qualificadores do tipo (§A.8.2) para o qual o ponteiro aponta são insignificantes, mas o tipo do resultado herda os qualificadores dos dois braços da condicional.

### A.7.17 Expressões de Atribuição

Há diversos operadores de atribuição; todos se associam da direita para a esquerda.

*expressão-atribuição:*
*expressão-condicional*
*expressão-unária operador-atrib expressão-atrib*

*operador-atrib: um dentre*
$= *= /= \%= += -= <<= >>= \&= \hat{}= |=$

Todos exigem um **lvalue** como operando esquerdo, e o **lvalue** deve ser modificável: ele não deve ser um vetor, e não deve ter um tipo incompleto, e nem ser uma função. Além disso, seu tipo não deve ser qualificado com *const*; se for uma estrutura ou união, não deve ter qualquer membro ou, recursivamente, sub-membro qualificado com const. O tipo de uma expressão de atribuição é aquele do seu operando esquerdo, e o valor é o valor armazenado no operando esquerdo após ocorrer a atribuição.

Na atribuição simples com =, o valor da expressão substitui o do objeto referenciado pelo **lvalue**. Um dos seguintes deve ser verdade: os dois operandos têm tipo aritmético, caso em que o operando direito é convertido para o tipo da esquerda pela atribuição; ou ambos os operandos são estruturas ou uniões do mesmo tipo; ou um operando é um ponteiro, e o outro é um ponteiro para *void*; ou o operando esquerdo é um ponteiro e o operando direito é uma expressão constante com o valor 0; ou os dois operandos são ponteiros para funções ou objetos cujos tipos são os mesmos exceto pela possível ausência de *const* ou *volatile* no operando direito.

Uma expressão da forma *E1 op = E2* é equivalente a *E1 = E1 op (E2)*, exceto que *E1* é avaliado somente uma vez.

### A.7.18 Operador Vírgula

expressão:
expressão-atribuição
expressão , expressão-atribuição

Um par de expressões separadas por uma vírgula é avaliado da **esquerda para a direita**, e valor da expressão esquerda é descartado. O tipo e valor do resultado são o **tipo** e **valor** do operando direito. Todos os efeitos colaterais da avaliação do operando esquerdo são completados antes da avaliação do operando direito. Nos contextos onde a vírgula recebe significado especial, por exemplo, em listas de argumentos de função (§A.7.3.2) e listas de inicializadores (§A.8.7), a unidade sintática solicitada é uma expressão de atribuição, de forma que o operador vírgula aparece em um agrupamento entre parênteses; por exemplo,

```
f(a, (t=3, t+2), c)
```

possui três argumentos, o segundo com o valor 5.

### A.7.19 Expressões Constantes

Sintaticamente, uma expressão constante é uma expressão restrita a um subconjunto de operadores.
expressão-constante:
expressão-condicional

Expressões avaliadas para uma constante são exigidas em diversos contextos: após *case*, como limites de vetor e tamanhos de campos de bit, como o valor de uma constante de enumeração, em inicializadores, e em certas expressões de pré-processador.

As expressões constantes não podem conter atribuições, operadores de incremento ou decremento, chamadas de função, ou operadores de vírgula, exceto em um operando de *sizeof*; Se a expressão constante tiver que ser integral, seus operandos devem consistir em inteiro, enumeração, caractere e constantes de ponto flutuante; os moldes devem especificar um tipo inteiro, e qualquer constante de ponto flutuante deve ser passada para inteiro. Isso necessariamente deixa de fora as operações com vetores, indireção, endereços de objetos e com membros de estruturas. (Contudo, qualquer operando é permitido para *sizeof*.)

Uma maior latitude é permitida para as expressões constantes de inicializadores; os operandos podem ser de qualquer tipo de constante, e o operador unário & pode ser aplicado a objetos externos ou estáticos, e a vetores externos ou estáticos indexados com uma expressão constante. O operador unário & também pode ser aplicado implicitamente pelo aparecimento de vetores não-indexados e funções. Os inicializadores devem ser avaliados ou para uma constante ou para o endereço de um objeto externo ou estático previamente declarado mais ou menos uma constante.

Uma menor latitude é permitida para as expressões constantes integrais após *#if*; expressões *sizeof*, constantes de enumeração e moldes não são permitidos. Veja §A.12.5.

## A.8 Declarações

As declarações especificam a interpretação dada a cada identificador: elas não reservam necessariamente memória associada com o identificador. As declarações que reservam memória são chamadas definições. Declarações têm o formato

declaração:
espec-declaração lista-declarador-inic$_{opc}$;

Os declaradores na *lista-declarador-inic* contêm os identificadores sendo declarados; os especificadores de declaração consistem em uma sequência de especificadores de tipo e classe de memória.

especif-declaração:

especif-classe-memória especif-dec1aração$_{opc}$
especif-tipo especif-declaração$_{opc}$
qualificador-tipo especif-declaração$_{opc}$
lista-declarador-inic:
declarador-inic
lista-declarador-inic , declarador-inic

declarador-inic:
declarador
declarador = inicializador

Os declaradores serão discutidos mais adiante (§A.8.5); eles contêm os nomes sendo declarados. Uma declaração deve ter pelo menos um declarador, ou seu especificador de tipo deve declarar uma etiqueta de estrutura, uma etiqueta de união, ou os membros de uma enumeração; declarações vazias não são permitidas.

### A.8.1 Especificadores de Classe de Memória

Os especificadores de classe de memória são:

especif-classe-memória:
auto
register
static
extern
typedef

Os significados das classes de memória foram discutidos no §A.4.4.

Os especificadores *auto* e *register* dão aos objetos uma classe de memória automática, e só podem ser usados dentro de funções. Tais declarações também servem como definições e fazem com que a memória seja reservada. Uma declaração *register* é equivalente a uma declaração *auto*, mas sugere que os objetos declarados serão acessados frequentemente. Somente alguns poucos objetos serão realmente colocados em registradores, e somente certos tipos podem ser armazenados lá; as restrições dependem da implementação. Contudo, se um objeto é declarado *register*, o operador unário & não pode ser aplicado a ele, explícita ou implicitamente. O operador & é implícito quando um vetor é declarado, desta forma, declarar um vetor como *register* é um comando inútil.

A regra de que é ilegal calcular o endereço de um objeto declarado *register*, mas realmente considerado auto, é nova.

O especificador *static* dá aos objetos declarados a classe de memória *static*, e pode ser usado dentro ou fora de funções. Dentro de uma função, este especificador faz com que a memória seja alocada, e serve como uma definição; para o seu efeito fora de uma função, veja §A.11.2

Uma declaração com *extern*, usada dentro de uma função, especifica que a memória para os objetos declarados é definida em algum outro ponto; para seus efeitos fora de uma função, veja §A.11.2.

O *typedef* não reserva memória e é chamado especificador de classe de memória somente por conveniência sintática; ele é discutido em §A.8.9.

No máximo um especificador de classe de memória pode ser dado em uma declaração. Se nenhum for dado, estas regras são usadas: objetos declarados dentro de uma função são considerados *auto*; funções declaradas dentro de uma função são consideradas *extern*; objetos e funções declarados fora de uma função são considerados *static*, com ligação externa. Veja §§A.10-A.11.

### A.8.2 Especificadores de Tipo

Os especificadores de tipo são

especificador-tipo:
void
char
short
int
long
float
double
signed
unsigned
especif-estrutura-ou-união
especif-enum
nome-typedef

No máximo uma das palavras *long* ou *short* pode ser especificada juntamente com *int*; o significado é o mesmo se *int* não for mencionado. A palavra *long* pode ser especificada juntamente com *double*. No máximo uma das palavras *signed* ou *unsigned* pode ser especificada juntamente com *int* ou qualquer uma de suas variedades *short* ou *long*, ou com *char*. Qualquer um pode aparecer sozinho, sendo que neste caso *int* é subentendido. O especificador *signed* é útil para forçar objetos *char* e levar um sinal; ele é permissível mas redundante com outros tipos integrais.

Caso contrário, no máximo um especificador de tipo pode ser dado em uma declaração. Se o especificador de tipo não existir em uma declaração, ele é considerado como *int* por omissão.

Os tipos também podem ser qualificados, indicando propriedades especiais dos objetos sendo declarados.

qualificador-tipo:
const
volatile

Os qualificadores de tipo podem aparecer com qualquer especificador de tipo. Um objeto *const* pode ser inicializado, mas não atribuído depois disso. Não existe semântica independente de implementação para objetos *volatile*.

As propriedades *const* e *volatile* são novas com o padrão **ANSI**. A finalidade de *const* é anunciar objetos que possam ser colocados em memória somente de leitura, e talvez aumentar oportunidades de otimização. A finalidade de *volatile* é forçar uma implementação a suprimir a otimização que, caso contrário, poderia ocorrer. Por exemplo, para uma máquina com entrada/saída mapeada na memória, um ponteiro para um registrador de dispositivo poderia ser declarado como um ponteiro para *volatile*, evitando que o compilador aparentemente remova referências redundantes por meio do ponteiro. Exceto por ter que diagnosticar tentativas explícitas de alterar objetos *const*, um compilador pode ignorar estes qualificadores.

### A.8.3 Declarações de Estrutura e União

Uma estrutura é um objeto contendo uma sequência de membros nomeados de vários tipos. Uma união é um objeto que contém, em ocasiões diferentes, qualquer um dentre diversos membros de vários tipos. Os especificadores de estrutura e união possuem a mesma forma.

especif-estrutura-ou-união:
identificador estrutura-ou-união$_{opc}$lista-declara-estrut
indeficador estrutura-ou-união

estrutura-ou-união:

struct
union

Uma *lista-declara-estrut* é uma sequência de declarações para os membros da estrutura ou união:

lista-declara-estrut:
declaração-estrut
lista-declara-estrut declaração-estrut

declaração-estrut: lista-qualificador-espec lista-declarador-estrut
lista-qualificador-especif:
especif-tipo lista-qualificador-especif$_{opc}$
qualificador-tipo lista-qualificador-especif$_{opc}$

lista-declarador-estrut:
declarador-estrut
lista-declarador-estrut , declarador-estrut

Geralmente, um *declarador-estrut* é apenas um declarador para um membro de uma estrutura ou união. Um membro de estrutura também pode consistir em um número especificado de bits. Esse membro também é chamado *campo-bit*, ou simplesmente campo; seu tamanho é destacado do declarado para o nome do campo por meio de um sinal de dois pontos.

declarador-estrut:
declarador
declarador$_{opc}$ : expressão-constante

Um especificador de tipo da forma

identif-estrut-ou-união lista-declara-estrut

declara o *identificador* como sendo uma etiqueta da estrutura ou união especificada pela lista. Uma declaração subsequente em um escopo igual ou mais interno pode referir-se ao mesmo tipo usando a etiqueta em um especificador sem a lista:

estrut-ou-união identificador

Se um especificador com uma etiqueta mas sem uma lista aparecer quando a etiqueta não é declarada, um tipo incompleto é declarado. Os objetos com um tipo de estrutura ou união incompleto podem ser mencionados em contextos onde seu tamanho não seja necessário, por exemplo em declarações (não definições), para especificar um ponteiro, ou para criar um *typedef*, mas não em outro local. O tipo torna-se completo na ocorrência de um especificador subsequente com essa etiqueta, e contendo uma lista de declaração. Mesmo em especificadores com uma lista, o tipo da estrutura ou união sendo declarada é incompleto dentro da lista, e só se toma completo ao } terminando o especificador.

Uma estrutura não pode conter um membro de tipo incompleto. Portanto, é impossível declarar uma estrutura ou união contendo uma ocorrência de si mesma. Contudo, além de dar nome a um tipo de estrutura ou união, as etiquetas permitem a definição de estruturas auto-referenciadas; uma estrutura ou união pode conter um ponteiro para uma ocorrência de si mesma, pois os ponteiros para tipos incompletos podem ser declarados.

Uma regra muito especial aplica-se a declarações da forma

estrut-ou-união identificador ;

que declara uma estrutura ou união, mas não tem lista de declaração e nem declaradores. Mesmo que o identificador seja uma etiqueta de estrutura ou união já declarada em um escopo mais externo (§A.11.1), esta declaração torna o identificador a etiqueta de uma estrutura ou união nova, de tipo incompleto, no escopo corrente.

Esta regra oculta é nova com o **ANSI**. Ela serve para lidar com as estruturas mutuamente recursivas declaradas em um escopo mais interno, mas cujas etiquetas já poderiam estar declaradas no escopo mais externo.

Um especificador de estrutura ou união com uma lista mas sem etiqueta cria um tipo único; ele só pode ser referenciado diretamente na declaração da qual faz parte.

Os nomes dos membros e etiquetas não entram em conflito uns com os outros ou com variáveis comuns. Um nome de membro não pode aparecer duas vezes na mesma estrutura ou união, mas o mesmo nome do membro pode ser usado em diferentes estruturas ou uniões.

Na primeira edição deste livro, os nomes dos membros da estrutura e união não eram associados a seus pais. Contudo, esta associação tornou-se comum em compiladores bem antes do padrão **ANSI**.

Um membro não-campo de uma estrutura ou união pode ter qualquer tipo de objeto. Um membro de campo (que não precisa ter um declarador, e pode não ser nomeado) possui tipo *int*, *unsigned int*, ou *signed int*, e é interpretado como objeto de tipo integral do tamanho especificado nos bits; se um campo *int* é tratado como sinalizado ou não, isso depende da implementação. Os membros de campo de bit adjacentes nas estruturas são compactados em unidades de armazenamento dependentes da implementação, em uma direção que da mesma forma depende da implementação. Quando um campo logo após outro campo não cabe em uma unidade de armazenamento parcialmente preenchida, ele pode ser dividido entre duas unidades, ou então a unidade pode ser preenchida até o final. Um campo não nomeado com tamanho 0 força este preenchimento, de forma que o próximo campo comece no princípio da próxima unidade de alocação.

O padrão **ANSI** toma os campos ainda mais dependentes da implementação do que na primeira edição. Aconselha-se ler as regras de linguagem para armazenamento de campos de bit como "*dependentes da implementação*" sem qualificação. As estruturas com campos de bit podem ser usadas como uma forma portátil de se tentar reduzir o armazenamento requisitado para uma estrutura (com o custo provável de aumentar o espaço da instrução e tempo necessários para acessar os campos), ou como forma não-portátil de descrever um formato de armazenamento conhecido ao nível de bit. No segundo caso, é necessário entender as regras da implementação local.

Os membros de uma estrutura possuem endereços que crescem na ordem de suas declarações. Um membro não-campo de uma estrutura é alinhado em um limite de endereçamento dependendo de seu tipo; portanto, pode haver buracos não nomeados em uma estrutura qualquer. Se um ponteiro para uma estrutura for caracterizado para o tipo de um ponteiro para seu primeiro membro, o resultado refere-se ao primeiro membro.

Uma união pode ser pensada como uma estrutura em que todos os membros começam no deslocamento 0 e cujo tamanho é suficiente para conter qualquer um de seus membros. No máximo um dos membros pode ser armazenado de cada vez. Se um ponteiro para uma união é caracterizado para o tipo de um ponteiro para um membro, o resultado refere-se a esse membro.

Um exemplo simples de uma declaração de estrutura é

```
struct nodot{
    char palavra[20];
    int contador;
    struct nodot *esq;
    struct nodot *dir;
};
```

que contém um vetor de 20 caracteres, um inteiro, e dois ponteiros para estruturas semelhantes. Uma vez dada esta declaração, a declaração

```
struct nodot s, *ps;
```

declara *s* como sendo uma estrutura do tipo mostrado e *ps* como um ponteiro para uma estrutura desse mesmo tipo. Com estas declarações, a expressão

```
ps->contador
```

refere-se ao campo *contador* da estrutura a qual *ps* aponta;

```
s.esq
```

refere-se ao ponteiro da subárvore esquerda da estrutura *s*; e

```
s.dir->palavrat[0]
```

refere-se ao primeiro caractere do membro *palavrat* da subárvore direita de *s*.

Em geral, um membro de uma união não pode ser inspecionado a menos que o valor da união tenha sido atribuído usando-se esse mesmo membro. Contudo, uma garantia especial simplifica o uso de uniões: se uma união contém diversas estruturas que compartilham uma sequência inicial comum, e se a união atualmente contém uma dessas estruturas, pode-se referir à parte inicial comum de qualquer uma das estruturas contidas. Por exemplo, o fragmento a seguir é válido:

```
union {
    struct {
        int tipo;
    } n;
    struct {
        int tipo;
        int nodoinic;
    } ni;
    struct {
        int tipo;
        float nodoflut;
    } nf;
} u;
...
u.nf.tipo = FLOAT;
u.nf.nodoflut = 3.14;
...
if (u.n.tipo == FLOAT)
    ... sin(u.nf.nodoflut) ...
```

### A.8.4 Enumerações

Enumerações são tipos únicos com valores variando por um conjunto de constantes nomeadas chamadas enumeradores. A forma de um especificador de enumeração é semelhante àquela usada para estruturas e uniões.

especif-enum:
enum identificador$_{opc}$ lista-enumerador
enum identificador
lista-enumerador:
enumerador
lista-enumerador , enumerador

enumerador:
identificador
identificador = expressão-constante

Os identificadores em uma lista de enumerador são declarados como constantes do tipo *int*, e podem aparecer onde quer que as constantes sejam solicitadas. Se nenhum enumerador com = aparecer, então os valores das constantes correspondentes começam em 0 e aumentam de 1 à medida que a declaração é lida da esquerda para a direita. Um enumerador com = dá ao identificador associado o valor especificado; identificadores subsequentes continuam a progressão a partir do valor atribuído.

Todos os nomes de enumerador no mesmo escopo devem ser distintos um do outro e dos nomes de variáveis comuns, mas os valores não precisam ser distintos.

O papel do identificador no *especif-enum* é semelhante àquele da etiqueta de uma estrutura num *especif-estrut*; ele nomeia uma enumeração particular. As regras para enumeração com e sem etiquetas e listas são as mesmas que para especificadores de estrutura ou união, exceto que não existem tipos enumerados incompletos; a etiqueta de um *especif-enum* sem uma lista de enumerador deve se referir a um especificador com uma lista dentro do escopo.

Enumerações são novas desde a primeira edição deste livro, mas têm feito parte da linguagem por alguns anos.

### A.8.5 Declaradores

Declaradores possuem a sintaxe:

declarador:
ponteiro$_{opc}$declarador-direto

declarador-direto:
identificador
(declarador)
declarador-direto [expressão-constante$_{opc}$]
declarador-direto (lista-tipo-parâmetro)
declarador-direto (lista-identificador$_{opc}$)

ponteiro:
* lista-qualificador-tipo$_{opc}$
* lista-qualificador-tipo$_{opc}$ponteiro

lista-qualificador-tipo:
qualificador-tipo
lista-qualificador-tipo qualificador-tipo

A estrutura dos declaradores parece-se com a das expressões de indireção, função e vetor; a associação é a mesma.

### A.8.6 Significado dos Declaradores

Uma lista de declaradores aparece após uma sequência de especificadores de tipo e classe de memória. Cada declarador declara um único identificador principal, aquele que aparece como primeira alternativa da produção para *declarador-direto*. Os especificadores de classe de memória aplicam-se diretamente ao identificador, mas seu tipo depende da forma do seu declarador. Um declarador é lido como uma afirmação de que, quando seu identificador aparece em uma expressão da mesma forma que o declarador, ela produz um objeto do tipo especificado.

Considerando apenas as partes de tipo dos especificadores de declaração (§A.8.2) e um declarador em particular, uma declaração tem a forma "*T D*", onde *T* é um tipo e *D* um declarador. O

tipo atribuído ao identificador nas várias formas de declarador é descrito indutivamente usando esta notação.

Em uma declaração *T D* onde *D* é um identificador sem enfeites, o tipo do identificador é *T*.

Em uma declaração *T D* onde *D* tem a forma

(D1)

então o tipo do identificador em *D1* é o mesmo que o de *D*. Os parênteses não alteram o tipo, mas podem alterar a ligação de declaradores complexos.

### A.8.6.1 Declaradores de Ponteiro

Em uma declaração *T D* onde *D* tem a forma

* lista-qualificador-tipo$_{opc}$D1

e o tipo do identificador na declaração *T D1* é "*modificador-tipo T*", o tipo do identificador de *D* é "*modificador-tipo lista-qualificador-tipo ponteiro para T*."Os qualificadores após * aplicam-se ao próprio ponteiro, e não ao objeto para o qual ele aponta.

Por exemplo, considere a declaração

```
int *ap[];
```

Aqui *ap[]* desempenha o papel de *D1*; uma declaração "*int ap[ ]*" (a seguir) daria a *ap* o tipo "*vetor de int*", a lista de qualificador-tipo vazia, e o modificador-tipo * "*vetor de*". Daí a declaração real dar a *ap* o tipo "*vetor de ponteiros para int*".

Como em outros exemplos, as declarações

```
int i, *pi, *const cpi = \&i;

const int ci = 3, *pci;
```

declaram um inteiro *i* e um ponteiro para um inteiro *pi*. O valor do ponteiro constante *cpi* não pode ser alterado; ele sempre apontará para o mesmo local, embora o valor para o qual ele se refira possa ser alterado. O inteiro *ci* é constante, e não pode ser alterado (embora possa ser inicializado, como aqui). O tipo de *pci* é "*ponteiro para const int*", e o próprio *pci* pode ser alterado para apontar para um outro lugar, mas o valor para o qual ele aponta não pode ser alterado pela atribuição por meio de *pci*.

### A.8.6.2 Declaradores de Vetor

Em uma declaração *T D* onde *D* tem a forma

D1 [expressão-constante$_{opc}$]

e o tipo do identificador na declaração *T D1* é "*modificador-tipo T*", o tipo do identificador de *D* é "*modificador-tipo vetor de T*". Se a expressão constante estiver presente, ela deve ter tipo integral, e valor maior do que 0. Se a *expressão-constante* especificando o limite não existir, o vetor tem um tipo incompleto.

Um vetor pode ser construído a partir de um tipo aritmético, a partir de um ponteiro, a partir de uma estrutura ou união, ou a partir de um outro vetor (gerando um vetor multidimensional). Qualquer tipo do qual um vetor é construído deve ser completo; ele não deve ser um vetor ou estrutura de tipo incompleto. Isso implica que, para um vetor multidimensional, somente a primeira dimensão pode não existir. O tipo de um objeto de tipo vetor incompleto é completado por uma outra declaração, completa, para o objeto (§A.10.2), ou inicializando-o (§A.8.7). Por exemplo,

```
float fa[17], *afp[17];
```

declara um vetor de números *float* e um vetor de ponteiros para números *float*. Além disso,

```
static int x3d[3][5][7];
```

declara um vetor tridimensional de inteiros, com 3*X*5*X*7 elementos. Com detalhes completos, *x3d* é um vetor de três itens; cada item é um vetor de cinco vetores; cada um desses vetores é um vetor de sete inteiros. Qualquer uma das expressões *x3d*, *x3d[i]*, *x3d[i][j]*, *x3d[i][j][k]* pode aparecer corretamente em uma expressão. As três primeiras têm o tipo "*vetor*", a última tem o tipo *int*. Mais especificamente, *x3d[i][j]* é um vetor de 7 inteiros, e *x3d[i]* é um vetor de 5 vetores de 7 inteiros.

A operação de indexação de vetor é definida de forma que *E1[E2]* seja idêntico a **(E1+E2)*. Portanto, apesar de sua aparência assimétrica, a indexação é uma operação comutativa. Por causa das regras de conversão que se aplicam a + e aos vetores (§§A.6.6, efgeracaodeponteiro, A.7.7), se *E1* for um vetor e *E2* um inteiro, então *E1[E2]* refere-se ao E2-*ésimo* membro de *E1*.

No exemplo, *x3d[i][j][k]* é equivalente a **(x3d[i][j]) + k)*. A primeira sub-expressão *x3d[i][j]* é convertida no § efgeracaodeponteiro para o tipo "*ponteiro para vetor de inteiros*"; no §A.7.7, a adição envolve a multiplicação pelo tamanho de um inteiro. Segue-se, segundo as regras, que os vetores são armazenados por linhas (o último subscrito varia mais rapidamente) e que o primeiro subscrito na declaração ajuda a determinar a quantidade de memória consumida por um vetor, mas não tem outro papel nos cálculos de subscrito.

### A.8.6.3 Declaradores de Função

Em uma função ao novo estilo, a declaração *T D*, onde *D* tem a forma

D1 (lista-tipo-parâmetro)

e o tipo do identificador na declaração *T D1* é "*modificador-tipo T*", o tipo do identificador de *D* é "*modificador-tipo função com argumentos lista-tipo-parâmetro retornando T*."

A sintaxe dos parâmetros é

lista-tipo-parâmetro:
lista-parâmetro
lista-parâmetro, ...

lista-parâmetro:
declaração-parâmetro
lista-parâmetro, declaração-parâmetro

declaração-parâmetro:
especif-declaração declarador
especif-declaração declarador-abstrato$_{opc}$

Na declaração em novo estilo, a lista de parâmetro especifica os tipos dos parâmetros. Como caso especial, o declarador para uma função em novo estilo sem parâmetros tem uma lista de tipo de parâmetro contendo simplesmente a palavra-chave *void*. Se a lista de tipo de parâmetro terminar com reticências, "..." , então a função pode aceitar mais argumentos do que o número de parâmetros explicitamente descritos; veja §A.7.3.2.

Os tipos de parâmetros que são vetores ou funções são alterados para ponteiros, de acordo com as regras para conversões de parâmetro; veja §A.10.1. O único especificador de classe de memória permitido no especificador de declaração de um parâmetro é *register*, e este especificador é ignorado a menos que o declarador de função inicie uma definição de função. Semelhantemente, se os declaradores nas declarações de parâmetro tiverem identificadores e o declarador de função não iniciar uma definição de função, os identificadores saem do escopo imediatamente. Os declaradores abstratos, que não mencionam os identificadores, são discutidos no §A.8.8.

Uma declaração de função ao estilo antigo *T D* onde *D* tem a forma

D1 (lista-identificador$_{opc}$)

e o tipo de identificador na declaração *T D1* é "*modificador-tipo T*", o tipo do identificador de *D* é "**modificador-tipo função de argumentos não especificados retornando T*". Os parâmetros (se

estiverem presentes) têm a forma

lista-identificador:
identificador
lista-identificador, identificador

No declarador ao estilo antigo, a lista de identificador deve estar ausente a menos que o declarador seja usado no início de uma definição de função (§A.10.1).

Nenhuma informação sobre os tipos dos parâmetros é fornecida pela declaração.

Por exemplo, a declaração

```
int f (), *fpi (), (*pfi) ();
```

declara uma função *f* retornando um inteiro, uma função *fpi* retornando um ponteiro para um inteiro, e um ponteiro *pfi* para uma função retornando um inteiro. Em nenhuma delas os tipos dos parâmetros são especificados; elas são do estilo antigo.

Na declaração em estilo novo

```
int strcpy (char *dest, const char *fonte), rand(void);
```

*strcpy* é uma função retornando *int*, com dois argumentos, o primeiro um ponteiro de caractere, e o segundo um ponteiro para caracteres constantes. Os nomes de parâmetros são apenas comentários. A segunda função *rand* não usa argumentos e retorna *int*.

Os declaradores de função com protótipos de parâmetro são, de longe, a mudança de linguagem mais importante introduzida pelo padrão **ANSI**. Eles oferecem uma vantagem sobre os declaradores ao "*velho estilo*" da primeira edição por fornecer detecção de erro e coerção de argumentos através das chamadas de função, mas isso tem um preço: tumulto e confusão durante sua introdução, e a necessidade de acomodar as duas formas. Uma construção sintática mais feia foi necessária por questões de compatibilidade, a saber *void* como marcador explícito das funções ao novo estilo sem parâmetros.

A notação de reticências ", . . . " para funções variáveis também é nova, e, juntamente com as macros no cabeçalho-padrão <*stdarg.h*>, formalizam um mecanismo que era oficialmente proibido mas não oficialmente perdoado na primeira edição.

Estas notações foram adaptadas da linguagem **C++**

### A.8.7 Inicialização

Quando um objeto é declarado, seu *declarador-inic* pode especificar um valor inicial para o identificador sendo declarado. O inicializador é precedido por =, e pode ser uma expressão ou uma lista de inicializadores entre chaves. Uma lista pode terminar com uma vírgula, um enfeite para uma bela formatação.

inicializador:
expressão-atribuição
lista-inicializador
lista-inicializador,
lista-inicializador:
inicializador
lista-inicializador, inicializador

Todas as expressões no inicializador para um objeto estático ou vetor devem ser expressões constantes, como descrito no §A.7.19. As expressões no inicializador para um objeto *auto* ou *register* ou *vetor* devem também ser expressões constantes se o inicializador for uma lista delimitada por chave. Contudo, se o inicializador para um objeto automático for uma única expressão, ela não precisa ser uma expressão constante, mas simplesmente deve ter o tipo adequado para atribuição ao objeto.

A primeira edição não tolerava a inicialização de estruturas automáticas, uniões ou vetores. O padrão **ANSI** permite, mas somente para construções constantes, a não ser que o inicializador possa ser expresso por uma expressão simples.

Um objeto estático não inicializado explicitamente é inicializado como se ele (ou seus membros) tivesse recebido a constante 0. O valor inicial de um objeto automático não explicitamente inicializado é indefinido.

O inicializador para um ponteiro ou para um objeto de tipo aritmético é uma expressão simples, talvez entre chaves. A **expressão** é atribuída ao **objeto**.

O inicializador para uma estrutura é uma expressão do mesmo tipo, ou uma lista delimitada por chaves de inicializadores para seus membros, em ordem. Se houver menos inicializadores na lista do que os membros da estrutura, os membros restantes são inicializados com 0. Não pode haver mais inicializadores do que membros. Membros de campo de bit não nomeados são ignorados, e não são inicializados.

O inicializador para um vetor é uma lista, delimitada por chaves, de inicializadores para seus elementos. Se o vetor tiver tamanho desconhecido, o número de inicializadores determina o tamanho do vetor e seu tipo toma-se completo. Se o vetor tiver tamanho fixo, o número de inicializadores não pode exceder o número de elementos do vetor; se houver menos, os elementos restantes são inicializados com 0.

Como caso especial, um vetor de caracteres pode ser inicializado por meio de uma literal de string; os caracteres sucessivos da string inicializam membros sucessivos do vetor. Semelhantemente, um literal de caractere estendido (§A.2.6) pode inicializar um vetor do tipo *wchar_t*. Se o vetor tiver tamanho desconhecido, o número de caracteres na string, incluindo o caractere nulo de término, determina seu tamanho; se seu tamanho é fixo, o número de caracteres na string, não contando o caractere nulo determino, não deve exceder o tamanho do vetor.

O inicializador para uma união pode ser uma expressão simples do mesmo tipo ou um inicializador entre chaves para o primeiro membro da união.

A primeira edição não permitia inicialização de uniões. A regra do "*primeiro-membro*" é confusa; mas é difícil generalizar sem a nova sintaxe. Além de permitir que uniões sejam explicitamente inicializadas em pelo menos uma forma primitiva, esta regra **ANSI** torna definida a semântica de uniões estáticas não inicializadas explicitamente.

Um **agregado** é uma estrutura ou vetor. Se um agregado contém membros de tipo agregado, as regras de inicialização aplicam-se recursivamente. Chaves podem ser eliminadas na inicialização como segue: se o inicializador para um membro do agregado que é também um agregado começar com uma abre-chave, então a lista de inicializadores separada por vírgulas a seguir inicializa os membros do subagregado; é um erro haver mais inicializadores do que membros ou elementos. Se, no entanto, o inicializador para um subagregado não começar com um abre-chave, então somente elementos suficientes da lista são tomados para representar os membros do subagregado; quaisquer membros restantes são deixados para inicializar o próximo membro do agregado do qual o subagregado faz parte.

Por exemplo,

```
int x[] = { 1, 3, 5 };
```

declara e inicializa *x* como um vetor unidimensional com três membros, pois nenhum tamanho foi especificado e há três inicializadores.

```
float y[4l[3] = {
     { 1, 3. 5 },
     { 2, 4, 6 },
     { 3, 5, 7 },
};
```

é uma inicialização completamente delimitada por chaves: 1, 3 e 5 inicializam a primeira linha do vetor, *y[0]*, a saber *y[0][0]*, *y[0][1]* e *y[0][2]*. Da mesma forma, as próximas duas linhas inicializam *y[1]* e *y[2]*. O inicializador termina mais cedo, e portanto os elementos de *y[3]* são inicializados com 0. Exatamente o mesmo efeito poderia ser obtido por

```
float y[4][3] = {
      1, 3, 5, 2, 4, 6, 3, 5, 7
};}
```

O inicializador para *y* começa com o abre-chave, mas aquele para *y[0]* não; portanto, três elementos da lista são usados. Da mesma forma, os próximos três são usados sucessivamente por *y[1]* e depois para *y[2]*. Além disso,

```
float y[4][3] = {
      { 1 }, { 2 }, { 3 }, { 4 }
};
```

inicializa a primeira coluna de *y* (considerada como um vetor bidimensional) e deixa o restante com 0.

Finalmente,

```
char msg[] = "Erro de sintaxe na linha %s\n";
```

mostra um vetor de caracteres cujos membros são inicializados com uma string; seu tamanho inclui o caractere nulo de término.

### A.8.8 Nomes do Tipo

Em diversos contextos (para especificar conversões de tipo explicitamente com um molde, para declarar tipos de parâmetro em declaradores de função, e como argumento de *sizeof*) é necessário fornecer o nome de um tipo de dado. Isto é feito usando-se um nome de tipo, que sintaticamente é uma declaração para um objeto desse tipo omitindo o nome do objeto.

nome-tipo:
    lista-qualificador-especif declarador-abstrato$_{opc}$

declarador-abstrato:
    ponteiro
    ponteiro$_{opc}$ declarador-abstrato-direto

declarador-abstrato-direto:
    ( declarador abstrato )
    declarador-abstrato-direto$_{opc}$ [ exp-constante$_{opc}$ ]
    declarador-abstrato-direto$_{opc}$ [ lista-tipo-param$_{opc}$ ]

É possível identificar unicamente o local no declarador abstrato onde o identificador apareceria se a construção fosse um declarador em uma declaração. O tipo nomeado é então o mesmo que o tipo do identificador hipotético. Por exemplo,

```
int
int *
int *[3]
int (*)[]
int *()
int (*[]) (void)
```

nomeia respectivamente os tipos
"*inteiro*",

"*ponteiro para inteiro*",
"*vetor de 3 ponteiros para inteiros*",
"*ponteiro para um vetor de um número não especificado de inteiros*",
"*função de parâmetros não especificados retornando ponteiro para inteiro*" e
"*vetor, de tamanho não especificado, de ponteiros para funções sem parâmetros, cada um retornando um inteiro*".

### A.8.9 Typedef

Declarações cujo especificador de classe de memória é *typedef* não declaram objetos; ao invés disso, elas definem identificadores que nomeiam tipos. Estes identificadores são chamados nomes de *typedef*.

nome-typedef:
identificador

Uma declaração *typedef* atribui um tipo a cada nome entre seus declaradores na forma normal (veja §A.8.6). Depois disso, cada um desses nomes de *typedef* é sintaticamente equivalente a uma palavra-chave de especificador de tipo para o tipo associado.

Por exemplo, após

```
typedef long Nrbloco, * Ptrbloco;
typedef struct { double r, teta; } Complexo;
```

as construções

```
Nrbloco b;
extern Ptrbloco bp;
Complexo z, *zp;
```

são declarações legais. O tipo de *b* é *long*, o de *bp* é "*ponteiro para long*", e o de *z* é a estrutura especificada; *zp** é um ponteiro para tal estrutura.

*typedef* não introduz novos tipos, somente sinônimos para tipos que poderiam ser especificados de outra forma. No exemplo, *b* tem o mesmo tipo que qualquer outro objeto *long*.

Os nomes de *typedef* podem ser redeclarados em um escopo mais interno, mas um conjunto não-vazio de especificadores de tipo deve ser dado. Por exemplo,

```
extern Nrbloco;
```

não redeclara *Nrbloco*, mas

```
extern int Nrbloco;
```

sim.

### A.8.10 Equivalência de Tipo

Duas listas de especificadores de tipo são equivalentes se tiverem o mesmo conjunto de especificadores de tipo, levando em consideração que alguns especificadores podem estar implícitos através de outros (por exemplo, *long* sozinho indica *long int*). Estruturas, uniões e enumerações com diferentes etiquetas são distintos, e uma união, estrutura ou enumeração sem etiquetas especifica um tipo único.

Dois tipos são iguais se seus declaradores abstratos (§A.8.8), após expandir quaisquer tipos *typedef*, e deletar quaisquer identificadores de parâmetro de função, são iguais até a equivalência das listas de especificador de tipo. Os tamanhos de vetor e tipos de parâmetro de função são significativos.

## A.9 Comandos

Exceto quando descrito, os comandos são executados em sequência. Os comandos são executados pelo seu efeito, e não possuem valores. Eles se agrupam em diversos grupos.

comando:
comando-rotulado
comando-expressão
comando-composto
comando-seleção
comando-iteração
comando-salto

### A.9.1 Comandos Rotulados

Comandos podem conter prefixos de rótulo.

comando-rotulado:
identificador: comando
case expressão-constante: comando
default: comando

Um rótulo contendo um identificador, declara o identificador. O único uso de um rótulo de identificador é como destino de *goto*. O escopo do identificador é a função corrente. Como os rótulos possuem seu próprio espaço de nome, eles não interferem com outros identificadores e não podem ser redeclarados. Veja §A.11.1.

Os rótulos de caso e omissão ("*case*" e "*default*") são usados com o comando *switch* (§A.9.4). A expressão constante de *case* deve ter tipo inteira.

Os rótulos por si mesmos não alteram o fluxo de controle.

### A.9.2 Comando de Expressão

A maior parte dos comandos são comandos de expressão, que têm a forma

comando-expressão:
$\text{expressão}_{\text{opc}}$;

A maior parte dos comandos de expressão são atribuições ou chamadas de função. Todos os efeitos colaterais da expressão são completados antes que o próximo comando seja executado. Se a expressão estiver faltando, a construção é chamada **comando nulo**; ele é normalmente usado para fornecer um corpo vazio para um comando de iteração ou para colocar um rótulo.

### A.9.3 Comando Composto

Para que vários comandos possam ser usados onde um é esperado, o comando composto (também chamado "*bloco*") tem sua utilidade. O corpo de uma definição de função é um comando composto.

comando-composto:
$\text{lista-declaração}_{\text{opc}}$ $\text{lista-comando}_{\text{opc}}$
lista-declaração:
declaração
lista-declaração declaração
lista-comando:
comando
lista-comando comando

Se um identificador na lista declaração estivesse no escopo fora do bloco, a declaração mais externa seria suspensa dentro do bloco (veja §A.11.1), após o que continuaria com sua força. Um identificador só pode ser declarado uma vez no mesmo bloco. Estas regras aplicam-se a

identificadores no mesmo espaço de nome (§A.11); identificadores em diferentes espaços de nome são tratados como distintos.

A inicialização de objetos automáticos é executada toda vez que o bloco é entrado no seu topo, e prossegue na ordem das declarações. Se um salto para o bloco for executado, estas inicializações não são executadas. As inicializações de objetos *static* são executadas somente uma vez, antes que o programa comece sua execução.

### A.9.4 Comandos de Seleção

Os comandos de seleção escolhem um dentre diversos fluxos de controle.

comando-seleção:
if (expressão) comando
if (expressão) comando else comando
switch (expressão) comando

Nas duas formas do comando *if*, a expressão, que deve ter tipo aritmético ou ponteiro, é avaliada, incluindo todos os efeitos colaterais, e se for comparada diferente de 0, o primeiro subcomando é executado. Na segunda forma, o segundo subcomando é executado se a expressão resultar em 0. A ambiguidade do *else* é resolvida conectando-se um *else* com o último *if* sem *else* encontrado ao mesmo nível de indentação de bloco.

O comando *switch* faz com que o controle seja transferido para um dentre diversos comandos dependendo do valor de uma expressão, que deve ter tipo inteiro. O subcomando controlado por um *switch* é tipicamente composto. Qualquer comando dentro do subcomando pode ser rotulado com um ou mais rótulos de caso (§A.9.1). A expressão controladora passa por promoção integral (§A.6.1), e as constantes de caso são convertidas para o tipo promovido. Duas constantes de caso associadas com o mesmo *switch* não podem ter o mesmo valor depois da conversão. Só pode haver no máximo um rótulo *default* associado a um *switch*. Os *switch*es podem ser indentados; um rótulo de case ou *default* está associado ao menor *switch* que o contém.

Quando o comando *switch* é executado, sua expressão é avaliada, incluindo todos os efeitos colaterais, e comparada com cada constante de caso. Se uma das constantes for igual ao valor da expressão, o controle passa para o comando do rótulo de *case* que combinou. Se nenhuma constante de caso combinar com a expressão, e se houver um rótulo *default*, o controle passa para o comando rotulado. Se não houver combinação e nem *default*, então nenhum dos subcomandos do *switch* é executado.

Na primeira edição deste livro, a expressão controladora do *switch*, além das constantes de caso, deveriam ter o tipo *int*.

### A.9.5 Comandos de Iteração

Os comandos de iteração especificam laços.

comando-iteração:
while (expressão) comando
do comando while (expressão);
for (expressão$_{opc}$; expressão$_{opc}$ ; expressão$_{opc}$ ) comando

Nos comandos *while* e *do*, o subcomando é executado repetidamente enquanto o valor da expressão permanecer diferente de 0; a expressão deve ter tipo aritmético ou ponteiro. Com *while*, o teste, incluindo todos os efeitos colaterais da expressão, ocorre antes de cada execução do comando; com *do* o teste vem após cada iteração.

No comando *for*, a primeira expressão é avaliada uma vez, e assim especifica a inicialização para o laço. Não existe restrição para seu tipo. A segunda expressão deve ter tipo aritmético ou ponteiro; ela é avaliada antes de cada iteração, e se passar a ser igual a 0, o comando *for* é terminado. A terceira expressão é avaliada após cada iteração, especificando assim a reinicialização para o

laço. Não há restrição para o seu tipo. Os efeitos colaterais de cada expressão são completados imediatamente após sua avaliação. Se o subcomando não contém continue, um comando

```
for (expressão1 ; expressão2; expressão3) comando
```

é equivalente a

```
expressão1;
while (expressão2){
  comando
  expressão3;
}
```

Qualquer uma das três expressões pode ser omitida. Uma segunda expressão faltando torna o teste implícito equivalente ao teste de uma constante diferente de zero.

### A.9.6 Comandos de Salto

Os comandos de salto transferem o controle incondicionalmente.

comando-salto:
goto identificador;
continue;
break;
return expressão$_{opc}$;

No comando *goto*, o identificador deve ser um rótulo (§A.9.1) localizado na função corrente. O controle é transferido para o comando rotulado;

Um comando *continue* só pode aparecer dentro de um comando de iteração. Ele faz com que o controle passe para a parte de continuação do laço do comando de iteração menos abrangente. Mais precisamente, dentro de cada um dos comandos

```
while (...){     do {              for (...) {
  ...              ...               ...
continua: ;      continua: ;       continua: ;
}                } while (...);     }
```

um *continue* (não contido em um comando de iteração menor) é o mesmo que *goto continua*.

Um comando *break* só pode aparecer em um comando de iteração ou em um comando *switch*, e termina a execução do comando de iteração menos abrangente; o controle passa para o comando após o comando terminado.

Uma função retorna para a chamadora por meio do comando *return*. Quando *return* é seguido por uma expressão, o valor é retornado para a chamadora de função. A expressão é convertida, da mesma forma que na atribuição, ao tipo retornado pela função em que ela aparece.

Continuar a execução até o fim de uma função é equivalente a retornar sem expressão. De qualquer forma, o valor retornado é indefinido.

## A.10 Declarações Externas

A unidade de entrada fornecida ao compilador **C** é chamada unidade de tradução; ela consiste em uma sequência de declarações externas, que podem ser declarações ou definições de função.

unidade-tradução:
declaração-externa
unidade-tradução declaração-externa
declaração-externa:
definição-função
declaração

O escopo das declarações externas persiste até o final da unidade de tradução em que são declaradas, assim como o efeito das declarações dentro dos blocos persiste até o final do bloco. A sintaxe das declarações externas é a mesma que a de todas as declarações, exceto que somente este nível pode ser dado o código para as funções.

### A.10.1 Definições de Função

As definições de função têm a forma

definição-função:

espec-função$_{opc}$ declarador lista-declarador$_{opc}$ comando-composto

Os únicos especificadores da classe de memória permitidos entre os especificadores de declaração são *extern* ou *static*; veja §A.11.2 para obter a distinção entre eles.

Uma função pode retornar um tipo aritmético, uma estrutura, uma união, um ponteiro ou *void*, não uma função ou um vetor. O declarador em uma declaração de função deve especificar explicitamente que o identificador declarado tem tipo função; ou seja, ele deve conter uma das formas (veja §A.8.6.3)

declarador-direto (lista-tipo-parâmetro)

declarador-direto (lista-identificador$_{opc}$)

onde o declarador direto é um identificador ou um identificador entre parênteses. Em particular, ele não deve conseguir o tipo função por meio de um *typedef*.

Na primeira forma, a definição é uma função ao novo estilo, e seus parâmetros, juntamente com seus tipos, são declarados na sua lista de tipo de parâmetro; a *lista-declaração* após o declarador de função deve estar ausente. A menos que a lista de tipo de parâmetro contenha somente *void*, mostrando que a função não utiliza parâmetros, cada declarador na lista de tipo de parâmetro dever conter um identificador. Se a lista de tipo de parâmetro terminar com ",..." então a função pode ser chamada com mais argumentos do que parâmetros; o mecanismo de macro *va_arg*, definido no cabeçalho-padrão <stdarg.h> e descrito no **Apêndice B**, deve ser usado para se referir aos argumentos extras. As funções variáveis devem ter pelo menos um parâmetro nomeado.

Na segunda forma, a definição está no estilo antigo: a lista de identificador nomeia os parâmetros, enquanto os atributos da lista de declaração os caracteriza. Se nenhuma declaração for dada para um parâmetro, seu tipo é considerado *int*. A lista de declaração deve declarar apenas parâmetros nomeados na lista, a inicialização não é permitida, e o único especificador de classe de memória possível é *register*.

Nos dois estilos de definição de função, os parâmetros são entendidos como sendo declarados após o início do comando composto constituindo o corpo da função, e assim os mesmos identificadores não devem ser redeclarados lá (embora possam, como outros identificadores, ser redeclarados nos blocos mais internos). Se um parâmetro é declarado para ter o tipo "*vetor de tipo*", a declaração é ajustada para que leia "*ponteiro para tipo*", semelhantemente, se um parâmetro for declarado como sendo do tipo "*função retornando tipo*", a declaração é ajustada para que se leia "*ponteiro para função retornando tipo*". Durante a chamada para uma função, os argumentos são convertidos quando necessários e atribuídos aos parâmetros: veja §A.7.3.2.

As definições ao novo estilo são novas com o padrão **ANSI**. Há também uma pequena mudança nos detalhes de promoção; a primeira edição especificava que as declarações de parâmetros *float* eram ajustadas para que se tornem *double*. A diferença torna-se observável quando um ponteiro para um parâmetro é gerado dentro de uma função.

Um exemplo completo de uma definição de função ao novo estilo poderia ser

```
int max(int a, int b, int c)
{
  int m;
```

```
    m = (a > b) ? a : b;
    return (m > c) ? m : c;
}
```

Aqui, *int* é o especificador da declaração; *max (int a , int b, int c)* é o declarador da função, e ... é o bloco dando o código para a função. A definição correspondente em estilo antigo seria

```
int max (a, b, c)
int a, b, c;
{
  /*...*/
}
```

onde agora *int max (a, b, c)* é o declarador, e *int a, b, c;* é a lista de declaração para os parâmetros.

### A.10.2 Declarações Externas

As declarações externas especificam as características de objetos, funções e outros identificadores. O termo "*externa*" refere-se ao seu local fora das funções, e não está ligado diretamente à palavra-chave *extern*; a classe de memória para um objeto declarado externamente pode ficar vazia, ou então pode ser especificada como *extern* ou *static*.

Diversas declarações externas para o mesmo identificador podem existir dentro da mesma unidade de tradução se concordarem em tipo e ligação, e se houver no máximo uma definição para o identificador.

Duas declarações para um objeto ou função supostamente combinam em tipo sob as regras discutidas no §A.8.10. Além disso, se as declarações diferirem porque um tipo é uma estrutura, união ou enumeração incompleta (§A.8.3) e o outro é o tipo completo correspondente, com a mesma etiqueta, os tipos irão combinar. Além do mais, se um tipo for um tipo vetor incompleto (§A.8.6.2) e o outro é um tipo vetor completo, os tipos, se quanto aos mais idênticos, também combinam. finalmente, se um tipo especificar uma função em estilo antigo, e o outro uma função ao novo estilo quanto ao mais idêntico, com declarações de parâmetro, os tipos são levados a combinar.

Se a primeira declaração externa para uma função ou objeto incluir o especificador *static*, o identificador tem ligação interna; caso contrário, ele tem ligação externa. A ligação é discutida no §A.11.2.

Uma declaração externa para um objeto é uma definição se tiver um inicializador. Uma declaração de objeto externo que não tem um inicializador, e não contêm o especificador *extern*, é uma definição tentativa. Se uma definição para um objeto aparecer em uma unidade de tradução, quaisquer definições de tentativa são tratadas simplesmente como declarações redundantes. Se nenhuma definição para o objeto aparecer na unidade de tradução, todas as suas definições tentativas tornam-se uma única definição com inicializador 0.

Cada objeto deve ter exatamente uma definição. Para objetos com ligação interna, esta regra aplica-se separadamente a cada unidade de tradução, pois os objetos ligados internamente são únicos a uma unidade de tradução. Para objetos com ligação externa, ela aplica-se ao programa inteiro.

Embora a regra de uma definição tenha sido formulada um pouco diferente na primeira edição deste livro, ela é idêntica ao que afirmamos aqui. Algumas implementações aliviam-na generalizando a noção de definição tentativa. Na formulação alternativa, que é normal em sistemas **UNIX** e reconhecida como extensão comum pelo Padrão, todas as definições tentativas para um objeto ligado externamente, em todas as unidades de tradução de um programa, são consideradas juntamente ao invés de separadamente em cada unidade de tradução. Se uma definição ocorre em algum outro local do programa, então as definições tentativas tornam-se simplesmente declarações, mas se não aparecer uma definição, então todas as suas definições tentativas tornam-se uma definição com inicializador 0.

## A.11 Escopo e Ligação

Um programa não precisa ser todo compilado de uma vez: o texto-fonte pode ser mantido em diversos arquivos contendo unidades de tradução, e rotinas pré-compiladas podem ser carregadas de bibliotecas. A comunicação entre as funções de um programa pode ser executada tanto por chamadas quanto pela manipulação de dados externos.

Portanto, há dois tipos de escopo a considerar: primeiro, o escopo léxico de um identificador, que é a região do texto do programa dentro da qual as características do identificador são conhecidas; e segundo, o escopo associado com objetos e funções com ligação externa, que determina as conexões entre identificadores em unidades de tradução compiladas separadamente.

### A.11.1 Escopo Léxico

Identificadores caem em diversos espaços de nome que não interferem um com o outro; o mesmo identificador pode ser usado para diferentes finalidades, até no mesmo escopo, se os usos estiverem em espaços de nome diferentes. Estas classes são: objeto, funções, nomes de *typedef* e constantes *enum*; rótulos; etiquetas de estruturas, uniões e enumerações; e membros de cada estrutura ou união individualmente.

Estas regras diferem em formas gerais daquelas descritas na primeira edição deste manual. Os rótulos não tinham previamente seu próprio espaço de nome; as etiquetas das estruturas e uniões tinham espaços separados, e em algumas implementações as etiquetas de enumeração também tinham; colocar diferentes tipos de etiquetas no mesmo espaço é uma nova restrição. A diferença mais importante da primeira edição é que cada estrutura ou união cria um espaço de nome separado para seus membros, de forma que o mesmo nome possa aparecer em diversas estruturas diferentes. Esta regra tem sido uma prática comum vários anos.

O escopo léxico de um identificador de objeto ou função em uma declaração externa começa no fim de seu declarador e persiste até o final da unidade de tradução em que ele aparece. O escopo de um parâmetro de uma definição de função começa no início do bloco definindo a função, e continua por toda a função; o escopo de um parâmetro em uma declaração de função termina ao final do declarador. O escopo de um identificador declarado no início de um bloco começa ao final de seu declarador e continua até o final do bloco. O escopo de um rótulo é a função inteira em que ele aparece. O escopo de uma estrutura, união, etiqueta de enumeração ou constante de enumeração, começa no seu aparecimento em um especificador de tipo, e continua até o fim da unidade de tradução (para declarações dentro de uma função).

Se um identificador é explicitamente declarado no início de um bloco, incluindo o bloco constituindo uma função, qualquer declaração do identificador fora do bloco é suspensa até o final do bloco.

### A.11.2 Ligação

Dentro de uma unidade de tradução, todas as declarações do mesmo identificador de objeto ou função com ligação interna referem-se à mesma coisa, e o objeto ou função é único nessa unidade de tradução. Todas as declarações para o mesmo identificador de objeto ou função com ligação externa referem-se à mesma coisa, e o objeto ou função é compartilhado pelo programa inteiro.

Conforme discutimos no §A.10.2, a primeira declaração externa para um identificador dá ao identificador ligação interna se o especificador *static* for usado, caso contrário, ligação externa. Se uma declaração para um identificador dentro de um bloco não inclui o especificador *extern*, então o identificador não tem ligação, e é único à função. Se ele inclui *extern*, e uma declaração externa para o identificador estiver ativa no escopo rodeando o bloco, então o identificador tem a mesma ligação que a declaração externa, e refere-se ao mesmo objeto ou função; mas se nenhuma declaração externa for visível, sua ligação é externa.

## A.12 Pré-Processamento

Um pré-processador executa substituição de macro, compilação condicional e inclusão de arquivos nomeados. As linhas começando com #, talvez precedido por um espaço em branco, comunicam-se com o pré-processador. A sintaxe destas linhas é independente do restante da linguagem; elas podem aparecer em qualquer lugar e ter um efeito que dura (independente do escopo) até o fim da unidade de tradução. Os limites de linha são significativos; cada linha é analisada individualmente (mas veja como juntar linhas no §A.12.2). Para o pré-processador, um código é qualquer código da linguagem, ou uma sequência de caracteres dando um nome de arquivo como na diretiva #include (§A.12.4); além do mais, qualquer caractere não definido de outra forma é considerado como um código. Contudo o efeito dos caracteres de espaço diferentes do espaço em branco e tabulação horizontal é indefinido dentro das linhas do pré-processador.

O próprio pré-processamento ocorre em diversas fases logicamente sucessivas que podem, numa implementação particular, ser condensadas.

1. Primeiro, sequências de trigrama, descritas em A.12.1, são substituídas por seus equivalentes. Se o sistema operacional exigir, caracteres de nova-linha são introduzidos entre as linhas do arquivo-fonte.
2. Cada ocorrência de um caractere de contrabarra \ seguido por uma nova-linha é deletada, separando as linhas (§A.12.2).
3. O programa é dividido em códigos separados por caracteres de espaço em branco; comentários são substituídos por um único espaço. Depois as diretivas de pré-processamento são obedecidas, e macros (§A.12.3-§A.12.10) são expandidas.
4. As sequências de escape em constantes de caracteres e literais de string (§A.2.5.2, §A.2.6) são substituídas por seus equivalentes; depois as literais de string adjacentes são concatenadas.
5. O resultado é traduzido, depois ligado com outros programas e bibliotecas, coletando os programas e dados necessários, e conectando referências de função e objetos externos às suas definições.

### A.12.1 Sequências de Trigrama

O conjunto de caracteres dos programas-fonte em **C** está contido dentro do conjunto **ASCII** de sete bits, mas é um superconjunto do Conjunto de Código Invariante **ISO 646-1983**. Para permitir que programas sejam representados no conjunto reduzido, todas as ocorrências, das seguintes sequências de trigrama são substituídas pelo caractere isolado correspondente. Esta substituição ocorre antes de qualquer outro processamento.

| | | | | | |
|---|---|---|---|---|---|
| ??= | # | ??( | [ | ??< | { |
| ??/ | \ | ??) | ] | ??> | } |
| ??' | ^ | ??! | \| | ??- | ~ |

Tabela A.3: Sequências de trigramas.

Nenhuma outra substituição desse tipo ocorre.
As sequências de trigrama são novas com o padrão **ANSI**.

### A.12.2 Divisão de Linha

As linhas que terminam com o caractere de contrabarra \ são separadas deletando-se a **contrabarra** e o caractere de nova-linha seguinte. Isso ocorre antes da divisão em códigos.

### A.12.3 Definição e Expansão de Macro

Uma linha de controle da forma

```
#define identificador sequência-de-código
```

faz com que o pré-processador substitua as ocorrências subsequentes do identificador pela sequência de códigos indicada; os espaços iniciais e finais ao redor da sequência são desconsiderados. Um segundo `#define` para o mesmo identificador é errôneo, a não ser que a segunda sequência de código seja idêntica à primeira, onde todas as separações de espaço em branco são consideradas equivalentes. A lista de argumentos é opcional (pode ser vazia) em definiçoes de macros com parênteses.

Uma linha da forma

```
#define identif(lista-identificadores) sequência-código
```

onde não existe espaço entre o primeiro identificado e o (, é uma definição de macro com parâmetros dados pela lista de identificador. Assim como a primeira forma, os espaços ao redor da sequência de código são descartados, e a macro só pode ser redefinida com uma definição em que o número e caracteres dos parâmetros, e a sequência de código, sejam idênticos.

Uma linha de controle da forma

```
#undef identificador
```

faz com que a definição de pré-processador de um identificador seja esquecida. Não é errado aplicar `#undef` a um identificador desconhecido.

Quando uma macro tiver sido definida na segunda forma, as ocorrências textuais subsequentes do identificador de macro seguidas pelo espaço em branco opcional e depois por (, uma sequência de códigos separados por vírgulas, e um ) constituem uma chamada da macro. Os argumentos da chamada são sequências de código separadas por vírgula; as vírgulas entre aspas ou protegidas por parênteses indentados não separam argumentos. Durante a coleção, os argumentos não são expandidos por macros. O número de argumentos na chamada deve combinar com o número de parâmetros na definição. Após serem isolados os argumentos, os espaços em branco não necessários são removidos deles. Depois a sequência de código resultante de cada argumento é substituída para cada ocorrência não entre aspas do identificador do parâmetro correspondente na sequência de substituição da macro. A não ser que o parâmetro na sequência de substituição seja precedido por #, ou precedido ou seguido por ##, os códigos do argumento são examinados para chamadas de macro, e expandidos quando necessário, logo antes da inserção.

Dois operadores especiais influenciam o processo de substituição. Primeiro, se uma ocorrência de um parâmetro na sequência de substituição é imediatamente precedida por #, as aspas de string (“) são colocadas ao redor do parâmetro correspondente, e depois tanto o # quanto o identificador de parâmetro são substituídos pelo argumento entre aspas. Um caractere \ é inserido antes de cada caractere “ ou \ que aparece ao redor, ou dentro de uma literal de string, ou constante de caractere no argumento.

Segundo, se a sequência do código de definição para qualquer tipo de macro tiver um operador ##, então logo após a substituição dos parâmetros, cada ## é deletado, juntamente com qualquer espaço em branco de um lado ou de outro, conseguindo concatenar os códigos adjacentes e formar um novo código. O efeito é indefinido se códigos inválidos forem produzidos, ou se o resultado depender da ordem de processamento dos operadores ##. Além disso, ## não pode aparecer no início ou fim de uma sequência de código de substituição.

Nos dois tipos de macro, a sequência de código de substituição é repetidamente reanalisada para identificadores mais definidos. Contudo, quando um certo identificador tiver sido substituído em uma determinada expansão, ele não é substituído se aparecer novamente durante a reanálise; ao invés disso, é deixado sem alteração.

Mesmo que o valor final de uma expansão de macro comece com #, ele não é considerado como uma diretiva de pré-processamento.

Os detalhes do processo de expansão de macro são descritos com mais precisão no padrão **ANSI** do que na primeira edição. A mudança mais importante é o acréscimo dos operadores # e ## , que torna admissível a concatenação e uso de aspas. Algumas das novas regras, especialmente aquelas envolvendo a concatenação, são bizarras. (Veja os exemplos a seguir.)

Por exemplo, esta facilidade pode ser usada para "constantes de manifesto", como em

```
#define TAMTAB 100
int tabela [TAMTAB];
```

A definição

```
#define DIFABS(a, b)  ((a)>(b) ? (a)-(b) : (b)-(a))
```

define uma macro para retornar o valor absoluto da diferença entre seus argumentos. Diferente de uma função realizando a mesma coisa, os argumentos e o valor retornado podem ter qualquer tipo aritmética ou podem até mesmo ser ponteiros. Além disso, os argumentos, que podem ter efeitos colaterais, são avaliados duas vezes, uma para o teste e uma para produzir o valor.

Dada a definição

```
#define arqtemp(dir)  #dir "%s"
```

a chamada de macro *arqtemp(/usr/tmp)* produz

```
"/usr/tmp" "%s"
```

que mais tarde será concatenada para uma única string. Depois de

```
#define cat(x, y) x##y
```

a chamada *cat(var, 123)* produz *var123*. Entretanto, a chamada a *cat(cat(1,2),3)* é indefinida: a presença de ## evita que os argumentos da chamada mais externa sejam expandidos. Assim, ela produz a string de códigos

```
cat( 1 , 2 )3
```

e )3 (a inclusão do último código do primeiro argumento com o primeiro código do segundo) não é válido. Se for introduzido um segundo nível de definição de macro,

```
#define xcat(x, y)   cat(x,y)
```

as coisas acontecem mais naturalmente; *xcat(xcat(1, 2), 3)* produz 123, pois a expansão do próprio *xcat* não envolve o operador ##.

De forma semelhante, *DIFABS(DIFABS(a, b), c)* produz o resultado esperado, totalmente expandido.

### A.12.4 Inclusão de Arquivo

Uma linha de controle da forma

```
#include <arquivo>
```

causa a substituição dessa linha pelo conteúdo inteiro de arquivo. Os caracteres no nome arquivo não devem incluir > ou nova-linha, e o efeito é indefinido se houver ", ', \ ou /*. O arquivo nomeado é procurado em uma sequência de lugares que depende da implementação.

Semelhantemente, uma linha de controle da forma

```
#include "arquivo"
```

procura primeiro em associação com o arquivo-fonte original (uma frase que deliberadamente depende da implementação), e se essa busca falhar, então a pesquisa é feita como na primeira forma. O efeito de usar ', \ ou /* no nome de arquivo permanece indefinido, mas > é permitido.

Finalmente, uma diretiva da forma

```
#include seg-código
```

não combinando com uma das formas anteriores é interpretada expandindo-se a sequência de código como para um texto normal; uma das duas formas com <... > ou "..."deve ter um resultado, sendo em seguida tratada conforme descrevemos.

Os arquivos `#include` podem ser indentados.

### A.12.5 Compilação Condicional

Partes de um programa podem ser compiladas condicionalmente, de acordo com a seguinte sintaxe esquemática. preprocessor-conditional:

condicional-pré-processador:
linha-if texto partes-elif$_{opc}$ parte-else$_{opc}$ `#endif`
linha-if:
`# if` expressão-constante
`# ifdef` identificador
`# ifndef` identificador
partes-elif:
linha-elif texto partes-elif$_{opc}$
linha-elif:
`# elif` expressão-constante
parte-else:
linha-else texto
linha-else:
`#else`

Cada uma das diretivas (*linha-if*, *linha-elif*, *linha-else* e *#endif*) aparece isoladamente em uma linha. As expressões constantes em *#if* e subsequentes linhas *#elif* são avaliadas em ordem até que seja encontrada uma expressão com um valor diferente de zero; o texto após uma linha com valor zero é desconsiderado. O texto após a linha de diretiva bem-sucedida é tratado normalmente. "*Texto*" aqui refere-se a qualquer material, incluindo linhas de pré-processador, que não seja parte da estrutura condicional; ele pode estar vazio. Uma vez encontrada uma linha *#if* ou *#elif* bem-sucedida e processado seu texto, outras linhas *#elif* e *#else*, juntamente com seu texto, são desconsideradas para a compilação. Se todas as expressões forem zero, e se houver um *#else*, o texto após o *#else* é tratado normalmente. O texto controlado por braços inativos da condicional é ignorado, exceto para checar a indentação de condicionais.

A expressão constante em *#if* e *#elif* está sujeita à substituição normal de macro. Além do mais, quaisquer expressões da forma

`defined` identificador

ou

`defined` (identificador)

são substituídas, antes de procurar macros, por 1 se o identificador estiver definido no pré-processador, e por 0 se não. Quaisquer identificadores restantes após a expansão de macro são substituídos por 0. Finalmente, cada constante inteira é considerada como tendo o sufixo *L*, de forma que toda a aritmética é considerada como *long* ou *unsigned long*.

A expressão constante obtida (§A.7.19) é restrita: ela deve ser integral, e não pode conter *sizeof*, um molde ou uma constante de enumeração.

As linhas de controle

```
#ifdef identificador
#ifndef identificador
```

são equivalentes a

```
#if defined identificador
#if !defined identificador
```

respectivamente.

*#endif* é novo desde a primeira edição, embora tenha estado disponível em alguns pré-processadores. O operador *defined* do pré-processador também é novo.

### A.12.6 Controle de Linha

Para o benefício de outros pré-processadores que geram programas em **C**, uma linha com uma das formas

```
#line constante "arquivo"
#line constante
```

faz com que o compilador acredite, para fins de diagnóstico de erro, que o número de linha da próxima linha fonte seja dado pela constante decimal inteira e o arquivo de entrada corrente seja nomeado pelo identificador. Se o arquivo entre aspas não existir, o nome registrado não se altera. As macros na linha são expandidas antes que isto seja interpretado.

### A.12.7 Geração de Erro

Uma linha de pré-processador da forma

```
#error seg-códigoopc
```

faz com que o processador escreva uma mensagem de diagnóstico que inclua a sequência de código.

### A.12.8 Pragmas

Uma linha de controle da forma

```
#pragma seg-códigoopc
```

faz com que o processador execute uma ação dependente da implementação. Uma pragma não reconhecida é ignorada.

### A.12.9 Diretiva Nula

Uma linha de pré-processador da forma

```
#
```

não tem efeito.

### A.12.10 Nomes Predefinidos

Diversos identificadores são predefinidos e expandidos para produzir informações especiais. Eles, juntamente com o operador de expressão do pré-processador *defined*, não podem ser indefinidos ou redefinidos.

**__LINE__** Uma constante decimal contendo o número de linha atual da fonte.
**__FILE__** Uma literal de string contendo o nome do arquivo sendo compilado.

**__DATE__** Uma literal de string contendo a data da compilação, na forma "*Mmmm dd aaaa*".
**__TIME__** Uma literal de string contendo a hora da compilação, na forma "hh:mm:ss".
**__STDC__** A constante 1. Entende-se que este identificador seja definido como 1 somente em implementações que estejam de acordo com o padrão.

*#error* e *#pragma* são novos no padrão **ANSI**; as macros predefinidas do pré-processador são novas, mas algumas delas já existem em algumas implementações.

## A.13 Gramática

A seguir vemos uma recapitulação da gramática que foi dada na primeira parte deste apêndice. Ela possui um conteúdo semelhante, mas em uma ordem diferente.

A gramática possui os símbolos terminais indefinidos *const-inteira*, *const-caractere*, *const-flutuante*, *identif*, *string* e *const-enumeração*; as palavras e símbolos mostrados em estilo destacado são terminais dados literalmente. Esta gramática pode ser transformada mecanicamente em entrada aceitável para um analisador-gerador automático. Além de incluir as marcações sintáticas usadas para indicar alternativas na produção, é necessário expandir as construções "*um dentre*", e (dependendo das regras do analisador-gerador) duplicar cada produção com um símbolo *opc*, uma vez com o símbolo e outra sem ele. Com mais uma mudança, a saber deletando a produção *nome-typedef*: *identificador*, e tornando *nome-typedef* um símbolo terminal, esta gramática é aceitável para o analisador-gerador **YACC**. Ele tem somente um conflito, gerado pela ambiguidade *if-else*.

*unidade-tradução:*
*declaração-externa*
*unidade-tradução declaração-externa*

*declaração-externa:*
*definição-função*
*declaração*

*definição-função:*
*especif-declaração*$_{opc}$ *declarador lista-declaração*$_{opc}$ *comando-composto*

*declaração:*
*especif-declaração lista-declarador-inic*$_{opc}$*;*

*lista-declaração:*
*declaração*
*lista-declaração declaração*

*especif-declaração:*
*especif-classe-armazenamento especif-declaração*$_{opc}$
*especif-tipo especif-declaração*$_{opc}$
*qualif-tipo especif-declaração*$_{opc}$

*especif-classe-armazenamento: um dentre*
`auto register static extern typedef`

*especif-tipo: um dentre*
`void char short int long float double signed unsigned`
*especif-estrut-ou-união especif-enum nome-typedef*

*qualif-tipo: um dentre*
`const volatile`

*especif-estrut-ou-união:*
*estrut-ou-união identificador*$_{opc}$ *{ lista-declaração-estrut }*
*strut-ou-união identificador*

*strut-ou-union: um dentre*
`struct union`

*lista-declaração-estrut:*
*declaração-estruturura*
*lista-declaração-estrut declaração-estrutura*

*lista-declarador-inic:*
*declarador-inic*
*lista-declarador-inic, declarador-inic*

*declarador-inic:*
*declarador*
*declarador = iniciador*

*declaração-estrutura:*
*lista-qualif-especif lista-declarador-estrut;*

*lista-qualif-especif:*
*especif-tipo lista-qualif-especif*$_{opc}$
*qualif-tipo lista-qualif-especif*$_{opc}$

*lista-declarador-estrut:*
*declarador-estrut*
*lista-declarador-estrut, declarador-estrut*

*declarador-estrut:*
*declarador*
*declarador*$_{opc}$ *: expressão-constante*

*especif-enum:*
`enum` *identificador*$_{opc}$ *{ lista-enumerador }*
`enum` *identificador*

*lista-enumerador:*
*enumerator*
*lista-enumerador , enumerador*

*enumerador:*
*identificador*
*identificador = expressão-constante*

*declarador:*
*ponteiro*$_{opc}$ *declarador-direto*

*declarador-direto:*
*identificador*
*(declarador)*
*declarador-direto [ expressão-constante*$_{opc}$ *]*
*declarador-direto ( lista-tipo-parâmetro )*
*declarador-direto ( lista-identificador*$_{opc}$ *)*

*ponteiro:*
** lista-qualif-tipo*$_{opc}$
** lista-qualif-tipo*$_{opc}$ *ponteiro*

*lista-qualif-tipo:*
*qualif-tipo*
*lista-qualif-tipo qualif-tipo*

*lista-tipo-parâmetro:*
*lista-parâmetro*
*lista-parâmetro , . . .*

*lista-parâmetro:*
*declaração-parâmetro*
*lista-parâmetro , declaração-parâmetro*

*declaração-parâmetro:*
*especif-declaração declarador*
*especif-declaração declarador-abstrato*$_{opc}$

*lista-identificador:*
*identificador*
*lista-identificador , identificador*

*iniciador:*
*expressão-atribuição*
*lista-iniciador*
*lista-iniciador ,*

*lista-iniciador:*
*iniciador*
*lista-iniciador , iniciador*

*nome-tipo:*
*lista-qualif-especif declarador-abstrato*$_{opc}$

*declarador-abstrato:*
*ponteiro*
*ponteiro*$_{opc}$ *declarador-abstrato-direto*

*declarador-abstrato-direto:*
*( declarador-abstrato )*
*declarador-abstrato-direto*$_{opc}$ *[ expressão-constante*$_{opc}$ *]*
*declarador-abstrato-direto*$_{opc}$ *( lista-tipo-parâmetro*$_{opc}$*)*

*nome-typedef:*
*identificador*

*comando:*
*comando-rotulado*
*comando-expressão*
*comando-composto*
*comando-seleção*
*comando-iteração*
*comando-salto*

*comando-rotulado:*
*identificador : comando*
`case` *expressão-constante : comando*
`default` *comando*

*comando-expressão:*
*expressão*$_{opc}$*;*

*comando-composto:*
*{ lista-declaração*$_{opc}$ *lista-comando*$_{opc}$ *}*

*lista-comando:*
*comando*
*lista-comando comando*

*comando-seleção:*
`if` *(expressão) comando*
`if` *(expressão) comando* `else` *comando*
`switch` *(expressão) comando*

*comando-iteração:*
`while` *(expressão) comando*
`do` *comando* `while` *(expressão);*
`for` *(expressão*$_{opc}$*; expressão*$_{opc}$*; expressão*$_{opc}$*) comando*

*comando-salto:*
`goto` *identificador;*
`continue;`
`break;`
`return` *expressão*$_{opc}$*;*

*expressão:*
*expressão-atribuição*

*expressão , expressão-atribuição*

*expressão-atribuição:*
*expressão-codicional*
*expressão-unária operador-atribuição expressão-atribuição*

*operador-atribuição: um dentre*
= *= /= %= += -= <<= >>= &= ^= |=

*expressão-codicional:*
*expressão-OU-lógico*
*expressão-OU-lógico ? expressão : expressão-codicional*

*expressão-constante:*
*expressão-codicional*

*expressão-OU-lógico:*
*expressão-E-lógico*
*expressão-OU-lógico* || *expressão-E-lógico*

*expressão-E-lógico:*
*inclusive-OU-expressão*
*expressão-E-lógico && expressão-OU-inclusivo*

*expressão-OU-inclusivo:*
*expressão-OU-exclusivo*
*expressão-OU-inclusivo* | *expressão-OU-exclusivo*

*expressão-OU-exclusivo:*
*expressão-E*
*expressão-OU-exclusivo* ^ *expressão-E*

*expressão-E:*
*expressão-igualdade*
*expressão-E & expressão-igualdade*

*expressão-igualdade:*
*expressão-relacional*
*expressão-igualdade == expressão-relacional*
*expressão-igualdade != expressão-relacional*

*expressão-relacional:*
*expressão-deslocamento*
*expressão-relacional < expressão-deslocamento*
*expressão-relacional > expressão-deslocamento*
*expressão-relacional <= expressão-deslocamento*
*expressão-relacional >= expressão-deslocamento*

*expressão-deslocamento:*

*expressão-aditiva*
*expressão-deslocamento «expressão-aditiva*
*expressão-deslocamento »expressão-aditiva*

*expressão-aditiva:*
*expressão-multiplicativa*
*expressão-aditiva* + *expressão-multiplicativa*
*expressão-aditiva* - *expressão-multiplicativa*

*expressão-multiplicativa:*
*expressão-multiplicativa* * *expressão-molde*
*expressão-multiplicativa* / *expressão-molde*
*expressão-multiplicativa* % *expressão-molde*

*expressão-molde:*
*expressão-unária*
*(nome-tipo) expressão-molde*

*expressão-unária:*
*expressão-pós-fixada*
++ *expressão-unária*
−− *expressão-unária*
*operador-unário expressão-molde*
`sizeof` *expressão-unária*
`sizeof` *(nome-tipo)*

*operador-unário: um dentre*
& ∗ + − ~ !

*expressão-pós-fixada:*
*expressão-primária*
*expressão-pós-fixada[expressão]*
*expressão-pós-fixada(lista-expressão-argumento*$_{opc}$*)*
*expressão-pós-fixada.identificador*
*expressão-pós-fixada->+identificador*
*expressão-pós-fixada*++
*expressão-pós-fixada*−−

*expressão-primária:*
*identificador*
*constante*
*string*
*(expressão)*

*lista-expressão-argumento:*
*expressão-atribuição*
*lista-expressão-atribuição , expressão-atribuição*

*constante:*

*constante-inteira*
*constante-caractere*
*constante-flutuante*
*constante-enumeração*

A seguinte gramática para o pré-processador resume a estrutura das linhas de controles, mas não é apropriada para uma análise mecanizada. Ela inclui o símbolo texto, que significa simplesmente texto de programa, linhas de controle não-condicional do pré-processador ou construções condicionais completas do pré-processador.

*linha-controle:*
# define *identificador sequência-símbolos*
# define *identificador(identificador, . . . , identificador) sequência-símbolos*
# undef *identificador*
# include <*nome-arquivo*>
# include *"nome-arquivo"*
# line *constante "nome-arquivo"*
# line *constante*
# error $sequência\text{-}símbolos_{opc}$
# pragma $sequência\text{-}símbolos_{opc}$
#
*condicional-pré-processador*

*condicional-pré-processador :*
*linha-if texto partes-elif* $parte\text{-}else_{opc}$ #endif

*linha-if:*
# if *expressão-constante*
# ifdef *identificador*
# ifndef *identificador*

*partes-elif:*
*linha-elif texto*
$partes\text{-}elif_{opc}$

*linha-elif:*
# elif *expressão-constante*

*parte-else:*
*linha-else texto*

*linha-else:*
#else

# B. Biblioteca-Padrão

Este apêndice é um resumo da biblioteca definida pelo padrão **ANSI**. A biblioteca-padrão não faz parte da própria linguagem **C**, mas um ambiente que suporte **C** padrão produzirá as declarações e tipo de função, e definições de macro desta biblioteca. Omitimos algumas poucas funções que são de uso limitado ou facilmente simuladas por meio de outras; omitimos caracteres *multibyte*; e omitimos também a discussão sobre problemas locais, ou seja, propriedades que dependem da linguagem, nacionalidade ou cultura local.

As funções, tipos e macros da biblioteca-padrão são declarados em cabeçalhos-padrão:

| | | | | |
|---|---|---|---|---|
| <assert.h> | <float.h> | <math.h> | <stdarg.h> | <stdIib.h> |
| <ctype.h> | <limits.h> | <setjmp.h> | <stddef.h> | <string.h> |
| <errno.h> | <locale.h> | <signal.h> | <stdio.h> | <time.h> |

Um cabeçalho pode ser acessado por meio de

```
#include <cabeçalho>
```

Os cabeçalhos podem ser incluídos em qualquer ordem e qualquer número de vezes. Um cabeçalho deve ser incluído fora de qualquer declaração ou definição externa e antes de qualquer uso de algo declarado nele. Um cabeçalho não precisa ser um arquivo-fonte.

Os identificadores que começam com um sublinhado são reservados para o uso na biblioteca, assim como todos os outros identificadores que começam com um sublinhado e uma letra maiúscula ou outro sublinhado.

## B.1 Entrada e Saida: <stdio.h>

As funções de entrada e saída, tipos e macros definidos em <*stdio.h*> representam um terço da biblioteca.

Um fluxo é uma fonte ou destino de dados que pode ser associado a um disco ou outro periférico. A biblioteca suporta fluxos de texto e fluxos binários, embora em alguns sistemas, notadamente o **UNIX**, eles sejam idênticos. Um fluxo de texto é uma sequência de linhas; cada linha tem zero ou mais caracteres e é terminada por '\n '. Um ambiente pode precisar converter um fluxo de texto

para ou de alguma outra representação (como trocar o '\n ' por retorno de carro e mudança de linha). Um fluxo binário é uma sequência de bytes não processados que registram dados internos, com a propriedade de, ao ser escrito e lido de volta ao mesmo sistema, os bytes não serão modificados.

Um fluxo é conectado a um arquivo ou dispositivo abrindo-o; a conexão é terminada fechando-se o fluxo. Abrir um arquivo retorna um apontador para um objeto do tipo *FILE*, que grava toda a informação necessária para controlar o fluxo. Usaremos os dois termos "*ponteiro de arquivo*" e "*fluxo*" da mesma forma quando não houver ambiguidade.

Quando um programa começar sua execução, os três fluxos *stdin*, *stdout* e *stderr* já estarão abertos.

### B.1.1 Operações de Arquivo

As funções a seguir lidam com operações em arquivos. O tipo *size-t* é o tipo integral não sinalizado produzido pelo operador *sizeof*.

```
FILE *fopen(const char *nomearq, const char *modo)
```

*fopen* abre o arquivo indicado e retorna um fluxo, ou **NULL** se a tentativa falhar. Os valores válidos para modo são

| | |
|---|---|
| "r" | abre arquivo de texto para leitura |
| "w" | cria arquivo de texto para gravação; elimina o conteúdo anterior, se houver |
| "a" | anexa: abre ou cria arquivo de texto para gravação no final do arquivo |
| "r+" | abre arquivo de texto para atualização; elimina o conteúdo anterior, se houver |
| "w+" | cria arquivo de texto para atualização; elimina o conteúdo anterior, se houver |
| "a+" | anexa; abre ou cria arquivo de texto para atualização, grava no final |

O modo de atualização permite a leitura e gravação no mesmo arquivo; *fflush* ou uma função de posicionamento no arquivo deve ser chamado entre uma leitura e uma gravação e vice-versa. Se o modo incluir *b* após a letra inicial, como em "rb" ou "w+b", isso indica um arquivo binário. Os nomes de arquivo são limitados a **FILENAME_MAX** caracteres. No máximo **FOPEN_MAX** arquivos podem ser abertos ao mesmo tempo.

```
FILE *freopen(const char *nomearq, const char *modo, FILE *fluxo)
```

*freopen* abre o arquivo com o modo especificado e associa o fluxo a ele. Ele retorna fluxo, ou **NULL** se houver um erro. *freopen* é usado normalmente para alterar os arquivos associados com *stdin*, *stdout* ou *stderr*.

```
int fflush (FILE *fluxo)
```

Em um fluxo de saída, *fflush* faz com que quaisquer dados armazenados em *buffer* mas não gravados sejam gravados; em um fluxo de entrada, o efeito é indefinido. Ele retorna **EOF** no caso de um erro na gravação, e zero em caso contrário. *fflush* (**NULL**) esvazia todos os fluxos de saída.

```
int fclose (FILE *fluxo)
```

*fclose* esvazia quaisquer dados não gravados para fluxo, elimina qualquer entrada no *buffer* não lida, libera qualquer *buffer* alocado automaticamente, e depois fecha o fluxo. Ele retorna **EOF** se houver erros, e zero caso contrário.

```
int remove(const char *nomearq)
```

*remove* retira o arquivo nomeado, de forma que uma tentativa subsequente de abrir esse arquivo falhará. Ele retorna um valor não-zero se a tentativa falhar.

```
int rename(const char *nomeant, const char *nomenovo)
```

*rename* altera o nome de um arquivo; ele retorna um valor não-zero se a tentativa falhar.

```
FILE *tmpfile(void)
```

*tmpfile* cria um arquivo temporário do modo "wb+" que será automaticamente removido quando fechado ou quando o programa terminar normalmente. *tmpfile* retorna um fluxo, ou **NULL** se não foi capaz de criar o arquivo.

```
char *tmpnam (char s[L_tmpnam])
```

*tmpnam(NULL)* cria uma string que não é o nome de um arquivo existente, e retorna um apontador para um vetor estático interno. *tmpnam(s)* armazena a string em *s* e a retorna como valor de função; *s* deve ter espaço para pelo menos *L_tmpnam* caracteres. *tmpnam* gera um nome diferente toda vez que for chamado; no máximo **TMP_MAX** nomes diferentes são garantidos durante a execução do programa. Observe que *tmpnam* cria um nome, e não um arquivo.

```
int setvbuf (FILE *fluxo, char *buf, int modo, size_t tamanho)
```

*setvbuf* controla a operação do *buffer* para o fluxo; ele deve ser chamado antes da leitura, escrita ou qualquer outra operação. Um modo **_IOFBF** causa a *bufferização* completa, **_IOLBF** a *bufferização* de linha dos arquivos de texto, e **_IONBF** nenhuma operação com *buffer*. Se *buf* não for **NULL**, ele será usado como *buffer*; caso contrário, um *buffer* será alocado. tamanho determina o tamanho do *buffer*. *setvbuf* retorna um valor não-zero em caso de erro.

```
void setbuf(FILE *fluxo, char *buf)
```

Se *buf* for **NULL**, a operação do *buffer* é desativada para o fluxo. Caso contrário, *setbuf* é equivalente a *(void) setvbuf (fluxo, buf, _IOFBF, BUFSIZ)*.

### B.1.2 Saída Formatada

As funções *printf* fornecem a conversão da saída formatada.

```
int fprintf(FlLE *fIuxo, const char *formato, ...)
```

*fprintf* converte e grava a saída para fluxo sob o controle de formato. O valor de retorno é o número de caracteres gravados, ou negativo se houver erro.

A string de formato contém dois tipos de objetos: caracteres normais, que são copiados para o fluxo de saída, e especificações de conversão, cada uma delas causando a conversão e impressão do próximo argumento sucessivo em *fprintf*. Cada especificação de conversão começa com o caractere % e termina com um caractere de conversão. Entre o % e o caractere de conversão pode haver, em ordem:

- Sinalizadores (em qualquer ordem), que modificam a especificação:
  - −, que especifica ajuste esquerdo do argumento convertido no seu campo.
  - +, que especifica que o número sempre será impresso com um sinal.
  - *espaço*, se o primeiro caractere não for um sinal, um espaço serpa prefixado.
  - 0, para conversões numéricas, especifica preenchimento até a largura do campo com zeros iniciais.

  - #, que especifica um formato de saída alternativo. Para *o*, o primeiro dígito será zero. Para *x* ou *X*, *0x* ou *0X* será prefixado a um resultado diferente de zero. Para *e*, *E*, *f*, *g* e *G*, a saída sempre terá um ponto decimal; para *g* e *G*, zeros após os dígitos significativos não serão removidos.
- Um número especificando um tamanho mínimo de campo. O argumento convertido será impresso em um campo com pelo menos esta largura, e maior se necessário. Se o argumento convertido tiver menos caracteres que a largura de campo, ele será preenchido à esquerda (ou direita, se o ajustamento à esquerda tiver sido solicitado) para compor a largura do campo. O caractere de preenchimento é normalmente o espaço, mas será 0 se o sinalizador de preenchimento com zero estiver presente.
- Um ponto, que separa o tamanho do campo da precisão.
- Um número, a precisão, que especifica o número máximo de caracteres impressos para uma string, ou o número de dígitos impressos após o ponto decimal para conversões *e*, *E* ou *f*, ou o número de dígitos significativos para conversões *g* ou *G*, ou o número mínimo de dígitos a serem impressos para um inteiro. (Os iniciais serão acrescentados para compor a largura necessária).
- Um modificador de tamanho *h*, *l* (letra ele), ou *L*, "*h*" indica que o argumento correspondente deve ser impresso como *short* ou *unsigned short*; "*l*" indica que o argumento é um *long* ou *unsigned long*; "*L*" indica que o argumento é um *long double*.

O tamanho ou precisão ou ambos podem ser especificados por *, quando o valor é calculado convertendo-se o(s) próximo(s) argumento(s), que deve(m) ser *int*.

Os caracteres de conversão e seus significados são mostrados na Tabela B.1. Se o caractere após o % não for um caractere de conversão, o comportamento é indefinido.

| CARACTERE | TIPO DE ARGUMENTO; CONVERTIDO PARA |
|---|---|
| d, i | *int*; número opcional sinalizado. |
| o | *unsigned int*; número octal não sinalizado (sem um zero inicial). |
| x, X | *unsigned int*; número hexadecimal não sinalizado (sem um 0x ou 0x inicial), usando abcdef ou ABCDEF para 10,...,15. |
| u | *unsigned int*; número decimal não sinalizado. |
| c | *int*; único caractere, após conversão para unsigned char. |
| s | *char* *; imprime caracteres de cadeia até um '\0' ou número de caracteres indicado na precisão. |
| f | *double*; [−] *mmm.ddd*, onde o número de ds é dado pela precisão (default 6). Precisão 0 suprime o ponto decimal. |
| e, E | *double*; [−] *m.dddddde±xx* ou [−]*m.ddddddE±xx*, onde o número de ds é dado pela precisão (default 6). Precisão 0 suprime o ponto decimal. |
| g, G | *double*; usa %e ou %E se o expoente for menor que −4 ou maior ou igual à precisão; caso contrário, usa %f. Zeros adicionais e um ponto decimal final não são impressos. |
| p | *void* *; apontador (representação dependente da implementação). |
| n | *int* *; o número de caracteres escritos até aqui por esta chamada a *printf* é escrito no argumento. Nenhum argumento é convertido. |
| % | nenhum argumento e convertido; imprime um %. |

Tabela B.1: Conversões de Printf

```
int printf(const char *formato, ...)
```

*printf(...)* é equivalente a *fprintf (stdout,...)*.

```
int sprintf(char *s. const char *formato, ...)
```

*sprintf* é o mesmo que *printf*, exceto que a saída é escrita em uma cadeia *s*, terminada com '\0'. *s* deve ser grande o suficiente para conter o resultado.

O contador de retorno não inclui o '\0'.

```
int vprintf(const char *formato, va_list arg)
int vfprintf(FILE *fIuxo, const char *formato, va_list arg)
int vsprintf(char *s, const char *formato, va_Iist arg)
```

As funções *vprintf*, *vfprintf* e *vsprintf* são equivalentes as funções *printf* correspondentes, exceto que a lista variável de argumentos é substituída por *arg*, que foi inicializado pela macro **va_start** e talvez chamadas a **va_arg**. Veja a discussão de <*stdarg.h*> na Seção B.7.

### B.1.3 Entrada Formatada

As funções *scanf* lidam com a conversão de entrada formatada.

```
int fscanf(FILE *fluxo, const char *formatado, ...)
```

*fscanf* lê do fluxo sob o controle do formato, e atribui valores convertidos por meio de argumentos subsequentes, cada um devendo ser um apontador. Ele retorna quando o formato tiver terminado. *fscanf* retorna **EOF** se ocorrer um fim-de-arquivo ou um erro antes de qualquer conversão; caso contrário, retorna o número de itens entrados convertidos e atribuídos.

A cadeia de formato geralmente contém especificadores de conversão que são usados para direcionarem a interpretação da entrada. A cadeia de formato pode conter:

- Espaços ou tabulações. Se um espaço ou tabulação ocorre em alguma posição no formato, qualquer espaço ou tabulação na entrada correspondente será ignorado.
- Caracteres comuns (não %), que devem combinar com o próximo caractere não-espaço do fluxo de entrada.
- Especificações de conversão, contendo um %, um caractere de supressão de atribuição opcional *, um numero opcional especificando um tamanho máximo de campo, um *h*, *l* ou *L* opcional indicando o tamanho do destino, e um caractere de conversão.

Uma especificação de conversão determina a conversão do próximo campo de entrada. Normalmente o resultado é colocado na variável apontada pelo argumento correspondente. Se a supressão de atribuição for indicada por *, como em *%*s*, entretanto, o campo de entrada é simplesmente saltado; nenhuma atribuição é feita. Um campo de entrada é definido como uma cadeia de caracteres não incluindo o espaço; ele estende-se até o próximo caractere de espaço ou até que o tamanho do campo, se especificado, tenha terminado. Isso significa que *scanf* lerá após limites de linha para encontrar sua entrada, pois as novas-linhas são espaços em branco. (Os caracteres de espaço em branco são o espaço em si, tabulação, nova-linha, retorno de carro, tabulação vertical e avanço de formulário.)

O caractere de conversão indica a interpretação do campo de entrada. O argumento correspondente deve ser um apontador. Os caracteres de conversão válidos são mostrados na Tabela B.2.

Os caracteres de conversão *d*, *i*, *n*, *o*, *u* e *x* podem-ser precedidos por *h* se o argumento for um apontador para short ao invés de para *int*, ou por *l* (letra ele) se o argumento é um apontador para *long*. Os caracteres de conversão *e*, *f* e *g* podem ser precedidos por *l* se um apontador para *double*, ao invés de para *float*, estiver na lista de argumento, e por *L* se houver um apontador para um *long double*.

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

| CARACTERE | DADO DE ENTRADA; TIPO DE ARGUMENTO |
|---|---|
| d | inteiro decimal; *int* *. |
| i | inteiro; *int* *. O inteiro pode estar em **octal** (0 no início) ou **hexadecimal** (0x ou 0x no início). |
| o | inteiro **octal** (com ou sem zero inicial); *unsigned int* *. |
| u | inteiro decimal não sinalizado; *unsigned int* *. |
| x | inteiro **hexadecimal** (com ou sem 0x ou 0X inicial); *unsigned int* *. |
| c | caracteres; *char* *. Os próximos caracteres de entrada (*default* 1) são colocados no ponto indicado, até o número dado pelo campo de tamanho; o *default* é 1. Nenhum '\0' é anexado. O salto normal sobre espaço em branco é suprimido; para ler o próximo caractere não-espaço, use *%1s*. |
| s | cadeia de caracteres sem espaço em branco (não entre aspas); *char* *, apontando para um vetor de caracteres com tamanho suficiente para a cadeia e o '\0' de término que será incluído. |
| e,f,g | número em ponto-flutuante; *float* *. O formato de entrada para *float* é um sinal opcional, uma cadeia de números podendo conter um ponto decimal e um campo opcional de expoente contendo um *E* ou *e* seguido por um inteiro possivelmente sinalizado. |
| p | valor apontador conforme impresso por *printf ("%p")*; *void* *. |
| n | escreve no argumento o número de caracteres lidos até o momento por esta chamada; *int* *. Nenhuma entrada elida. O contador de item convertido não sera incrementado. |
| [...] | combina com a maior cadeia não vazia de caracteres de entrada dentre o conjunto entre colchetes; *char* *. Um '\0' é anexado. []...] inclui ] ao conjunto. |
| [^...] | combina com a maior cadeia não vazia de caracteres de entrada não incluída no conjunto entre colchetes; *char* *. Um '\0' é anexado. [^]...]inclui ] ao conjunto. |
| % | Literal %; nenhuma atribuição é feita. |

Tabela B.2: Conversões de Scanf

*scanf (...)* é idêntico a *fscanf (stdin,...)*.

```
int sscanf(const char *s, const char *formato, ...)
```

*sscanf(s, ...)* é equivalente a *scanf(...)* exceto que os caracteres de entrada são tomados da cadeia *s*.

### B.1.4 Funções de Entrada e Saída de Caractere

```
int fgetc(FILE *fluxo)
```

*fgetc* retorna o próximo caractere do fluxo como um *unsigned char* (convertido para um *int*), ou **EOF** se houver um fim de arquivo ou erro.

```
char *fgets(char *s, int n, FILE *fluxo)
```

*fgets* lê no máximo os próximos $n-1$ caracteres no vetor *s*, parando se uma nova-linha for encontrada; a nova-linha é incluída no vetor, que é terminado por '\0'. *fgets* retorna *s*, ou **NULL** se houver um fim de arquivo ou erro.

```
int fputc(int c, FILE *fluxo)
```

*fputs* escreve o caractere *c* (convertido para um *unsigned char*) no fluxo. Ele retorna o caractere escrito, ou **EOF** se houver erro.

```
int fputs(const char *s, FILE *fluxo)
```

*fputs* grava uma cadeia *s* (que não precisa conter '\n ') no fluxo; ele retorna um valor não-negativo, ou **EOF** se houver erro.

```
int getc(FILE *fluxo)
```

*getc* é equivalante a *fgetc*, exceto que se for uma macro, fluxo pode ser avaliado mais de uma vez.

```
int getchar(void)
```

*getchar* é equivalente a *getc(stdin)*.

```
char *gets(char *s)
```

*gets* lê a próxima linha de entrada no vetor *s*; o caractere de nova-linha final é substituído por '\0'. Ele retorna *s*, ou **NULL** se houver um fim de arquivo ou erro.

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

*putc* é equivalente a *fputc*, exceto que se for uma macro, pode avaliar o fluxo mais de uma vez.

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

*putchar(c)* é equivalente a *putc(c,stdout)*.

```
int puts(const char *s)
```

*puts* grava a cadeia *s* e uma nova-linha em *stdout*. Ele retorna **EOF** se houver erro, caso contrário, um valor não negativo e retornado.

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

*ungetc* coloca *c* (convertido para um *unsigned char*) de volta no fluxo, onde será retornado na próxima leitura. Somente um caractere retornado por fluxo é garantido. **EOF** não pode ser colocado de volta. *ungetc* retorna o caractere colocado de volta, ou **EOF** se houver erro.

### B.1.5 Funções de Entrada e Saída Direta

```
size_t fread(void *ptr, size_t tamanho, size_t nobj, FILE *fluxo)
```

*fread* lê o fluxo para o vetor *ptr* no máximo *nobj* objetos do tamanho indicado. *fread* retorna o número de objetos lidos; este pode ser menor que o número solicitado. *feof* e *ferror* devem ser usados para determinarem o status.

```
size_t fwrite(const void *ptr, size_t tamanho, size_t nobj, FILE *fluxo)
```

*fwrite* escreve, a partir do vetor *ptr*, *nobj* objetos do tamanho indicado em fluxo. Ele retorna o número de objetos gravados, que é menor do que *nobj* em caso de erro.

### B.1.6 Funções de Posicionamento de Arquivo

```
int fseek(FILE *fluxo, long deslocamento, int origem)
```

*fseek* define a posição de arquivo para o fluxo; uma leitura ou gravação subsequente acessará dados a partir da nova posição. Para um arquivo binário, a posição é definida para deslocamento caracteres a partir da origem, que pode ser **SEEK_SET** (início), **SEEK_CUR** (posição corrente) ou **SEEK_END** (fim do arquivo). Para um fluxo de texto, o deslocamento deve ser zero ou um valor retornado por *ftell* (neste caso a origem deve ser **SEEK_SET**). *fseek* retorna um valor diferente de zero em caso de erro.

```
long ftell(FILE *fluxo)
```

*ftell* retorna a posição corrente do arquivo para fluxo, ou $-1$ em caso de erro.

```
void rewind(FILE *fluxo)
```

*rewind(fp)* é equivalente a *fseek(fp, 0L, SEEK_SET)*; *clearerr(fp)*.

```
int fgetpos(FILE *fluxo, fpos_t *ptr)
```

*fgetpos* registra a posição corrente no fluxo em **ptr*, para uso subsequente por *fsetpos*. O tipo *fpos_t* é adequado para o registro desses valores. *fgetpos* retorna um valor diferente de zero em caso de erro.

```
int fsetpos(FILE *fluxo, const fpos_t *ptr)
```

*fsetpos* posiciona o fluxo na posição registrada por fgetpos em **ptr*. *fsetpos* retorna um valor diferente de zero em caso de erro.

### B.1.7 Funções de Erro

Muitas das funções na biblioteca definem indicadores de estado quando ocorre um erro ou fim de arquivo. Estes indicadores podem ser setados e testados explicitamente. Além disso, a expressão inteira *errno* (declarada em <*errno.h*>) pode conter um número de erro que dá mais informações sobre o erro mais recente.

```
void clearerr (FILE *fluxo)
```

*clearerr* apaga os indicadores de erro e fim de arquivo para o fluxo.

```
int feof (FILE *fluxo)
```

*feof* retorna um valor diferente de zero se o indicador de fim de arquivo para o fluxo estiver setado.

```
int ferror (FILE *fluxo)
```

*ferror* retorna um valor diferente de zero se o indicador erro para o fluxo estiver setado.

```
void perror (const char *s)
```

*perror(s)* imprime *s* e uma mensagem de erro definida pela implementação correspondente ao inteiro em *errno*, como em

```
  fprintf(stderr, "\%s: \%s\n", s, "mensagem de error ");
```

Veja *strerror* na Seção B.3.

## B.2 Testes de Classe de Caractere: <ctype.h>

O cabeçalho <*ctype.h*> declara funções para testar caracteres. Para cada função, o argumento é um *int*, cujo valor deve ser **EOF** ou outro representável como *unsigned char*, e o valor de retorno é um *int*. As funções retornam um valor **diferente de zero** (verdade) se o argumento *c* satisfizer a condição descrita, e **zero** se não.

| | |
|---|---|
| **isalnum(c)** | *isalpha(c)* ou *isdigit(c)* é verdade |
| **isalpha(c)** | *isupper(c)* ou *islower(c)* é verdade |
| **iscntrl(c)** | caractere de controle |
| **isdigit(c)** | dígito decimal |
| **isgraph(c)** | caractere de impressão exceto espaço |
| **islower(c)** | letra minúscula |
| **isprint(c)** | caractere de impressão inclui espaço |
| **ispunct(c)** | caractere de impressão exceto espaço ou letra ou dígito |
| **isspace(c)** | espaço, mudança de página, nova-linha, retorno de carro, qualquer tabulação |
| **isupper(c)** | letra maiúscula |
| **isxdigit(c)** | dígito hexadecimal |

No conjunto de caracteres **ASCII** de sete bits, os caracteres de impressão são de **0x20** (' ') a **0x7E** ('~'); os caracteres de controle são de **0** (**NUL**) a **0x1F** (**US**) e **0x7F** (**DEL**).

Além disso, existem duas funções que convertem os tipos de letras:

| | |
|---|---|
| **int tolower(int c)** | converte *c* para minúscula |
| **int toupper(int c)** | converte *c* para maiúscula |

Se *c* é uma letra maiúscula, *tolower(c)* retorna a letra minúscula correspondente; caso contrário, retorna *c*. Se *c* é uma letra minúscula, *toupper(c)* retorna a letra maiúscula correspondente; caso contrário, novamente o próprio *c* é retornado.

## B.3 Funções de String: <string.h>

Há dois grupos de funções de string definidos no cabeçalho <*string.h*>. O primeiro tem nomes começando com *str*; o segundo tem nomes começando com mem. Exceto por *memmove*, o comportamento é indefinido se a cópia ocorre entre objetos superpostos. As funções de comparação tratam os argumentos como vetores de *unsigned char*.

Na tabela a seguir, as variáveis *s* e *t* são do tipo *char* *; *cs* e *ct* são do tipo *const char* *; *n* é do tipo *size_t*; e *c* é um *int* convertido para *char*.

| | |
|---|---|
| `char *strcpy(s,ct)` | copia cadeia *ct* para a string *s*, incluindo '\0'; retorna *s*. |
| `char *strncpy(s,ct,n)` | copia no máximo *n* caracteres da string *ct* para *s*; retorna *s*. Preenche com '\0' se *s* tiver menos de *n* caracteres. |
| `char *strcat(s,ct)` | concatena a string *ct* ao final da string *s*; return *s*. |
| `char *strncat(s,ct,n)` | concatena no máximo n caracteres da string *ct* para a string *s*, termina *s* com '\0'; retorna *s*. |
| `int strcmp(cs,ct)` | compara a string cs com a string *ct*; retorna $< 0$ se $cs < ct$, 0 se $cs == ct$, ou >0 Se $cs > ct$. |
| `int strncmp(cs,ct,n)` | compara no máximo *n* caracteres da string cs com a string *ct*; retorna $< 0$ se $cs < ct$, 0 se $cs == ct$ ou $> 0$ se $cs > ct$. |
| `char *strchr(cs,c)` | retorna apontador para primeira ocorrência de *c* em cs, ou **NULL** se não estiver presente. |

| | |
|---|---|
| `char *strrchr(cs,c)` | retorna apontador para última ocorrência de *c* em *cs*, ou **NULL** se não estiver presente. |
| `size_t strspn(cs,ct)` | retorna tamanho do prefixo de *cs*, consistindo em caracteres em *ct*. |
| `size_t strcspn(cs,ct)` | retorna tamanho do prefixo de *cs*, consistindo em caracteres não em *ct*. |
| `char *strpbrk(cs,ct)` | retorna apontador para primeira ocorrência na string *cs* de qualquer caractere na string *ct*, ou **NULL** se não achar. |
| `char *strstr(cs,ct)` | retorna apontador para primeira ocorrência da string *ct* em *cs*, ou **NULL** se não achar. |
| `size_t strlen(cs)` | retorna tamanho de *cs*. |
| `char *strerror(n)` | retorna apontador para string definida pela implementação correspondente ao erro *n*. |
| `char *strtok(s,ct)` | *strtok* procura em *s* códigos delimitados por caracteres em *ct*; veja a seguir. |

Uma sequência de chamadas a *strtok(s,ct)* divide *s* em códigos, cada um delimitado por um caractere de *ct*. A primeira chamada em uma sequência tem um *s* diferente de **NULL**. Ela procura o primeiro código em *s* contendo caracteres ausentes em *ct*; o processo termina superpondo-se o próximo caractere de *s* com '\0' e retornando um apontador para o código. Cada chamada subsequente, indicada por um valor **NULL** de *s* , retorna o próximo código, pesquisando a partir do próximo caractere. *strtok* retorna **NULL** quando nenhum outro código for encontrado. A cadeia *ct* pode ser diferente a cada chamada.

As funções *mem...* servem para manipular objetos como vetores de caractere; a ideia é obter uma interface para rotinas eficientes. Na tabela a seguir, *s* e *t* são do tipo *void* *; *cs* e *ct* são do tipo *const void**; *n* é do tipo *size_t*; e *c* é um *int* convertido para um *unsigned char*.

| | |
|---|---|
| `void *memcpy(s,ct,n)` | copia *n* caracteres de *ct* para *s*, e retorna *s*. |
| `void *memmove(s,ct,n)` | o mesmo que *memcpy*, mas funciona mesmo com objetos superpostos. |
| `int memcmp(cs,ct,n)` | compara os primeiros *n* caracteres de *cs* com *ct*; retorna como em *strcmp*. |
| `void *memchr(cs,c,n)` | retorna apontador para primeira ocorrência do caractere *c* em *cs*, ou **NULL** se não estiver presente entre os primeiros *n* caracteres. |
| `void *memset(s,c,n)` | coloca caractere *c* nos primeiros *n* caracteres de *s*, retorna *s*. |

## B.4 Funções Matemáticas: <math.h>

O cabeçalho <*math.h*> declara funções matemáticas e macros.

As macros **EDOM** e **ERANGE** (encontradas em <*errno.h*> são constantes integrais diferentes de zero usadas para indicarem erros de domínio e faixa para as funções; **HUGE_VAL** é um valor *double* positivo. Um erro de domínio ocorre se um argumento estiver fora do domínio sobre o qual a função é definida. Em um erro de domínio, *errno* é definido para **EDOM**; o valor de retorno depende da implementação. Um erro de faixa ocorre se o resultado de uma função não puder ser representado como um *double*. Se o resultado estourar, a função retorna **HUGE_VAL** com o sinal correto, e *errno* é definido para **ERANGE**. Se o resultado estiver abaixo da faixa, a função retorna zero; se *errno* é definido para **ERANGE** depende da implementação.

Na tabela a seguir, *x* e *y* são do tipo *double*, *n* é um *int* e todas as funções retornam *double*. Os ângulos para funções trigonométricas são expressos em radianos.

| | |
|---|---|
| `sin(x)` | seno de $x$ |
| `cos(x)` | co-seno de $x$ |
| `tan(x)` | tangente de $x$ |
| `asin(x)` | $sin^{-1}(x)$ na faixa $[-\pi/2, \pi/2], x \in [-1,1]$. |
| `acos(x)` | $cos^{-1}(x)$ na faixa $[0,\pi], x \in [-1,1]$. |
| `atan(x)` | $tan^{-1}(x)$ na faixa $[-\pi/2, \pi/2]$. |
| `atan2(y,x)` | $tan^{-1}(y/x)$ na faixa $[-\pi,\pi]$. |
| `sinh(x)` | seno hiperbólico de $x$ |
| `cosh(x)` | co-seno hiperbólico de $x$ |
| `tanh(x)` | tangente hiperbólica de $x$ |
| `exp(x)` | função exponencial $e^x$. |
| `log(x)` | logaritmo natural $ln(x)$, $x > 0$. |
| `log10(x)` | logaritmo base $log_{10}(x)$, $x > 0$. |
| `pow(x,y)` | $x^y$. Um erro de domínio ocorre se $x = 0$ e $y \leq 0$, ou se $x < 0$ e $y$ não for inteiro. |
| `sqrt(x)` | Raiz quadrada de $x$, $x \geq 0$. |
| `ceil(x)` | menor inteiro não menor que $x$, como *double* |
| `floor(x)` | maior inteiro não maior que $x$, como *double* |
| `fabs(x)` | valor absoluto $\|x\|$ |
| `ldexp(x,n)` | $x * 2^n$. |
| `frexp(x, int *exp)` | $s$ divide $x$ em uma fração normalizada no intervalo $[1/2, 1)$ que é retornada, e uma potência de 2, que é armazenada em $*exp$. Se $x$ é zero, as duas partes do resultado são zero. |
| `modf(x, double *ip)` | divide $x$ em partes inteira e fracionária, cada uma com o mesmo sinal de $x$. Armazena a parte inteira em $*ip$, e retorna a parte fracionária. |
| `fmod(x,y)` | resto em ponto flutuante de $x/y$, com o mesmo sinal de $x$. Se $y$ é zero, o resultado depende da implementação. |

## B.5 Funções Utilitárias: <stdlib.h>

O cabeçalho *<stdlib.h>* declara funções para conversão de número, alocação de memória e tarefas semelhantes.

```
double atof(const char *s)
```

*atof* converte *s* para *double*; ela é equivalente a *strtod(s, (char**)NULL)*.

```
int atoi(const char *s)
```

converte *s* para *int*; é equivalente a *(int)strtol(s, (char**)NULL, 10)*.

```
long atol(const char *s)
```

converte *s* para *long*; é equivalente a *strtol(s, (char**)NULL, 10)*.

```
double strtod(const char *s, char **pfim)
```

*strtod* converte o prefixo de *s* para *double*, ignorando o espaço não significativo; ele armazena um apontador para qualquer sufixo não convertido em **pfim*, a menos que *pfim* seja **NULL**. Se a resposta estourar, **HUGE_VAL** é retornado com o sinal apropriado; se a resposta estiver abaixo da faixa, zero é retornado. Nesses dois casos, *errno* é setado para **ERANGE**.

```
long strtol(const char *s, char **pfim, int base)
```

*strtol* converte o prefixo de *s* para *long*, ignorando o espaço em branco não significativo; ela armazena um apontador para qualquer sufixo não convertido em **pfim*, a não ser que *pfim* seja **NULL**. Se base estiver entre 2 e 36, a conversão é feita supondo-se que a entrada é escrita nessa base. Se base é zero, a base pode ser 8, 10 ou 16; um 0 inicial indica octal, e 0*x* ou 0*X*, hexadecimal. As letras de qualquer tipo representam dígitos de 10 até *base* − 1; um 0*x* ou 0*X* é permitido na base 16. Se a resposta estourar, **LONG_MAX** ou **LONG_MIN** é retornado, dependendo do sinal do resultado, e *errno* é setado para **ERANGE**.

```
unsigned long strtoul(const char *s, char **pfim, int base)
```

*strtoul* é o mesmo que *strtol*, exceto que o resultado é *unsigned long* e o valor de erro é **ULONG_MAX**.

```
int rand(void)
```

*rand* retorna um inteiro pseudo-randômico na faixa de 0 a *RAND_MAX*, que é pelo menos 32767.

```
void srand(unsigned int semente)
```

*srand* usa semente como uma semente para uma nova sequência de números pseudo-randômicos. A semente inicial é 1.

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

*calloc* retorna um apontador para o espaço de um vetor de *nobj* objetos, cada um com tamanho indicado por *tamanho*, ou **NULL** se o pedido não puder ser satisfeito. O espaço é inicializado com o byte **zero**.

```
void *malloc(size_t tamanho)
```

*malloc* retorna um apontador para o espaço de um objeto com tamanho indicado por *tamanho*, ou **NULL** se o pedido não puder ser satisfeito. O espaço não é inicializado.

```
void *realloc(void *p, size_t tamanho)
```

*realloc* altera o tamanho do objeto apontado por *p* para tamanho. O conteúdo ficará inalterado até o menor entre os tamanhos antigo e novo. Se o novo tamanho é maior, o novo espaço não é inicializado. *realloc* retorna um apontador para o novo espaço, ou **NULL** se o pedido não puder ser satisfeito, caso em que **p* permanece inalterado.

```
void free(void *p)
```

*free* desaloca o espaço apontado por *p*; ela não faz nada se *p* for **NULL**. *p* deve ser um ponteiro para um espaço previamente alocado por *calloc*, *malloc* ou *realloc*.

```
void abort(void)
```

*abort* faz com que o programa termine anormalmente, como se fosse por *raise(SIGABRT)*.

```
void exit(int status)
```

*exit* realiza um término normal no programa. As funções de *atexit* são chamadas em ordem contrária de registro, arquivos abertos são esvaziados, fluxos abertos são fechados e o controle é retornado ao ambiente. Como status é retornado ao ambiente depende da implementação, mas zero é considerado um término com sucesso. Os valores **EXIT_SUCCESS** e **EXIT_FAILURE** também podem ser usados.

```
int atexit(void (*fcn)(void))
```

*atexit* registra a função *fcn* para ser chamada quando o programa terminar normalmente; ela retorna não-zero se o registro não puder ser feito.

```
int system(const char *s)
```

*system* passa a string *s* para o ambiente para execução. Se *s* é **NULL**, system retorna não-zero se houver um processador de comando. Se *s* não é **NULL**, o valor de retorno depende da implementação.

```
char *getenv(const char *nome)
```

*getenv* retorna a string de ambiente associada com nome, ou **NULL** se não existir uma string. Os detalhes dependem da implementação.

```
void *bsearch(const void *chave,
              const void *base,
              size_t n, size_t tamanho,
              int (*cmp)(const void *valchave,
              const void *datum))
```

*bsearch* procura em $base[0] \ldots base[n-1]$ um item que combine com **chave*. A função *cmp* deve retornar negativa se seu primeiro argumento (a chave de pesquisa) for menor que seu segundo (uma entrada da tabela), zero se igual, e positivo se maior. Os itens de vetor *base* devem estar em ordem ascendente. *bsearch* retorna um apontador para o item que combinar, ou **NULL** se não existir.

```
void qsort(void *base,
           size_t n,
          size_t tamanho,
          int (*cmp)(const void *, const void *))
```

*qsort* ordena ascendentemente um vetor $base[0] \ldots base[n-1]$ de objetos do tamanho indicado. A função de comparação *cmp* funciona como em *bsearch*.

```
  int abs(int n)
```

*abs* retorna o valor absoluto de seu argumento *int*.

```
long labs(long n)
```

*labs* retorna o valor absoluto de seu argumento *long*.

```
div_t div (int num, int denom)
```

*div_t* calcula o quociente e resto de $num/denom$. Os resultados são armazenados nos membros inteiros *quot* e *rem* de uma estrutura de tipo *div_t*.

```
ldiv_t ldiv (long num, long denom)
```

*ldiv_t* calcula o quociente e resto de $num/denom$. Os resultados são armazenados nos membros long *quot* e *rem* de uma estrutura do tipo *ldiv_t*.

## B.6 Diagnósticos: <assert.h>

A macro *assert* é usada para incluir diagnósticos em programas:

```
void assert (int expressão)
```

Se expressão é zero quando

```
assert (expressão)
```

é executada, a macro *assert* imprimirá em *stderr* uma mensagem, como

```
Assertion failed: expressão file arquivo, line mm
```

Depois ela chama *abort* para terminar a execução. O nome de arquivo e número de linha da fonte vêm das macros de pré-processador **__FILE__** e **__LINE__**.

Se NDEBUG estiver definido quando <assert.h> for incluído, a macro *assert* é ignorada.

## B.7 Listas de Argumento Variáveis: <stdarg.h>

O cabeçalho <*stdarg.h*> provê facilidades para examinar uma lista de argumentos de função de número e tipo desconhecidos.

Suponha que *ultarg* seja o último parâmetro nomeado de uma função *f* com um número variável de argumentos. Depois, declare dentro de *f* uma variável ap de tipo *va_list* que aponte para cada argumento por sua vez:

```
va_list ap;
```

*ap* deve ser inicializada uma vez com a macro *va_start* antes que qualquer argumento não nomeado seja acessado:

```
va_start(va_list ap, ultarg);
```

Depois disso, cada execução da macro *va_arg* produzirá um valor que tem o tipo e valor do próximo argumento não nomeado, e também modificará *ap* de modo que o próximo uso de *va_arg* retorne o próximo argumento:

```
tipo va_arg(va_list ap, tipo);
```

A macro

```
void va_end(va_list ap);
```

deve ser chamada uma vez depois que os argumentos tenham sido processados, mas antes que *f* saia.

## B.8 Saltos Não-locais: <setjmp.h>

As declarações em <*setjmp.h*> provê uma forma de evitar a sequência normal de chamada e retorno de função, normalmente para permitir um retorno imediato de uma chamada de função profundamente aninhada.

```
int setjmp(jmp_buf amb)
```

A macro *setjmp* salva a informação de estado em *amb* para uso em *longjmp*. O retorno é zero de uma chamada direta de *setjmp*, e não-zero de uma chamada subsequente de *longjmp*. Uma chamada a *setjmp* só pode ocorrer em certos contextos, basicamente no teste de *if*, *switch* e laços, e somente em expressões relacionais simples.

```
if (setjmp (amb) == 0)
/* vem para cá na chamada direta */
else
/* vem para cá chamando longjmp */
```

```
void longjmp(jmp_buf amb, int val)
```

*longjmp* recupera o estado salvo pela chamada mais recente a *setjmp*, usando a informação salva em *amb*, e a execução continua como se a função *setjmp* tivesse acabado de ser executada e retornasse com o valor não-zero *val* . A função contendo o *setjmp* não pode ter terminado. Os objetos acessíveis possuem os valores que tinham quando *longjmp* foi chamada; os valores não são salvos por *setjmp*.

## B.9 Sinais: <signal.h>

O cabeçalho <*signal.h*> prove facilidades para manipular condições excepcionais que surgem durante a execução, como um sinal de interrupção de uma fonte externa ou um erro na execução.

```
void (*signal(int sig, void (*manip)(int)))(int)
```

*signal* determina como os sinais subsequentes serão manipulados. Se *manip* for **SIG_DFL**, o comportamento *default* definido pela implementação é usado; se for **SIG_IGN**, o sinal é ignorado; caso contrário, a função apontada por *manip* será chamada com o argumento do tipo de sinal. Os sinais válidos incluem

| | |
|---|---|
| **SIGABRT** | término anormal, p. ex., de *abort* |
| **SIGFPE** | erro aritmético, p. ex., divisão por zero ou estouro |
| **SIGILL** | imagem de função ilegal, p. ex., instrução ilegal |
| **SIGINT** | atenção interativa, p. ex., interrupção |
| **SIGSEGV** | acesso ilegal à memória, p. ex., acesso fora dos limites da memória |
| **SIGTERM** | pedido de término enviado a este programa |

*signal* retorna o valor anterior de *manip* para o sinal específico, ou **SIG_ERR** se houver um erro.

Quando um sinal *sig* subsequentemente ocorrer, o sinal é restaurado ao seu comportamento *default*; então, a função de manipulação de sinal é chamada, como se por *(*manip)(sig)*. Se o manipulador retornar, a execução continuará onde estava quando ocorreu o sinal.

O estado inicial dos sinais é definido pela implementação.

```
int raise (int sig)
```

*raise* envia o sinal *sig* para o programa; ela retorna não-zero se não tiver sucesso.

## B.10 Funções de Data e Hora: <time.h>

O cabeçalho <*time.h*> declara tipos e funções para manipular a data e hora. Algumas funções processam a hora local, que pode diferir da hora do calendário, p. ex., por causa do fuso-horário. *clock_t* e *time_t* são tipos aritméticos representando horas, e *struct tm* contém os componentes da hora do calendário.

| | |
|---|---|
| **int tm_sec;** | segundo após o minuto (0,61) |
| **int tm_min;** | minutos após a hora (0,59) |
| **int tm_hour;** | horas desde a meia-noite (0,23) |
| **int tm_mday;** | dia do mês (1,31) |
| **int tm_mon;** | meses desde janeiro (0,11) |
| **int tm_ano;** | anos desde 1900 |

**int tm_wday;** dias desde domingo (0,6)
**int tm_yday;** dias desde 1 de janeiro (0,365)
**int tm_isdst;** sinalizador de "horário de verão"

*tm_isdst* é positivo se o **Horário de Verão** estiver em efeito, zero se não, e negativo se a informação não for disponível.

```
clock_t clock (void)
```

*clock* retorna a hora do processador usada pelo programa desde o início da execução, ou $-1$ se não for disponível. *clock()/CLOCKS_PER_SEC* é uma hora em segundos.

```
time_t time (time_t *tp)
```

*time* retorna a hora do calendário corrente ou $-1$ se a hora não for disponível. Se *tp* não é **NULL**, o valor de retorno é também atribuído a **tp*.

```
double difftime (time_t hora2, time_t hora1)
```

*difftime* retorna a diferença $hora2 - hora1$ em segundos.

```
time_t mktime (struct tm *tp)
```

*mktime* converte a hora local na estrutura **tp* em hora de calendário na mesma representação usada por *time*. Os componentes terão valores nas faixas mostradas. *mktime* retorna a hora do calendário ou $-1$ se ela não puder ser representada.

As quatro funções a seguir retornam ponteiros para objetos estáticos que podem ser superpostos por outras chamadas.

```
char *asctime (const struct tm *tp)
```

*asctime* converte a hora na estrutura **tp* para uma cadeia no formato

```
Wed May 13 15:14:13 2020\n\0
```

```
char *ctime (const time_t *tp)
```

*ctime* converte a hora do calendário **tp* para a hora local; ela é equivalente a

```
asctime (localtime (tp))
```

```
struct tm *gmtime(const time_t *tp)
```

*gmtime* converte a hora do calendário **tp* na hora de **Greenwich**. Ela retorna **NULL** se o horário no meridiano de Greenwich não estiver disponível. O nome *gmtime* tem um significado histórico.

```
struct tm *localtime(const time_t *tp)
```

*localtime* converte a hora do calendário **tp* em hora local.

```
size_t strftime(char *s, size_t smax, const char *fmt, const struct tm *tp)
```

*strftime* formata a informação de data e hora de **tp* em *s* de acordo com *fmt*, que é semelhante a um formato de *printf*. Os caracteres comuns (incluindo o '\0' terminal) são copiados para *s*. Cada %c é substituído como descrevemos a seguir, usando valores adequados para o ambiente local. Não mais do que *smax* caracteres são colocados em *s*. *strftime* retorna o número de caracteres, excluindo o '\0', ou zero se forem produzidos mais do que *smax* caracteres.

| | |
|---|---|
| %a | dia da semana abreviado. |
| %A | dia da semana completo. |
| %b | nome do mês abreviado. |
| %B | nome do mês completo. |
| %c | representação local da data e hora. |
| %d | dia do mês (01-31). |
| %H | hora (relógio 24 horas) (00-23). |
| %I | hora (relógio 12 horas) (01-12). |
| %j | dia do ano (001-366). |
| %m | mês (01-12). |
| %M | minuto (00-59). |
| %p | equivalente local de AM ou PM. |
| %S | segundo (00-61). |
| %U | número da semana no ano (domingo como primeiro dia da semana (00-53). |
| %w | dia da semana (0-6, domingo é 0). |
| %W | número da semana no ano (segunda como primeiro dia da semana. (00-53). |
| %x | representação local da data. |
| %X | representação local da hora. |
| %y | ano sem século (00-99). |
| %Y | ano com século. |
| %Z | nome do fuso-horário, se houver. |
| %% | % |

## B.11 Limites Definidos pela Implementação: <limits.h> e <float.h>

O cabeçalho <*limits.h*> define constantes para os tamanhos de tipos integrais. Os valores abaixo são magnitudes mínimas aceitáveis; valores maiores podem ser usados.

| | | |
|---|---|---|
| **CHAR_BIT** | 8 | bits em um *char* |
| **CHAR_MAX** | **UCHAR_MAX** ou **SCHAR_MAX** | valor máximo de *char* |
| **CHAR_MIN** | 0 ou **SCHAR_MIN** | valor mínimo de *char* |
| **INT_MAX** | 32767 | valor máximo de *int* |
| **INT_MIN** | -32767 | valor mínimo de *int* |
| **LONG_MAX** | 2147483647 | valor máximo de *long* |
| **LONG_MIN** | -2147483647 | valor mínimo de *long* |
| **SCHAR_MAX** | +127 | valor máximo de *signed char* |
| **SCHAR_MIN** | -127 | valor mínimo de *signed char* |
| **SHRT_MAX** | +32767 | valor máximo de *short* |
| **SHRT_MIN** | -32767 | valor mínimo de *short* |
| **UCHAR_MAX** | 255 | valor máximo de *unsigned char* |
| **UINT_MAX** | 65535 | valor máximo de *unsigned int* |
| **ULONG_MAX** | 4294967295 | valor máximo de *unsigned long* |
| **USHRT_MAX** | 65535 | valor máximo de *unsigned short* |

Os nomes na tabela a seguir, um subconjunto de <*float.h*>, são constantes relacionadas à aritmética de ponto flutuante. Quando um valor é dado, ele representa a magnitude mínima para a quantidade correspondente. Cada implementação define os valores apropriados.

| | | |
|---|---|---|
| **FLT_RADIX** | 2 | base de representação de expoente, p. ex; *2*, *16* |
| **FLT_ROUNDS** | | modo de arredondamento de ponto flutuante para adição. |
| **FLT_DIG** | 6 | dígitos decimais de precisão |
| **FLT_EPSILON** | 1E-5 | menor número *x* tal que *1.0+x != 1.0* |
| **FLT_MANT_DIG** | | número de dígitos da base **FLT_RADIX** na mantissa |
| **FLT_MAX** | 1E+37 | número máximo em ponto flutuante |
| **FLT_MAX_EXP** | | máximo n tal que **FLT_RADIX**$^{n-1}$ seja representável |
| **FLT_MIN** | 1E-37 | número mínimo normalizado em ponto flutuante |
| **FLT_MIN_EXP** | | mínimo *n* tal que $10^n$ seja um número normalizado |
| **DBL_DIG** | 10 | dígitos decimais de precisão |
| **DBL_EPSILON** | 1E-9 | menor número *x* tal que *1.0+x != 1.0* |
| **DBL_MANT_DIG** | | número de dígitos de base **FLT_RADIX** na mantissa |
| **DBL_MAX** | 1E+37 | número *double* máximo em ponto flutuante |
| **DBL_MAX_EXP** | | máximo n tal que **FLT_RADIX**$^{n-1}$ seja representável |
| **DBL_MIN** | 1E-37 | número *double* mínimo normalizado |
| **DBL_MIN_EXP** | | mínimo *n* tal que $10^n$ seja um número normalizado |

# C. Resumo das Mudanças

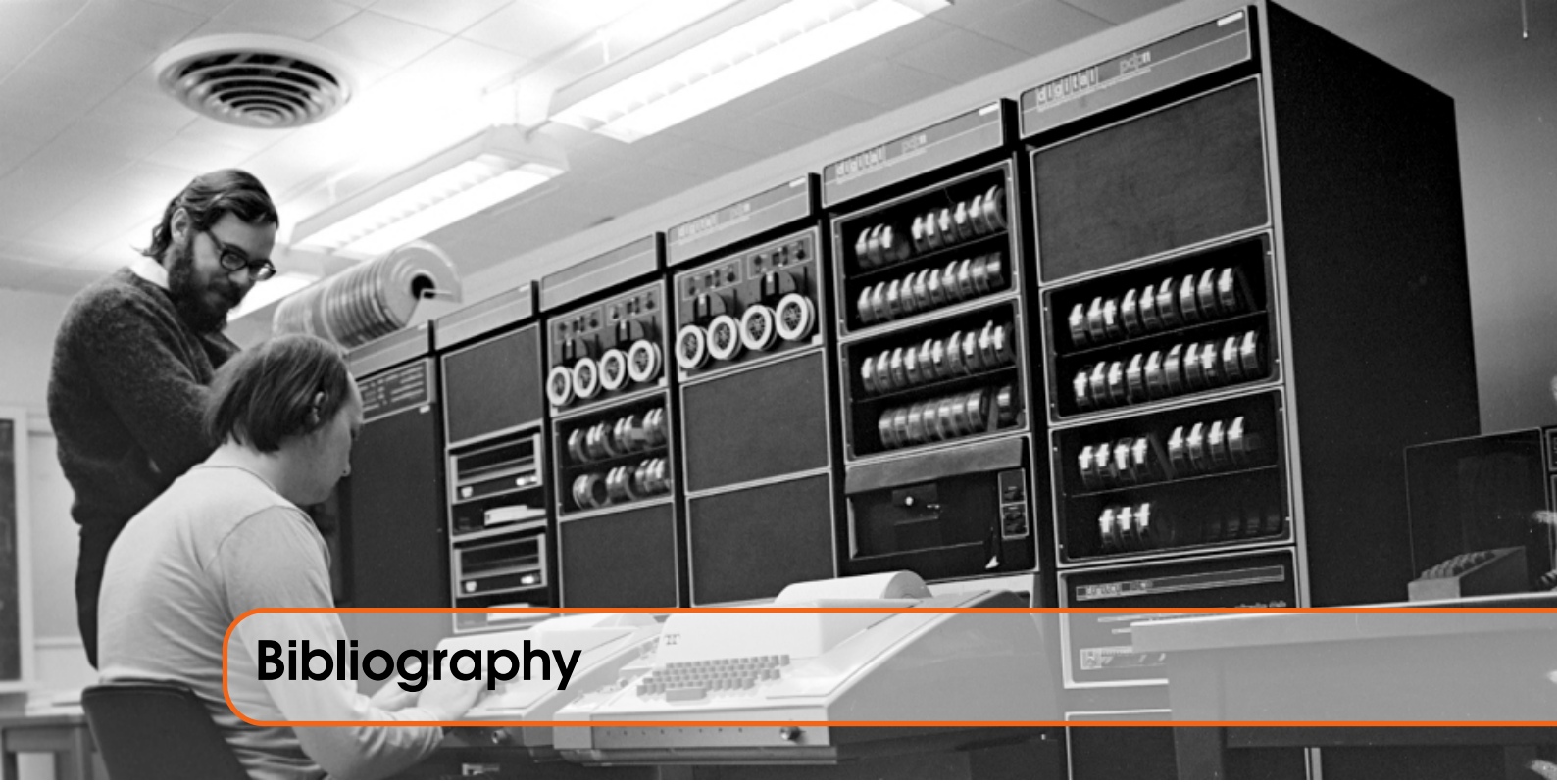
Desde a publicação da primeira edição deste livro, a definição da linguagem **C** sofreu mudanças. Quase todas foram extensões da linguagem original, e foram cuidadosamente projetadas para permanecerem compatíveis com a prática existente; algumas repararam ambiguidades na descrição original; e algumas representam modificações que mudam a prática existente. Muitas das novas facilidades foram anunciadas nos documentos que acompanham os compiladores disponíveis pela **AT&T**, e subsequentemente adotadas por outros fornecedores de compiladores C. Mais recentemente, o comitê da **ANSI**, padronizando a linguagem, incorporou a maioria destas mudanças, e também introduziu outras modificações significativas. Seu relatório foi em parte antecipado por alguns compiladores comerciais mesmo antes da emissão do padrão **C** formal.

Este apêndice resume as diferenças entre a linguagem definida pela primeira edição deste livro, e que será definida pelo Padrão final. Ele trata somente da linguagem em si, e não do seu ambiente ou biblioteca; embora seja uma parte importante do Padrão, há pouca coisa para comparar, pois a primeira edição não tentou ditar um ambiente ou biblioteca.

- O pré-processamento é definido mais cuidadosamente no Padrão do que na primeira edição, e é estendido: ele é baseado explicitamente em códigos; há novos operadores para concatenação de código (##), e criação de strings (#); há novas linhas de controle como *#elif* e *#pragma*; a redeclaração de macros pela mesma sequência de códigos é explicitamente permitida; os parâmetros dentro de strings não são mais substituídos. A divisão de linhas por meio de \é permitida em qualquer lugar, e não apenas em strings e definições de macro. Veja §A.12.
- O significado mínimo de todos os identificadores internos é aumentado para **31** caracteres; o menor significado de ordem dos identificadores com ligação externa permanece seis letras de único tipo. (Muitas implementações fornecem mais.)
- Sequências de trigrama introduzidas por **??** permitem a representação de caracteres inexistentes em alguns conjuntos de caracteres. Os escapes para #\^[]{}|~ são definidos; veja §A.12.1. Observe que a introdução de trigramas pode alterar o significado de strings contendo a sequência **??**.
- Novas palavras-chave (*void*, *const*, *volatile*, *signed*, *enum*) são introduzidas. A recente palavra-chave *entry* é retirada.

- Novas sequências de escape, para uso dentro de constantes de caracteres e literais de string, são definidas. O efeito de seguir \ com um caractere ausente na sequência de escape aprovada é indefinido. Veja §A.2.5.2.
- A mudança trivial favorita de todos: 8 e 9 não são dígitos octais.
- O padrão introduz um grande conjunto de sufixos para tomar explícito o tipo de constantes: **U** ou **L** para inteiros, **F** ou **L** para flutuantes. Ele também retira as regras para o tipo das constantes sem sufixo (§A.2.5).
- Literais de string adjacentes são concatenadas.
- Existe uma notação para literais de string com caracteres estendidos e constantes de caractere; veja §A.2.6.
- Caracteres, bem como outros tipos, podem ser explicitamente declarados para que conduzam ou não um sinal por meio das palavras-chave *signed* ou *unsigned*. A locução *long float* como sinônimo para *double* é retirada, mas *long double* pode ser usado para declarar uma quantidade de ponto flutuante com precisão extra.
- Por algum tempo, o tipo *unsigned char* esteve disponível. O padrão introduz a palavra-chave *signed* para tomar a sinalização explícita para *char* e outros objetos de tipo integral.
- O tipo *void* tem estado há alguns anos disponível na maioria das implementações. O padrão introduz o uso do tipo *void** como tipo apontador genérico; anteriormente *char** desempenhava este papel. Ao mesmo tempo, são decretadas regras explícitas contra a mistura de ponteiros e inteiros, e ponteiros de tipo diferente, sem o uso de moldes.
- O padrão torna explícito um mínimo para as faixas de tipos aritméticos, e exige cabeçalhos (<*limits.h*> e <*float.h*>) dando as características de cada implementação particular.
- Enumerações são novas desde a primeira edição deste livro.
- O padrão adota do **C++** a noção de qualificador de tipo, por exemplo *const* (§A.8.2).
- As strings não são mais modificáveis, e assim podem ser colocadas em memória somente de leitura.
- As "*conversões aritméticas comuns*" são mudadas, essencialmente de "*para inteiros, unsigned sempre vence; para ponto flutuante, sempre use double*" para "*promova para o tipo de menor capacidade possível*". Veja §A.6.5.
- Os antigos operadores de atribuição, como $=+$, realmente acabaram. Além disso, operadores de atribuição são agora símbolos únicos; na primeira edição, eles eram pares, e podiam ser separados por um espaço em branco.
- É revogada a licença de um compilador para tratar operadores matematicamente associativos como computacionalmente associativas.
- Um operador unário $+$ é introduzido por simetria com o unário $-$.
- Um ponteiro para uma função pode ser usado como designador de função sem um operador $*$ explícito. Veja §A.7.3.2.
- Estruturas podem ser atribuídas, passadas a funções e retornadas por funções.
- Aplicar o operador endereço-do a vetores é permitido, e o resultado é um apontador para o vetor.
- O operador *sizeof*, na primeira edição, produzia o tipo *int*; subsequentemente, muitas implementações o tornaram *unsigned*. O padrão torna seu tipo explicitamente dependente da implementação, mas exige que o tipo *size_t* seja definido em um cabeçalho-padrão (<*stddef.h*>). Uma mudança semelhante ocorre no tipo (*ptrdiff_t*) da diferença entre ponteiros. Veja §A.7.4.8 e §A.7.7.
- O operador endereço-de (&) não pode ser aplicado a um objeto declarado *register*, mesmo que a implementação escolha não manter o objeto em um registrador.
- O tipo de uma expressão de deslocamento é aquele do operando esquerdo; o operando da direita não pode promover o resultado. Veja §A.7.8.

- O padrão legaliza a criação de um apontador para logo após o fim de um vetor, e permite aritmética e expressões relacionais com ele; veja §A.7.7.
- O padrão introduz (emprestado de **C++**) a noção de uma declaração de protótipo de função que incorpora os tipos dos parâmetros, e inclui um reconhecimento explícito de funções variantes juntamente com uma forma aprovada de lidar com elas. Veja §§A.7.3.2, A.8.6.3 e B.7. O estilo antigo é ainda aceitável, com restrições.
- Declarações vazias, que não possuem declaradores e não declaram pelo menos uma estrutura, união ou enumeração, são proibidas pelo padrão. Por outro lado, uma declaração com apenas uma etiqueta de estrutura ou união redeclara essa etiqueta mesmo que tenha sido declarada em um escopo mais externo.
- As declarações de dados externos sem qualquer especificador ou qualificador (assim como um declarador vazio) são proibidas.
- Algumas implementações, quando apresentadas com uma declaração *extern* em um bloco mais interno, exportariam a declaração para o restante do arquivo. O padrão deixa claro que o escopo de tal declaração é apenas o bloco.
- O escopo de parâmetros é injetado no comando composto de uma função, de forma que declarações variáveis no nível mais alto da função não podem esconder os parâmetros.
- Os espaços de nome dos identificadores são um pouco diferentes. O padrão agrupa todas as etiquetas em um único espaço de nome, e também introduz um espaço separado para rótulos; veja §A.11.1. Além disso, os nomes de membro são associados à estrutura ou união da qual eles fazem parte. (Esta tem sido uma prática comum por algum tempo).
- Uniões podem ser inicializadas; o inicializador refere-se ao primeiro membro.
- As estruturas, uniões e vetores automáticos podem ser inicializados, embora de uma forma restrita.
- Os vetores de caractere com um tamanho explícito podem ser inicializados por uma literal de string com exatamente a mesma quantidade de caracteres (o \0 é silenciosamente retirado).
- A expressão controladora, além dos rótulos de caso em um switch, podem ter qualquer tipo integral.



# Bibliography

# Índice

## G



## H



## I



## L



## M



## N



## O



## P



## Q



## R



## S



## T



## U



## V



## W