6

Funções

Funções são os blocos de construção de C e o local onde toda a atividade do programa ocorre. Elas são uma das características mais importantes de C.

A Forma Geral de uma Função

A forma geral de uma função é

especificador_de_tipo nome_da_função(lista de parâmetros)
{
corpo da função
}

O *especificador_de_tipo* especifica o tipo de valor que o comando **return** da função devolve, podendo ser qualquer tipo válido. Se nenhum tipo é especificado, o compilador assume que a função devolve um resultado inteiro. A *lista de parâmetros* é uma lista de nomes de variáveis separados por vírgulas e seus tipos associados que recebem os valores dos argumentos quando a função é chamada. Uma função pode não ter parâmetros, neste caso a lista de parâmetros é vazia. No entanto, mesmo que não existam parâmetros, os parênteses ainda são necessários.

Nas declarações de variáveis, você pode declarar muitas variáveis como sendo de um tipo comum, usando uma lista de nomes de variáveis separados por vírgulas. Em contraposição, todos os parâmetros de função devem incluir o tipo e o nome da variável. Isto é, a lista de declaração de parâmetros para uma função tem esta forma geral:

f(tipo nomevar1, tipo nomevar2, ..., tipo nomevarN)

Regras de Escopo de Funções

As *regras de escopo* de uma linguagem são as regras que governam se uma porção de código conhece ou tem acesso a outra porção de código ou dados.

Em C, cada função é um bloco discreto de código. Um código de uma função é privativo àquela função e não pode ser acessado por nenhum comando em uma outra função, exceto por meio de uma chamada à função. (Por exemplo, você não pode usar **goto** para saltar para o meio de outra função.) O código que constitui o corpo de uma função é escondido do resto do programa e, a menos que use variáveis ou dados globais, não pode afetar ou ser afetado por outras partes do programa. Colocado de outra maneira, o código e os dados que são definidos internamente a uma função não podem interagir com o código ou dados definidos em outra função porque as duas funções têm escopos diferentes.

Variáveis que são definidas internamente a uma função são chamadas variáveis locais. Uma variável local vem a existir quando ocorre a entrada da função e ela é destruída ao sair. Ou seja, variáveis locais não podem manter seus valores entre chamadas a funções. A única exceção ocorre quando a variável é declarada com o especificador de tipo de armazenamento **static**. Isso faz com que o compilador trate a variável como se ela fosse uma variável global para fins de armazenamento, mas ainda limita seu escopo para dentro da função. (O Capítulo 2 aborda variáveis globais e locais em profundidade.)

Em C, todas as funções estão no mesmo nível de escopo. Isto é, não é possível definir uma função internamente a uma função. Esta é a razão de C não ser tecnicamente uma linguagem estruturada em blocos.

Argumentos de Funções

Se uma função usa argumentos, ela deve declarar variáveis que aceitem os valores dos argumentos. Essas variáveis são chamadas de *parâmetros formais* da função. Elas se comportam como quaisquer outras variáveis locais dentro da função e são criadas na entrada e destruídas na saída. Como mostra a função seguinte, a declaração de parâmetros ocorre após o nome da função:

```
/* Devolve 1 se c é parte da string s; 0 caso contrário. */
is_in(char *s, char c)
{
  while(*s)
    if(*s==c) return 1;
```

```
    else s++;
  return 0;
}
```

A função **is_in()** tem dois parâmetros: **s** e **c**. Essa função devolve 1 se o caractere **c** faz parte da string **s**; caso contrário, ela devolve 0.

Você deve assegurar-se de que os argumentos usados para chamar a função sejam compatíveis com o tipo de seus parâmetros. Se os tipos são incompatíveis, o compilador não gera uma mensagem de erro, mas ocorrem resultados inesperados. Ao contrário de muitas outras linguagens, C é robusta e geralmente faz alguma coisa com qualquer programa sintaticamente correto, mesmo que o programa contenha incompatibilidades de tipos questionáveis. Por exemplo, se uma função espera um ponteiro mas é chamada com um valor, podem ocorrer resultados inesperados. O uso de protótipos de funções (discutidos em breve) pode ajudar a achar esses tipos de erro.

Como no caso com variáveis locais, você pode fazer atribuições a parâmetros formais ou usá-los em qualquer expressão C permitida. Embora essas variáveis realizem a tarefa especial de receber o valor dos argumentos passados para a função, você pode usá-las como qualquer outra variável local.

Chamada por Valor, Chamada por Referência

Em geral, podem ser passados argumentos para sub-rotinas de duas maneiras. A primeira é *chamada por valor*. Esse método copia o valor de um argumento no parâmetro formal da sub-rotina. Assim, alterações feitas nos parâmetros da sub-rotina não têm nenhum efeito nas variáveis usadas para chamá-la.

Chamada por referência é a segunda maneira de passar argumentos para uma sub-rotina. Nesse método, o endereço de um argumento é copiado no parâmetro. Dentro da sub-rotina, o endereço é usado para acessar o argumento real utilizado na chamada. Isso significa que alterações feitas no parâmetro afetam a variável usada para chamar a rotina.

Com poucas exceções, C usa chamada por valor para passar argumentos. Em geral, isso significa que você não pode alterar as variáveis usadas para chamar a função. (Você aprenderá, mais tarde, neste capítulo, como forçar uma chamada por referência, utilizando um ponteiro para permitir alterações na variável usada na chamada.) Considere o programa seguinte:

```
#include <stdio.h>

int sqr(int x);
```

```
void main(void)
{
  int t=10;

  printf("%d %d", sqr(t), t);
}

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

Neste exemplo, o valor do argumento para **sqr()**, 10, é copiado no parâmetro **x**. Quando a atribuição **x = x*x** ocorre, apenas a variável local **x** é modificada. A variável **t**, usada para chamar **sqr()**, ainda tem o valor 10. Assim, a saída é **100 10**.

Lembre-se de que é uma cópia do valor do argumento que é passada para a função. O que ocorre dentro da função não tem efeito algum sobre a variável usada na chamada.

Criando uma Chamada por Referência

Muito embora a convenção de C de passagem de parâmetros seja por valor, você pode criar uma chamada por referência passando um ponteiro para o argumento. Como isso faz com que o endereço do argumento seja passado para a função, você pode, então, alterar o valor do argumento fora da função.

Ponteiros são passados para as funções como qualquer outra variável. Obviamente, é necessário declarar os parâmetros como do tipo ponteiro. Por exemplo, a função **swap()**, que troca os valores dos seus dois argumentos inteiros, mostra como.

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

  temp = *x; /* salva o valor no endereço x */
  *x = *y; /* põe y em x */
  *y = temp; /* põe x em y */
}
```

swap() é capaz de trocar os valores das duas variáveis apontadas por x e y porque são passados seus endereços (e não seus valores). Daí que, dentro da função, o conteúdo das variáveis pode ser acessado usando as operações padrão de ponteiro. Portanto, o conteúdo das variáveis usadas para chamar a função é trocado.

Lembre-se de que **swap()** (ou qualquer outra função que usa parâmetros de ponteiros) deve ser chamada com o endereço dos argumentos. O programa seguinte mostra a maneira correta de chamar **swap()**:

```
void swap(int *x, int *y);

void main(void)
{
  int i, j;

  i = 10;
  j = 20;

  swap(&i, &j); /* passa os endereços de i e j */
}
```

Neste exemplo, é atribuído 10 à variável **i** e 20 à variável **j**. Em seguida, **swap()** é chamada com os endereços de **i** e **j**. (O operador unário **&** é usado para produzir o endereço das variáveis.) Assim, os endereços de **i** e **j**, não seus valores, são passados para a função **swap()**.

Chamando Funções com Matrizes

Matrizes são examinadas em detalhes no Capítulo 4. No entanto, esta seção discute a operação de passagem de matrizes, como argumentos, para funções, porque é uma exceção à convenção de passagem de parâmetros com chamada por valor.

Quando uma matriz é usada como um argumento para uma função, apenas o endereço da matriz é passado, não uma cópia da matriz inteira. Quando você chama uma função com um nome de matriz, um ponteiro para o primeiro elemento na matriz é passado para a função. (Não se esqueça: em C, um nome de matriz sem qualquer índice é um ponteiro para o primeiro elemento na matriz.) Isso significa que a declaração de parâmetros deve ser de um tipo de ponteiro compatível. Existem três maneiras de declarar um parâmetro que receberá um ponteiro para matriz. Primeiro, ele pode ser declarado como uma matriz, conforme mostrado aqui:

```
/* Imprime alguns números. */
#include <stdio.h>

void display(int num[10]);

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

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

void display(int num[10])
{
  int i;

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

Muito embora o parâmetro **num** seja declarado como uma matriz de inteiros com dez elementos, o compilador C converte-o automaticamente para um ponteiro de inteiros. Isso é necessário porque nenhum parâmetro pode realmente receber uma matriz inteira. Assim, como apenas um ponteiro para matriz é passado, um parâmetro de ponteiro deve estar lá para recebê-lo.

A segunda forma de declarar um parâmetro de matriz é especificá-lo como uma matriz sem dimensão, conforme mostrado aqui:

```
void display(int num[])
{
  int i;

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

Neste caso, **num** é declarado como uma matriz de inteiros de tamanho desconhecido. Como C não fornece nenhuma verificação de limites em matrizes, o tamanho real da matriz é irrelevante para o parâmetro (mas não para o programa, é claro). Esse método de declaração realmente define **num** como um ponteiro de inteiros.

A última forma em que **num** pode ser declarado — a mais comum em programas escritos profissionalmente em C — é como um ponteiro, conforme mostrado a seguir:

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

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

Isso é permitido porque qualquer ponteiro pode ser indexado usando **[]**, como se fosse uma matriz. (Na verdade, matrizes e ponteiros estão intimamente ligados.)

Por outro lado, um elemento de uma matriz pode ser usado como um argumento igual a qualquer outra variável simples. Por exemplo, o programa anterior poderia ser escrito sem passar toda a matriz:

```
/* Imprime alguns números. */
#include <stdio.h>

void display(int num);

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

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

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

Como você pode ver, o parâmetro para **display()** é do tipo **int**. Não é relevante que **display()** seja chamada usando um elemento de matriz, pois apenas um valor da matriz é usado.

É importante entender que, quando uma matriz é usada como um argumento para uma função, seu endereço é passado para a função. Isso é uma exceção à convenção de C no que diz respeito a passar parâmetros. Nesse caso,

o código dentro da função está operando com, e potencialmente alterando, o conteúdo real da matriz usada para chamar a função. Por exemplo, considere a função **print_upper()**, que imprime seu argumento string em maiúsculas:

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

void print_upper(char *string);

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

  gets(s);
  print_upper(s);
}

/* Imprime uma string em maiúsculas. */
void print_upper(char *string)
{
  register int t;

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

Após a chamada a **print_upper()**, o conteúdo da matriz **s** em **main()** estará alterado para maiúsculas. Se não é isso o que você quer, o programa poderia ser escrito dessa forma:

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

void print_upper(char *string);

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

  gets(s);
  print_upper(s);
}
```

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

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

Nesta versão, o conteúdo da matriz **s** permanece inalterado, porque seus valores não são modificados.

A função **gets()** da biblioteca padrão é um exemplo clássico de passagem de matrizes para funções. A função **gets()** da bibllioteca padrão é mais sofisticada e complexa. Porém, a função mais simples **xgets()**, dada a seguir, dá uma idéia de como ela funciona.

```
/* Uma versão muito simples da função gets()
   da biblioteca padrão */
char *xgets(char *s)
{
  char ch, *p;
  int t;

  p = s; /* gets () devolve um ponteiro para s */
  for(t=0; t<80; ++t) {
    ch = getchar();

    switch(ch) {
      case '\n':
        s[t] = '\0'; /* termina a string */

        return p;
      case '\b':
        if(t>0) t--;
        break;
      default:
        s[t] = ch;
    }
  }
  s[80] = '\0';
  return p;
}
```

A função **xgets()** deve ser chamada com um ponteiro para caractere, que pode ser uma variável declarada como um ponteiro para caractere ou o nome de uma matriz de caracteres, que, por definição, é um ponteiro de caractere. Na entrada, **xgets()** estabelece um laço **for** de 0 a 80. Isso evita que strings maiores sejam inseridas pelo teclado. Se mais de 80 caracteres forem inseridos, a função retorna. (A função **gets()** real não tem essa restrição.) Como C não tem verificação interna de limites, você deve assegurar-se de que qualquer variável utilizada para chamar **xgets()** pode aceitar pelo menos 80 caracteres. Ao digitar caracteres no teclado, eles são colocados na string. Se você pressionar a tecla de retrocesso, o contador **t** é reduzido em 1. Quando você pressiona **ENTER**, um caractere nulo é colocado no final da string, sinalizando sua terminação. Como a matriz usada para chamar **xgets()** é modificada, ao retornar ela contém os caracteres digitados.

argc e argv — Argumentos para main()

Algumas vezes é útil passar informações para um programa quando o executamos. Geralmente, você passa informações para a função **main()** via argumentos da linha de comando. Um *argumento da linha de comando* é a informação que segue o nome do programa na linha de comando do sistema operacional. Por exemplo, quando compila programas em C, você digita algo após aviso de comando na tela semelhante a:

cc *nome_programa*

onde *nome_programa* é o programa que você deseja compilar. O nome do programa é passado para o compilador C como um argumento.

Há dois argumentos internos especiais, **argc** e **argv**, que são usados para receber os argumentos da linha de comando. O parâmetro **argc** contém o número de argumentos da linha de comando e é um inteiro. Ele é sempre pelo menos 1 porque o nome do programa é qualificado como primeiro argumento. O parâmetro **argv** é um ponteiro para uma matriz de ponteiros para caractere. Cada elemento nessa matriz aponta para um argumento da linha de comando. Todos os argumentos da linha de comando são strings — quaisquer números terão de ser convertidos pelo programa no formato interno apropriado. Por exemplo, esse programa simples imprime **Ola** e seu nome na tela se você o digitar imediatamente após o nome do programa.

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

void main(int argc, char *argv[])
```

```
{
  if(argc!=2) {
    printf("Você esqueceu de digitar seu nome.\n");
    exit(1);
  }
  printf("Ola %s", argv[1]);
}
```

Se esse programa chamasse **nome** e seu nome fosse Tom, então, para rodar o programa, você deveria digitar **nome Tom**. A resposta do programa seria **Ola Tom**.

Em muitos ambientes, cada argumento da linha de comando deve ser separado por um espaço ou um caractere de tabulação. Vírgulas, pontos-e-vírgulas etc. não são considerados separadores. Por exemplo,

```
run Spot, run
```

é constituído de três strings, enquanto

```
Herb,Rick,Fred
```

é uma única string, uma vez que vírgulas não são separadores legais.

Alguns ambientes permitem que se coloque entre aspas uma string contendo espaços. Isso faz com que a string inteira seja tratada como um único argumento. Verifique o manual do seu sistema operacional para mais detalhes sobre a definição dos parâmetros na linha de comando do seu sistema.

É importante declarar **argv** adequadamente. O método de declaração mais comum é

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

Os colchetes vazios indicam que a matriz é de tamanho indeterminado. Você pode, agora, acessar os argumentos individuais indexando **argv**. Por exemplo, **argv[0]** aponta para a primeira string, que é sempre o nome do programa; **argv[1]** aponta para o primeiro argumento e assim por diante.

Um outro exemplo usando argumentos da linha de comando é o programa chamado **countdown**, mostrado aqui. Ele conta regressivamente a partir do valor especificado na linha de comando e avisa quando chega a 0. Observe que o primeiro argumento contendo o número é convertido em um inteiro pela função padrão **atoi()**. Se a string "display" é o segundo argumento da linha de comando, a contagem regressiva também será mostrada na tela.

```
/* Programa de contagem regressiva. */
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>

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

  if(argc<2) {
    printf("Você deve digitar o valor a contar\n");
    printf("na linha de comando. Tente novamente.\n");
    exit(1);
  }

  if(argc==3 && !strcmp(argv[2], "display")) disp = 1;
  else disp = 0;

  for(count=atoi(argv[1]); count; --count)
    if(disp) printf("%d\n", count);

  putchar('\a'); /* isso irá tocar a campainha na maioria
                    dos computadores */
  printf("Terminou");
}
```

Observe que, se nenhum argumento for especificado, é mostrada uma mensagem de erro. Um programa com argumentos na linha de comando geralmente apresenta instruções se o usuário executou o programa sem inserir a informação apropriada.

Para acessar um caractere individual em uma das strings de comando, acrescente um segundo índice a **argv**. Por exemplo, o próximo programa mostra todos os argumentos com os quais foi chamado, um caractere por vez:

```
#include <stdio.h>

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

  for(t=0; t<argc; ++t) {
    i = 0;
```

```
    while(argv[t][i]) {
      putchar(argv[t][i]);
      ++i;
    }
  }
}
```

Lembre-se: o primeiro índice acessa a string e o segundo acessa os caracteres individuais da string.

Normalmente, você usa **argc** e **argv** para obter os comandos iniciais do seu programa. Teoricamente, você pode ter até 32.767 argumentos, mas a maioria dos sistemas operacionais não permite mais que alguns poucos. Geralmente, você usa esses argumentos para indicar um nome de arquivo ou uma opção. O uso dos argumentos da linha de comando dá aos seus programas uma aparência profissional e facilita o uso do programa em arquivos de lote (*batch files*).

É uma prática comum declarar **main()** como não tendo parâmetro algum, usando a palavra-chave (ou palavra reservada) **void** quando os parâmetros da linha de comando não estão sendo usados. (Esta abordagem é usada pelos programas neste livro). Porém, você pode simplesmente deixar os parênteses vazios.

Os nomes **argc** e **argv** são tradicionais, porém arbitrários. Você pode dar quaisquer nomes a esses dois parâmetros de **main()**. Além disso, alguns compiladores podem suportar argumentos adicionais para **main()**; assegure-se, então, de verificar seu manual do usuário.

O Comando return

O comando **return** tem dois importantes usos. Primeiro, ele provoca uma saída imediata da função que o contém. Isto é, faz com que a execução do programa retorne ao código chamador. Segundo, ele pode ser usado para devolver um valor.

Retornando de uma Função

Existem duas maneiras pelas quais uma função termina a execução e retorna ao código que a chamou. A primeira ocorre quando o último comando da função for executado e, conceitualmente, a chave final do programa (}) é encontrada. (Obviamente, a chave não está realmente presente no código-objeto, mas você pode imaginá-la como se estivesse.) Por exemplo, a função **pr_reverse()** nesse programa, simplesmente escreve a string "Eu gosto de C" de trás para frente na tela.

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

void pr_reverse(char *s);

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

void pr_reverse(char *s)
{
  register int t;

  for(t=strlen(s)-1; t>=0; t--) putchar(s[t]);
}
```

Uma vez que a string tenha sido mostrada, não fica nada por fazer na função **pr_reverse()**, de forma que ela retorna para o lugar de onde foi chamada.

Na realidade, poucas funções usam esse método default de terminar a execução. A maioria das funções adota o comando **return** para encerrar a execução, seja porque um valor deve ser devolvido seja para tornar o código da função mais simples e eficiente.

Lembre-se de que uma função pode ter diversos comandos **return**. Por exemplo, a função **find_substr()**, no programa a seguir, devolve a posição inicial de uma substring dentro de uma string ou, se não for encontrada, a função devolve -1.

```
#include <stdio.h>

int find_substr(char *s1, char *s2);

void main(void)
{
  if(find_substr("C é legal", "é")!=-1)
    printf("a substring não foi encontrada);
}

/* Devolve o índice de s1 em s2. */
find_substr(char *s1, char *s2)
{
  register int t;
  char *p, *p2;
```

```
  for(t=0; s1[t]; t++) {
    p = &s1[t];
    p2 = s2;

    while(*p2 && *p2==*p) {
      p++;
      p2++;
    }
    if(!*p2) return t; /* 1º retorno */
  }
  return -1; /* 2º retorno */
}
```

Retornando Valores

Todas as funções, exceto as do tipo **void**, devolvem um valor. Esse valor é especificado explicitamente pelo comando **return**. Se nenhum comando **return** estiver presente, então o valor de retorno da função será tecnicamente indefinido. (Geralmente, os compiladores C devolvem 0 quando nenhum valor de retorno for especificado explicitamente, mas você não deve contar com isso se há interesse em portabilidade.) Em outras palavras, a partir do momento que uma função não é declarada como **void**, ela pode ser usada como operando em qualquer expressão válida de C. Assim, cada uma das seguintes expressões é válida em C:

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

Porém, uma função não pode ser o destino de uma atribuição. Um comando tal como

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

está errado. O compilador C indicará isso como um erro e não compilará programas que contenham um comando como esse.

Quando você escreve programas, suas funções geralmente serão de três tipos. O primeiro tipo é simplesmente computacional. Essas funções são projetadas especificamente para executar operações em seus argumentos e devolver um valor. Uma função computacional é uma função "pura". Exemplos são as funções da biblioteca padrão **sqrt()** e **sin()**, que calculam a raiz quadrada e o seno de seus argumentos.

O segundo tipo de função manipula informações e devolve um valor que simplesmente indica o sucesso ou a falha dessa manipulação. Um exemplo é a função da biblioteca padrão **fclose()**, que é usada para fechar um arquivo. Se a operação de fechamento for bem-sucedida, a função devolverá 0; se a operação não for bem-sucedida, ela devolverá um código de erro.

O último tipo não tem nenhum valor de retorno explícito. Em essência, a função é estritamente de procedimento e não produz nenhum valor. Um exemplo é **exit()**, que termina um programa. Todas as funções que não devolvem valores devem ser declaradas como retornando o tipo **void**. Ao declarar uma função como **void**, você a protege de ser usada em uma expressão, evitando uma utilização errada acidental.

Algumas vezes, funções que, na realidade, não produzem um resultado relevante de qualquer forma devolvem um valor. Por exemplo, **printf()** devolve o número de caracteres escritos. Entretanto, é muito incomum encontrar um programa que realmente verifique isso. Em outras palavras, embora todas as funções, exceto aquelas do tipo **void**, devolvam valores, você não tem necessariamente de usar o valor de retorno. Uma questão envolvendo valores de retorno de funções é: "Eu não tenho de atribuir esse valor a alguma variável já que um valor está sendo devolvido?". A resposta é não. Se não há nenhuma atribuição especificada, o valor de retorno é simplesmente descartado. Considere o programa seguinte, que utiliza a função **mul()**:

```
#include <stdio.h>

int mul(int a, int b);

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

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

mul(int a, int b)
{
  return a*b;
}
```

Na linha 1, o valor de retorno de **mul()** é atribuído a **z**. Na linha 2, o valor de retorno não é realmente atribuído, mas é usado pela função **printf()**. Finalmente, na linha 3, o valor de retorno é perdido porque não é atribuído a outra variável nem usado como parte de uma expressão.

Funções Que Devolvem Valores Não-Inteiros

Quando o tipo da função não é explicitamente declarado, o compilador C atribui automaticamente a ela o tipo padrão, que é **int**. Para muitas funções em C, esse tipo padrão é aplicável. No entanto, quando é necessário um tipo de dado diferente, o processo envolve dois passos. Primeiro, deve ser dada à função um especificador de tipo explícito. Segundo, o tipo da função deve ser identificado antes da primeira chamada feita a ela. Apenas assim C pode gerar um código correto para funções que não devolvem valores inteiros.

As funções podem ser declaradas como retornando qualquer tipo de dado válido em C. O método da declaração é semelhante à declaração de variáveis: o especificador de tipo precede o nome da função. O especificador de tipo informa ao compilador que tipo de dado a função devolverá. Essa informação é crítica para o programa rodar corretamente, porque tipos de dados diferentes têm tamanhos e representações internas diferentes.

Antes que uma função que não retorne um valor inteiro possa ser usada, seu tipo deve ser declarado ao programa. Isso porque, a menos que informado em contrário, o compilador C assume que uma função devolve um valor inteiro. Se seu programa usa uma função que devolve um tipo diferente antes da sua declaração, o compilador gera erroneamente o código para a chamada. Para evitar isso, você deve usar uma forma especial de declaração, perto do início do seu programa, informando ao compilador o tipo de dado que sua função realmente devolverá.

Existem duas maneiras de declarar uma função antes de ela ser usada: a forma tradicional e o moderno método de protótipos. A abordagem tradicional era o único método disponível quando C foi inventada, mas agora está obsoleto. Os protótipos foram acrescentados pelo padrão C ANSI. A abordagem tradicional é permitida pelo padrão ANSI para assegurar compatibilidade com códigos mais antigos, mas novos usos são desencorajados. Muito embora seja antiquado, muitos milhares de programas ainda o usam, de forma que você deve familiarizar-se com ele. Além disso, o método com protótipos é basicamente uma extensão do conceito tradicional.

Nesta seção, examinaremos a abordagem tradicional. Embora desatualizada, muitos dos programas existentes ainda a utilizam. Além disso, um método do protótipo é basicamente uma extensão do conceito tradicional. (Os protótipos de função são discutidos na próxima seção.)

Com a abordagem tradicional, você especifica o tipo e o nome da função próximos ao início do programa para informar ao compilador que uma função devolverá algum tipo de valor diferente de um inteiro, como ilustrado aqui:

```
#include <stdio.h>

float sum(); /* identifica a função */
float first, second;

void main(void)
{
  first = 123.23;
  second = 99.09;

  printf("%f", sum());
}

float sum()
{
  return first + second;
}
```

A primeira declaração da função informa ao compilador que **sum()** devolve um tipo de dado em ponto flutuante. Isso permite que o compilador gere corretamente o código para a chamada a **sum()**. Sem a declaração, o compilador indicaria um erro de incompatibilidade de tipos.

O comando tradicional de declaração de tipos tem a forma geral

especificador_de_tipo nome_da_função();

Mesmo que a função tenha argumentos, eles não constam na declaração de tipo.

Sem o comando de declaração de tipo, ocorreria um erro de incompatibilidade entre o tipo de dado que a função devolve e o tipo de dado que a rotina chamadora espera. Os resultados serão bizarros e imprevisíveis. Se ambas as funções estão no mesmo arquivo, o compilador descobre a incompatibilidade de tipos e não compila o programa. Contudo, se as funções estão em arquivos diferentes, o compilador não detecta o erro. Nenhuma verificação de tipos é feita durante o tempo de linkedição ou tempo de execução, apenas em tempo de compilação. Por essa razão, você deve assegurar-se de que ambos os tipos são compatíveis.

NOTA: Quando um caractere é devolvido por uma função declarada como sendo do tipo **int**, *o valor caractere é convertido em um inteiro. Visto que C faz a conversão de caractere para inteiro, e vice-versa, uma função que devolve um caractere geralmente não é declarada como devolvendo um valor caractere. O programador confia na conversão padrão de caractere em inteiro e vice-versa. Esse tipo de coisa é freqüentemente encontrado em códigos em C mais antigos e tecnicamente não é considerado um erro.*

Protótipos de Funções

O padrão C ANSI expandiu a declaração tradicional de função, permitindo que a quantidade e os tipos dos argumentos das funções sejam declarados. A definição expandida é chamada *protótipo de função*. Protótipos de funções não faziam parte da linguagem C original. Eles são, porém, um dos acréscimos mais importantes do ANSI à C. Neste livro, todos os exemplos incluem protótipos completos das funções. Protótipos permitem que C forneça uma verificação mais forte de tipos, algo como aquela fornecida por linguagens como Pascal. Quando você usa protótipos, C pode encontrar e apresentar quaisquer conversões de tipos ilegais entre o argumento usado para chamar uma função e a definição de seus parâmetros. C também encontra diferenças entre o número de argumentos usados para chamar a função e o número de parâmetros da função.

A forma geral de uma definição de protótipo de função é

tipo nome_func(tipo nome_param1, tipo nome_param2,...,
tipo nome_paramN);

O uso dos nomes dos parâmetros é opcional. Porém, eles habilitam o compilador a identificar qualquer incompatibilidade de tipos por meio do nome quando ocorre um erro, de forma que é uma boa idéia incluí-los.

Por exemplo, o programa seguinte produz uma mensagem de erro porque ele tenta chamar **sqr_it()** com um argumento inteiro em vez do exigido ponteiro para inteiro. (Você não pode transformar um inteiro em um ponteiro.)

```
/* Esse programa usa um protótipo de função para forçar uma
   verificação forte de tipos. */

void sqr_it(int *i); /* protótipo */

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

```
  x = 10;
  sqr_it(x);  /* incompatibilidade de tipos */
}

void sqr_it(int *i)
{
  *i = *i * *i;
}
```

Devido à necessidade de compatibilidade com a versão original de C, algumas regras especiais são aplicadas aos protótipos de funções. Primeiro, quando o tipo de retorno de uma função é declarado sem nenhuma informação de protótipo, o compilador simplesmemte assume que nenhuma informação sobre os parâmetros é dada. No que se refere ao compilador, a função pode ter diversos ou nenhum parâmetro. Assim, como pode ser dado um protótipo a uma função que não tem nenhum parâmetro? A resposta é: quando uma função não tem parâmetros, seu protótipo usa **void** dentro dos parênteses. Por exemplo, se uma função chamada **f()** devolve um **float** e não tem parâmetros, seu protótipo será:

```
float f(void);
```

Isso informa ao compilador que a função não tem parâmetros e qualquer chamada à função com parâmetros é um erro.

O uso de protótipos afeta a promoção automática de tipos de C. Quando uma função sem protótipo é chamada, todos os caracteres são convertidos em inteiros e todos os **floats** em **doubles**. Essas promoções um tanto estranhas estão relacionadas com as características do ambiente original em que C foi desenvolvida. No entanto, se a função tem protótipo, os tipos especificados no protótipo são mantidos e não ocorrem promoções de tipo.

Protótipos de funções ajudam a detectar erros antes que eles ocorram. Além disso, eles auxiliam a verificar se seu programa está funcionando corretamente, não permitindo que funções sejam chamadas com argumentos inconsistentes.

Tenha um fato em mente: embora o uso de protótipos de função seja bastante recomendado, tecnicamente não é errado que uma função não tenha protótipos. Isso é necessário para suportar códigos C sem protótipos. Todavia, seu código deve, em geral, incluir total informação de protótipos.

NOTA: Embora os protótipos sejam opcionais em C, eles são exigidos pela sucessora de C: C++.

Retornando Ponteiros

Embora funções que devolvem ponteiros sejam manipuladas da mesma forma que qualquer outro tipo de função, alguns conceitos importantes precisam ser discutidos.

Ponteiros para variáveis não são variáveis e tampouco inteiros sem sinal. Eles são o endereço na memória de um certo tipo de dado. A razão para a distinção deve-se ao fato de a aritmética de ponteiros ser feita relativa ao tipo de base. Por exemplo, se um ponteiro inteiro é incrementado, ele conterá um valor que é maior que o seu anterior em 2 (assumindo inteiros de 2 bytes). Em geral, cada vez que um ponteiro é incrementado, ele aponta para o próximo item de dado do tipo correspondente. Desde que cada tipo de dado pode ter um comprimento diferente, o compilador deve saber para que tipo de dados o ponteiro está apontando. Por esta razão, uma função que retorna um ponteiro deve declarar explicitamente qual tipo de ponteiro ela está retornando.

Para retornar um ponteiro, deve-se declarar uma função como tendo tipo de retorno ponteiro. Por exemplo, esta função devolve um ponteiro para a primeira ocorrência do caractere **c** na string **s**:

```
/* Devolve um ponteiro para a primeira ocorrência de c em s. */
char *match(char c, char *s)
{
  while(c!=*s && *s) s++;
  return(s);
}
```

Se nenhuma coincidência for encontrada, é devolvido um ponteiro para o terminador nulo. Aqui está um programa pequeno que usa **match()**:

```
#include <stdio.h>

char *match(char c, char *s); /* protótipo */

void main(void)
{
  char s[80], *p, ch;
  gets(s);
  ch = getchar();
  p = match(ch, s);

  if(*p) /* encontrou */
```

```
    printf("%s ", p);
  else
    printf("Não encontrei.");
}
```

Esse programa lê uma string e, em seguida, um caractere. Se o caractere está na string, o programa imprime a string do ponto em que há a coincidência. Caso contrário, ele imprime **Não encontrei**.

Funções do Tipo void

Um dos usos de **void** é declarar explicitamente funções que não devolvem valores. Isso evita seu uso em expressões e ajuda a afastar um mau uso acidental. Por exemplo, a função **print_vertical()** imprime seu argumento string verticalmente para baixo, do lado da tela. Visto que não devolve nenhum valor, ela é declarada como **void**.

```
void print_vertical(char *str)
{
  while(*str)
    printf(%c\n", *str++);
}
```

Antes de poder usar qualquer função **void**, você deve declarar seu protótipo. Se isso não for feito, C assumirá que ela devolve um inteiro e, quando o compilador encontrar de fato a função, ele declarará um erro de incompatibilidade. O programa seguinte mostra um exemplo apropriado que imprime verticalmente na tela um único argumento da linha de comando:

```
#include <stdio.h>

void print_vertical(char *str); /* protótipo */

void main(int argc, char *argv[])
{
  if(argc) print_vertical(argv[1]);
}

void print_vertical(char *str)
{
  while(*str)
    printf("%c\n", *str++);
}
```

Antes que o padrão C ANSI definisse **void**, funções que não devolviam valores simplesmente eram assumidas como do tipo **int** por padrão. Portanto, não fique surpreso ao ver muitos exemplos disto em códigos mais antigos.

O Que main() Devolve?

De acordo com o padrão ANSI, a função **main()** devolve um inteiro para o processo chamador, que é, geralmente, o sistema operacional. Devolver um valor em **main()** é equivalente a chamar **exit()** com o mesmo valor. Se **main()** não devolve explicitamente um valor, o valor passado para o processo chamador é tecnicamente indefinido. Na prática, a maioria dos compiladores C devolve 0, mas não conte com isso se há interesse em portabilidade.

Você também pode declarar **main()** como **void** se ela não devolve um valor. Alguns compiladores geram uma mensagem de advertência, se a função não é declarada como **void** e também não devolve um valor.

Recursão

Em C, funções podem chamar a si mesmas. A função é *recursiva* se um comando no corpo da função a chama. Recursão é o processo de definir algo em termos de si mesmo e é, algumas vezes, chamado de *definição circular*.

Um exemplo simples de função recursiva é **factr()**, que calcula o fatorial de um inteiro. O fatorial de um número **n** é o produto de todos os números inteiros entre 1 e **n**. Por exemplo, fatorial de 3 é 1 x 2 x 3, ou seja, 6. Tanto **factr()** como sua equivalente iterativa são mostradas aqui:

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

  if (n==1) return (1);
  answer = factr(n-1)*n; /* chamada recursiva */
  return(answer);
}

fact(int n)  /* não-recursiva */
{
  int t, answer;
```

```
    answer = 1;

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

      return(answer);
}
```

A versão não-recursiva de **fact()** deve ser clara. Ela usa um laço que é executado de 1 a **n** e multiplica progressivamente cada número pelo produto móvel.

A operação de **factr()** recursiva é um pouco mais complexa. Quando **factr()** é chamada com um argumento de 1, a função devolve 1. Caso contrário, ela devolve o produto de **factr(n-1)*n**. Para avaliar essa expressão, **factr()** é chamada com **n-1**. Isso acontece até que **n** se iguale a 1 e as chamadas à função começem a retornar.

Calculando o fatorial de 2, a primeira chamada a **factr()** provoca uma segunda chamada com o argumento 1. Essa chamada retorna 1, que é, então, multiplicado por 2 (o valor original de **n**). A resposta, então, é 2. (Você pode achar interessante inserir comandos **printf()** em **factr()** para ver o nível de cada chamada e quais são as respostas intermediárias.)

Quando uma função chama a si mesma, novos parâmetros e variáveis locais são alocados na pilha e o código da função é executado com essas novas variáveis. Uma chamada recursiva não faz uma nova cópia da função; apenas os argumentos são novos. Quando cada função recursiva retorna, as variáveis locais e os parâmetros são removidos da pilha e a execução recomeça do ponto da chamada à função dentro da função.

A maioria das funções recursivas não minimiza significativamente o tamanho do código nem melhora a utilização da memória. Além disso, as versões recursivas da maioria das rotinas podem ser executadas um pouco mais lentamente que suas equivalentes iterativas devido às repetidas chamadas à função. De fato, muitas chamadas recursivas a uma função podem provocar um estouro da pilha. Como o armazenamento para os parâmetros da função e variáveis locais está na pilha e cada nova chamada cria uma nova cópia dessas variáveis, a pilha pode provavelmente escrever sobre outra memória de dados ou de programa. Contudo, você possivelmente nunca terá de se preocupar com isso, a menos que uma função recursiva seja executada de forma desenfreada.

A principal vantagem das funções recursivas é que você pode usá-las para criar versões mais claras e simples de vários algoritmos. Por exemplo, o QuickSort, na Parte 3, é muito difícil de implementar numa forma iterativa. Além disso, alguns problemas, especialmente aqueles relacionados com inteligência ar-

tificial, resultaram em soluções recursivas. Finalmente, algumas pessoas parecem pensar recursivamente com mais facilidade que iterativamente.

Ao escrever funções recursivas, você deve ter um comando **if** em algum lugar para forçar a função a retornar sem que a chamada recursiva seja executada. Se você não o fizer, a função nunca retornará quando chamada. Omitir o **if** é um erro comum quando se escrevem funções recursivas. Use **printf()** e **getchar()** deliberadamente durante o desenvolvimento do programa de forma que você possa ver o que está acontecendo e encerrar a execução se localizar um erro.

Declarando uma Lista de Parâmetros de Extensão Variável

Em C, você pode especificar uma função que possui a quantidade e os tipos de parâmetros variáveis. O exemplo mais comum é **printf()**. Para informar ao compilador que um número desconhecido de parâmetros será passado para uma função, você deve terminar a declaração dos seus parâmetros usando três pontos. Por exemplo, esta declaração especifica que **func()** terá pelo menos dois parâmetros inteiros e um número desconhecido (incluindo 0) de parâmetros após eles.

```
func(int a, int b, ...);
```

Essa forma de declaração também é usada por um protótipo de função.

Qualquer função que use um número variável de argumentos deve ter pelo menos um argumento verdadeiro. Por exemplo, isto está incorreto:

```
func(...);
```

Para mais informações sobre número e tipos variáveis, veja a Parte 2, sobre a função **va_arg()** da biblioteca C padrão.

Declaração de Parâmetros de Funções Moderna *Versus* Clássica

C originalmente usava um método de declaração de parâmetros diferente, algumas vezes chamado de forma *clássica*. Este livro usa a abordagem de declaração chamada de forma *moderna*. O padrão ANSI para C suporta as duas formas, mas

recomenda fortemente a forma moderna. Porém, você deve saber a forma clássica porque, literalmente, milhões de linhas de código já existentes a usam! (Além disso, muitos programas usam essa forma porque ela funciona com todos os compiladores — mesmo os antigos.)

A declaração clássica de parâmetros de funções consiste em duas partes: uma lista de parâmetros, que ficam dentro dos parênteses que seguem o nome da função, e as declarações reais dos parâmetros, que ficam entre o fecha-parênteses e o abre-chaves da função. A forma geral da declaração clássica é

tipo nome_func(param1, param2,...paramN)
tipo param1;
tipo param2;
.
.
.
tipo paramN;
{
código da função
}

Por exemplo, esta declaração moderna:

```
float f(int a, int b, char ch)
{
  /*...*/
}
```

irá se parecer com isto na sua forma clássica:

```
float f(a, b, ch)
int a, b;
char ch;
{
  /*...*/
}
```

Observe que a forma clássica pode suportar mais de um parâmetro em uma lista após o nome do tipo.

Lembre-se de que a forma clássica de declaração de parâmetro está obsoleta. Contudo, seu compilador ainda pode compilar programas mais antigos que usam a forma clássica sem qualquer problema. Isso permite a manutenção de códigos mais antigos.

Questões sobre a Implementação

Há uns poucos pontos a lembrar, quando se criam funções em C, que afetam sua eficiência e usabilidade. Essas questões são o tópico desta seção.

Parâmetros e Funções de Propósito Geral

Uma função de propósito geral é aquela que será usada, em uma variedade de situações, talvez por muitos outros programadores. Tipicamente, você não deve basear funções de propósito geral em dados globais. Todas as informações de que uma função precisa devem ser passadas para ela por meio de seus parâmetros. Quando isso não é possível, você deve usar variáveis estáticas.

Além de tornar suas funções de propósito geral, os parâmetros deixam seu código legível e menos suscetível a erros resultantes de efeitos colaterais.

Eficiência

Funções são os blocos de construção de C e são cruciais para todos os programas, exceto os mais simples. Entretanto, em certas aplicações especializadas, você talvez precise eliminar uma função e substituí-la por código *em linha* (*in-line*). Código em linha é o equivalente aos comandos da função usados sem uma chamada à função. Deve-se usar código em linha em lugar de chamadas a funções apenas quando o tempo de execução é crítico.

Código em linha é mais rápido que a chamada a uma função por duas razões. Primeiro, uma instrução CALL leva tempo para ser executada. Segundo, se há argumentos para passar, eles devem ser colocados na pilha, o que também toma tempo. Para a maioria das aplicações, esse aumento muito pequeno no tempo de execução não é significativo. Mas, se for, lembre-se de que cada chamada à função usa um tempo que poderia ser economizado se o código da função fosse colocado em linha. Por exemplo, seguem duas versões de um programa que imprime o quadrado dos números de 1 a 10. A versão em linha é executada mais rapidamente que a outra porque a chamada à função toma tempo.

em linha

```
#include <stdio.h>

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

chamada à função

```
#include <stdio.h>
int sqr(int a);
void main(void)
{
   int x;
```

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

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

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

Bibliotecas e Arquivos

Uma vez que tenha escrito uma função, pode fazer três coisas com ela: você pode deixá-la no mesmo arquivo da função main(); pode colocá-la em um arquivo separado com outras funções que você escreveu; ou pode colocá-la em uma biblioteca. Nesta seção, discutimos alguns tópicos relacionados com essas opções.

Arquivos Separados

Ao se trabalhar em um grande programa, uma das tarefas mais frustrantes porém comuns é procurar em cada arquivo para encontrar onde determinada função foi colocada. Uma organização preliminar ajudará a evitar esse tipo de problema.

Primeiro, agrupe *todas* as funções que estão conceitualmente relacionadas em um arquivo. Por exemplo, se você está escrevendo um editor de texto, pode colocar todas as funções para exclusão de texto em um arquivo, todas para procura de texto em outro e assim por diante.

Segundo, ponha todas as funções de uso geral juntas. Por exemplo, em um programa de banco de dados, as funções de formatação de entrada/saída são usadas por diversas outras funções e devem estar em um arquivo separado.

Terceiro, agrupe todas as funções de nível mais alto em um arquivo separado ou, se houver espaço, no arquivo **main()**. As funções de nível superior são usadas para iniciar a atividade geral do programa, essencialmente definindo a operação do programa.

Bibliotecas

Tecnicamente, uma biblioteca de funções é diferente de um arquivo de funções compilado separadamente. Quando as rotinas em uma biblioteca são linkeditadas com o restante do seu programa, apenas as funções que seu programa realmente

usa são carregadas e linkeditadas. Em um arquivo compilado separadamente, todas as funções são carregadas e linkeditadas com seu programa. Para a maioria dos arquivos que cria, você provavelmente estará interessado em ter todas as funções no arquivo. No caso de uma biblioteca C padrão, você nunca iria querer todas as funções linkeditadas com seu programa, porque o código-objeto seria enorme!

Há momentos em que você pode desejar a criação de uma biblioteca. Por exemplo, suponha que você tenha escrito um conjunto especializado de funções estatísticas. Você não gostaria de carregar todas essas funções se seu programa precisasse apenas encontrar a média de um conjunto de valores. Nesse caso, uma biblioteca seria útil.

A maioria dos compiladores C inclui instruções para criar uma biblioteca. Uma vez que esse processo varia de compilador para compilador, estude seu manual do usuário para determinar que procedimento você deve seguir.

De Que Tamanho Deve Ser um Arquivo de Programa?

Em virtude de C permitir compilação separada, a questão do tamanho ótimo para um arquivo naturalmente cresce. Isso é importante porque o tempo de compilação está diretamente relacionado com o tamanho do arquivo que está sendo compilado. Geralmente, o processo de linkedição é muito menor que a compilação e elimina a necessidade de constantemente recompilar o código em que se está trabalhando. Por outro lado, manter uma organização de múltiplos arquivos pode ser trabalhoso.

O tamanho que um arquivo deve ter é diferente para todo usuário, todo compilador e todo ambiente operacional. Porém, como regra geral, nenhum arquivo-fonte deve ser maior que 10.000 ou 15.000 bytes. Além desse tamanho, deve-se dividir o arquivo em um ou mais arquivos.