3

Comandos de Controle do Programa

Este capítulo discute os ricos e variados comandos de controle do programa em C. O padrão ANSI divide os comandos de C nestes grupos:

- Seleção
- Iteração
- Desvio
- Rótulo
- Expressão
- Bloco

Estão incluídos nos comando de seleção **if** e **switch**. (O termo "comando condicional" é freqüentemente usado em lugar de "comando de seleção". No entanto, o padrão ANSI usa "seleção", como também este livro.) Os comandos de iteração são **while**, **for** e **do-while**. São também normalmente chamados de comandos de laço. Os comandos de salto ou desvio são **break**, **continue**, **goto** e **return**. Os comandos de rótulo são **case** e **default** (discutidos juntamente com o comando **switch**) e o comando **label** (discutido com **goto**). Sentenças de expressão são aquelas compostas por uma expressão C válida. Sentenças de bloco são simplesmente blocos de código. (Lembre-se de que um bloco começa com um { e termina com um }.)

Como muitos comandos em C contam com a saída de alguns testes condicionais, começaremos revendo os conceitos de verdadeiro e falso em C.

Verdadeiro e Falso em C

Muitos comandos em C contam com um teste condicional que determina o curso da ação. Uma expressão condicional chega a um valor verdadeiro ou falso. Em C, ao contrário de muitas outras linguagens, um valor verdadeiro é qualquer valor diferente de zero, incluindo números negativos. Um valor falso é 0. Esse método para verdadeiro e falso permite que uma ampla gama de rotinas sejam codificadas de forma extremamente eficiente, como você verá em breve.

Comandos de Seleção

C suporta dois tipos de comandos de seleção: **if** e **switch**. Além disso, o operador **?** é uma alternativa ao **if** em certas circunstâncias.

if

A forma geral da sentença **if** é

if (*expressão*) *comando*;
else *comando*;

onde *comando* pode ser um único comando, um bloco de comandos ou nada (no caso de comandos vazios). A cláusula **else** é opcional.

Se a *expressão* é verdadeira (algo diferente de 0), o comando ou bloco que forma o corpo do **if** é executado; caso contrário, o comando ou bloco que é o corpo do **else** (se existir) é executado. Lembre-se de que apenas o código associado ao **if** ou o código associado ao **else** será executado, nunca ambos.

O comando condicional controlando o **if** deve produzir um resultado escalar. Um *escalar* é um inteiro, um caractere ou tipo de ponto flutuante. No entanto, é raro usar um número em ponto flutuante para controlar um comando condicional, porque isso diminui consideravelmente a velocidade de execução. (A CPU executa diversas instruções para efetuar uma operação em ponto flutuante. Ela usa relativamente poucas instruções para efetuar uma operação com caractere ou inteiro.)

Por exemplo, considere o programa a seguir, que é uma versão muito simples do jogo de adivinhar o "número mágico". Ele imprime a mensagem "** **Certo** **" quando o jogador acerta o número mágico. Ele produz o número mágico usando o gerador de números randômicos de C, que devolve um número arbitrário entre 0 e **RAND-MAX** (que define um valor inteiro que é 32.767 ou maior) exige o arquivo de cabeçalho **stdlib.h**.

```
/* Programa de números mágicos #1. */
#include <stdio.h>
#include <stdlib.h>

void main(void)
{
  int magic;  /* número mágico */
  int guess;  /* palpite do usuário */

  magic = rand();  /* gera o número mágico */

  printf("adivinhe o número mágico: ");
  scanf("%d", &guess);

  if(guess == magic) printf("** Certo **");
}
```

Levando o programa do número mágico adiante, a próxima versão ilustra o uso da sentença **else** para imprimir outra mensagem em resposta a um número errado.

```
/* Programa de números mágicos #2. */
#include <stdio.h>
#include <stdlib.h>

void main(void)
{
  int magic;  /* número mágico */
  int guess;  /* palpite do usuário */

  magic = rand();  /* gera o número mágico */

  printf("adivinhe o número mágico: ");
  scanf("%d", &guess);

  if(guess == magic) printf("** Certo **");
  else printf("Errado");
}
```

ifs Aninhados

Um **if** aninhado é um comando **if** que é o objeto de outro **if** ou **else**. **if**s aninhados são muito comuns em programação. Em C, um comando **else** sempre se refere

ao comando **if** mais próximo, que está dentro do mesmo bloco do **else** e não está associado a outro **if**. Por exemplo

```
if(i)
{
  if(j) comando 1;
  if(k) comando 2; /* este if */
  else  comando 3; /* está associado a este else */
}
else comando 4;  /* associado a if(i) */
```

Como observado, o último **else** não está associado a **if(j)** porque não pertence ao mesmo bloco. Em vez disso, o último **else** está associado ao **if(i)**. O **else** interno está associado ao **if(k)**, que é o **if** mais próximo.

O padrão C ANSI especifica que pelo menos 15 níveis de aninhamento devem ser suportados. Na prática, a maioria dos compiladores permite substancialmente mais.

Você pode usar um **if** aninhado para melhorar o programa do número mágico dando ao jogador uma realimentação sobre um palpite errado.

```
/* Programa de números mágicos #3. */
#include <stdio.h>
#include <stdlib.h>

void main(void)
{
  int magic;  /* número mágico */
  int guess;  /* palpite do usuário */

  magic = rand();  /* gera o número mágico */

  printf("Adivinhe o número mágico: ");
  scanf("%d", &guess);

  if(guess == magic) {
    printf("** Certo **");
    printf(" %d é o número mágico\n", magic);
  }
  else {
    printf("Errado, ");
    if(guess > magic)  printf("muito alto\n");
    else printf("muito baixo\n");
  }
}
```

A Escada if-else-if

Uma construção comum em programação é a forma *if-else-if*, algumas vezes chamada de *escada if-else-if* devido a sua aparência. A sua forma geral é

```
if(expressão)comando;
else
  if(expressão)comando;
  else
    if(expressão)comando;
.
.
.
      else comando;
```

As condições são avaliadas de cima para baixo. Assim que uma condição verdadeira é encontrada, o comando associado a ela é executado e desvia do resto da escada. Se nenhuma das condições for verdadeira, então o último **else** é executado. Isto é, se todos os outros testes condicionais falham, o último comando **else** é efetuado. Se o último **else** não está presente, nenhuma ação ocorre se todas as condições são falsas.

Embora seja tecnicamente correta, o recuo da escada if-else-if anterior pode ser excessivamente profundo. Por essa razão, a escada if-else-if é geralmente recuada desta forma:

```
if(expressão)
  comando;
else if(expressão)
  comando;
else if(expressão)
  comando;
.
.
.
else
  comando;
```

Usando uma escada if-else-if, o programa do número mágico torna-se:

```
/* Programa de números mágicos #4. */
#include <stdio.h>
#include <stdlib.h>

void main(void)
```

```
{
  int magic;  /* número mágico */
  int guess;  /* palpite do usuário */

  magic = rand();  /* gera o número mágico */

  printf("Adivinhe o número mágico: ");
  scanf("%d", &guess);

  if(guess == magic) {
    printf("** Certo **");
    printf("%d é o número mágico", magic);
  }
  else if(guess > magic)
   printf("Errado, muito alto");
  else printf("Errado, muito baixo");
}
```

O ? Alternativo

Você pode usar o operador **?** para substituir comandos **if-else** na forma geral:

if(*condição*) *expressão;*
else *expressão;*

Contudo, os corpos de **if** e **else** devem ser uma expressão simples — nunca um outro comando de C.

O **?** é chamado de um *operador ternário* porque ele requer três operandos. Ele tem a seguinte forma geral

Exp1 ? *Exp2* : *Exp3*

onde *Exp1*, *Exp2* e *Exp3* são expressões. Note o uso e o posicionamento dos dois-pontos.

O valor de uma expressão **?** é determinada como segue: *Exp1* é avaliada. Se for verdadeira, *Exp2* será avaliada e se tornará o valor da expressão **?** inteira. Se *Exp1* é falsa, então *Exp3* é avaliada e se torna o valor da expressão. Por exemplo, considere

```
x = 10;
y = x>9 ? 100 : 200;
```

Nesse exemplo, o valor 100 é atribuido a **y**. Se **x** fosse menor que 9, **y** teria recebido o valor 200. O mesmo código escrito com o comando **if-else** seria

```
x = 10;
if (x>9) y = 100;
else y = 200;
```

O seguinte programa usa o operador **?** para elevar ao quadrado um valor inteiro digitado pelo usuário. Contudo, este programa preserva o sinal (10 ao quadrado é 100 e -10 ao quadrado é -100).

```
#include <stdio.h>

void main(void)
{
  int isqrd, i;

  printf("Digite um número: ");
  scanf("%d", &i);

  isqrd = i>0 ? i*i : -(i*i);

  printf("%d ao quadrado é %d", i, isqrd);
}
```

O uso do operador **?** para substituir comandos **if-else** não é restrito a atribuições apenas. Lembre-se de que todas as funções (exceto aquelas declaradas como **void**) podem retornar um valor. Logo, você pode usar uma ou mais chamadas a funções em uma expressão em C. Quando o nome da função é encontrado, a função é executada e, então, seu valor de retorno pode ser determinado. Portanto, você pode executar uma ou mais chamadas a funções usando o operador **?**, colocando as chamadas nas expressões que formam os operandos, como em

```
#include <stdio.h>

int f1(int n);
int f2(void);

void main(void)
{
  int t;

  printf("Digite um número: ");
```

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

  /* imprime a mensagem apropriada */
  t ? f1(t) +  f2() : printf("foi digitado zero");
}

f1(int n)
{
  printf("%d ", n);
  return 0;
}

f2(void)
{
  printf("foi digitado");
  return 0;
}
```

Inserir um 0 nesse exemplo faz com que a função **printf()** seja chamada, mostrando a mensagem **foi digitado zero**. Se você inseriu qualquer outro número, **f1()** e **f2()** serão executadas. Note que o valor da expressão **?** é descartado nesse exemplo. Você não precisa atribuí-lo a nada.

Um aviso: alguns compiladores C rearranjam a ordem de avaliação de uma expressão numa tentativa de otimizar o código-objeto. Isso pode fazer com que as funções que formam os operandos do operador **?** sejam executadas em uma seqüência diferente da pretendida.

Usando o operador **?**, você pode reescrever o programa do número mágico mais uma vez.

```
/* Número mágico programa #5. */
#include <stdio.h>
#include <stdlib.h>

void main(void)
{
  int magic;
  int guess;

  magic = rand(); /* gera o número mágico */

  printf("Adivinhe o número mágico: ");
  scanf("%d", &guess);
```

```
    if(Guess == magic) {
      printf("** Certo **");
      printf("%d é o número mágico", magic);
    }
    else
      guess > magic ? printf("Alto") : printf("Baixo");
}
```

Aqui, o operador **?** mostra a mensagem apropriada baseado no resultado do teste **guess > magic**.

A Expressão Condicional

Algumas vezes, os iniciantes em C se confundem pelo fato de que você pode usar qualquer expressão válida em C para controlar o **if** ou o operador **?**. Isto é, você não fica restrito a expressões envolvendo os operadores relacionais e lógicos (como é o caso nas linguagens como BASIC ou Pascal). O programa precisa simplesmente chegar a um valor zero ou não-zero. Por exemplo, o seguinte programa lê dois inteiros do teclado e mostra o quociente. Ele usa um comando **if**, controlado pelo segundo número, para evitar um erro de divisão por zero.

```
/* Divide o primeiro número pelo segundo. */

#include <stdio.h>

void main(void)
{
  int a, b;

  printf("Digite dois números: ");
  scanf("%d%d", &a, &b);

  if(b) printf("%d\n", a/b);
  else printf("não pode dividir por zero\n");
}
```

Esse método funciona porque, se **b** é 0, a condição que controla **if** é falsa e o **else** é executado. De outra forma, a condição é verdadeira (não-zero) e a divisão é efetuada. Contudo, escrever o comando **if** desta forma:

```
if(b != 0) printf("%d\n", a/b);
```

é redundante, potencialmente ineficiente e considerado mau estilo.

switch

C tem um comando interno de seleção múltipla, **switch**, que testa sucessivamente o valor de uma expressão contra uma lista de constantes inteiras ou de caractere. Quando o valor coincide, os comandos associados àquela constante são executados. A forma geral do comando **switch** é

```
switch(expressão) {
case constante1:
        seqüência de comandos
        break;
case constante2:
        seqüência de comandos
        break;
case constante3:
        seqüência de comandos
        break;
.
.
.
default:
        seqüência de comandos
}
```

O valor da *expressão* é testado, na ordem, contra os valores das constantes especificadas nos comandos **case**. Quando uma coincidência for encontrada, a seqüência de comandos associada àquele **case** será executada até que o comando **break** ou o fim do comando **switch** seja alcançado. O comando **default** é executado se nenhuma coincidência for detectada. O **default** é opcional e, se não estiver presente, nenhuma ação será realizada se todos os testes falharem.

O padrão ANSI C especifica que um **switch** pode ter pelo menos 257 comandos **case**. Na prática, você desejará limitar o número de comandos **case** a uma quantidade menor, para obter mais eficiência. Embora **case** seja um rótulo, ele não pode existir sozinho, fora de um **switch**.

O comando **break** é um dos comandos de desvio em C. Você pode usá-lo em laços tal como no comando **switch** (veja a seção "Comandos de Iteração"). Quando um **break** é encontrado em um **switch**, a execução do programa "salta" para a linha de código seguinte ao comando **switch**.

Há três coisas importantes a saber sobre o comando **switch**:

- O comando **switch** difere do comando **if** porque **switch** só pode testar igualdade, enquanto **if** pode avaliar uma expressão lógica ou relacional.

- Duas constantes **case** no mesmo **switch** não podem ter valores idênticos. Obviamente, um comando **switch** incluído em outro switch mais externo pode ter as mesmas constantes **case**.
- Se constantes de caractere são usadas em um comando **switch**, elas são automaticamente convertidas para seus valores inteiros.

O comando **switch** é freqüentemente usado para processar uma entrada, via teclado, como em uma seleção por menu. Como mostrado aqui, a função **menu()** mostra o menu de um programa de validação da ortografia e chama os procedimentos apropriados:

```
void menu (void)
{
  char ch;

  printf("1. Checar Ortografia\n");
  printf("2. Corrigir Erros de Ortografia\n");
  printf("3. Mostrar Erros de Ortografia\n");
  printf("Pressione Qualquer Outra Tecla para Abandonar\n");
  printf("     Entre com sua escolha:   ");

  ch=getchar();  /* Lê do teclado a seleção */

  switch(ch) {
    case '1':
      check_spelling();
      break;
    case '2':
      correct_errors();
      break;
    case '3':
      display_errors();
      break;
    default :
      printf("Nenhuma opção selecionada");
    }
}
```

Tecnicamente, os comandos **break**, dentro do **switch**, são opcionais. Eles terminam a seqüência de comandos associados com cada constante. Se o comando **break** é omitido, a execução continua pelos próximos comandos **case** até que um **break**, ou o fim do **switch**, seja encontrado. Por exemplo, a seguinte função usa a natureza de "passar caindo" pelos **case**s para simplificar o código de um device-driver que manipula a entrada:

```
/* Processa um valor */
void inp_handler(int i)
{
  int flag;

  flag = -1;

  switch(i) {
    case 1:  /* Estes cases têm uma seqüência */
    case 2:  /* de comandos em comum */
    case 3:
      flag = 0;
      break;
    case 4:
      flag = 1;
    case 5:
      error(flag);
      break;
    default:
      process(i);
  }
}
```

Essa rotina ilustra dois aspectos do comando **switch**. Primeiro, você pode ter comandos case sem comandos associados. Quando isto ocorre, a execução simplesmente cai no **case** seguinte. Nesse caso, todos os três primeiros **case**s executam os mesmos comandos.

```
flag = 0;
break;
```

Segundo, a execução continua no próximo **case** se nenhum comando **break** estiver presente. Se **i** for igual a 4, **flag** receberá o valor 1 e, como não há nenhum comando **break** no fim desse **case**, a execução continuará e a função **error(flag)** será chamada. Se **i** fosse igual a 5, **error(flag)** teria sido chamada com o valor -1 em **flag**.

O fato de os **case**s poderem ser executados em conjunto quando nenhum **break** estiver presente evita a duplicação indesejável de código, resultando em um código muito eficiente.

Note que os comandos associados a cada **case** não são blocos de código mas, sim, *seqüências de comandos*. (Obviamente, o comando **switch** inteiro define um bloco.) Essa distinção técnica é importante apenas em certas situações. Por exemplo, o seguinte fragmento de código está errado e não será compilado por-

que você não pode declarar uma variável local em uma seqüência de comandos (você só pode declarar variáveis locais no início de um bloco.)

```
/* Isto está incorreto. */
switch(c) {
   case 1:
     int t;
     .
     .
     .
```

Embora um pouco estranho, você poderia adicionar uma variável local, como mostrado aqui. Neste caso, a declaração ocorre no início do bloco do **switch**.

```
/* Embora estranho, isto está correto. */
switch(c)
{
   int t;
   case 1:
     .
     .
     .
```

Obviamente, você pode criar um bloco de código como um dos comandos da seqüência e declarar uma variável local dentro dele, como exibido aqui:

```
/* Isto é correto. */
switch (c) {
  case 1:
    { /* Cria bloco */
       int t;
      .
      .
      .
    }
    .
    .
    .
```

Comandos switch Aninhados

Você pode ter um **switch** como parte de uma seqüência de comandos de um outro **switch**. Mesmo se as constantes dos **case**s dos **switch**s interno e externo possuírem valores comuns, não ocorrerão conflitos. Por exemplo, o seguinte fragmento de código é perfeitamente aceitável:

```
switch(x) {
   case 1:
     switch(y) {
       case 0: printf ("erro de divisão por zero");
               break;
       case 1: process(x,y);
     }
     break;
   case 2:
     .
     .
     .
```

Comandos de Iteração

Em C, e em todas as outras linguagens modernas de programação, comandos de iteração (também chamados laços) permitem que um conjunto de instruções seja executado até que ocorra uma certa condição. Essa condição pode ser predefinida (como no laço **for**) ou com o final em aberto (como nos laços **while** e **do-while**).

O Laço for

O formato geral do laço **for** de C é encontrado, de uma forma ou de outra, em todas as linguagens de programação baseadas em procedimentos. Contudo, em C, ele fornece flexibilidade e capacidade surpreendentes.

A forma geral do comando **for** é

for(*inicialização; condição; incremento*) *comando;*

O laço **for** permite muitas variações. Entretanto, a inicialização é, geralmente, um comando de atribuição que é usado para colocar um valor na variável de controle do laço. A *condição* é uma expressão relacional que determina quando o laço acaba. O *incremento* define como a variável de controle do laço varia cada

vez que o laço é repetido. Você deve separar essas três seções principais por pontos-e-vírgulas. Uma vez que a condição se torne falsa, a execução do programa continua no comando seguinte ao **for**.

Por exemplo, o seguinte programa imprime os números de 1 a 100 na tela:

```
#include <stdio.h>

void main(void)
{
   int x;

   for(x=1; x <= 100; x++) printf("%d ",x);
}
```

No programa, **x** é inicialmente ajustado para 1. Uma vez que **x** é menor que 100, **printf()** é executado e **x** é incrementado em 1 e testado para ver se ainda é menor ou igual a 100. Esse processo se repete até que **x** fique maior que 100; nesse ponto, o laço termina. Nesse exemplo, **x** é a variável de controle do laço, que é alterada e testada toda vez que o laço se repete.

O seguinte exemplo é um laço **for** que contém múltiplos comandos:

```
for(x=100; x != 65; x-=5) {
   z = x*x;
   printf("O quadrado de %d, %f",x, z);
}
```

Tanto a multiplicação de **x** por si mesmo como a função **printf()** são executadas até que **x** seja igual a 65. Note que o laço *é executado de forma inversa*: **x** é inicializado com 100 e será subtraído de 5 cada vez que o laço se repetir.

Nos laços **for**, o teste condicional sempre é executado no topo do laço. Isso significa que o código dentro do laço pode não ser executado se todas as condições forem falsas logo no início. Por exemplo, em

```
x = 10;
for(y=10; y!=x; ++y) printf("%d", y);
printf("%d", y); /* este é o único comando
                    printf() que executará */
```

o laço nunca executará, pois **x** e **y** já são iguais desde a sua entrada. Já que a expressão condicional é avaliada como falsa, nem o corpo do laço nem a porção de incremento do laço são executados. Logo, y ainda tem o valor 10 e a saída é o número 10 escrito apenas uma vez na tela.

Variações do Laço for

A discussão anterior descreveu a forma mais comum do laço **for**. Porém, C oferece diversas variações que aumentam a flexibilidade e a aplicação do laço **for**.

Uma das variações mais comuns usa o operador vírgula para permitir que duas ou mais variáveis controlem o laço. (Lembre-se de que você usa o operador vírgula para encadear expressões numa configuração "faça isto e isto". Veja o Capítulo 2.) Por exemplo, as variáveis **x** e **y** controlam o seguinte laço e ambas são inicializadas dentro do comando **for**:

```
for(x=0, y=0; x+y<10; ++x) {
  y = getchar();
  y = y-'0'; /* subtrai o código ASCII do caractere 0 de y */
     .
     .
     .
}
```

Vírgulas separam os dois comandos de inicialização. Cada vez que **x** é incrementado, o laço repete e o valor de **y** é ajustado pela entrada do teclado. Tanto **x** como **y** devem ter o valor correto para que o laço termine. Embora os valores de **y** sejam definidos pela entrada do teclado, **y** deve ser inicializado com 0 de forma que seu valor seja definido antes da avaliação da expressão condicional. (Se **y** não fosse definido, ele poderia conter um 10, tornando o teste condicional falso e impedindo a execução do laço.)

A função **converge()**, mostrada a seguir, ilustra múltiplas variáveis de controle de laço. A função **converge()** expõe uma string escrevendo os caracteres vindos de ambas as extremidades, convergindo no meio da linha especificada. Isso requer um posicionamento do cursor em vários e desconexos pontos da tela. Uma vez que C roda sob uma ampla variedade de ambientes, ele não define uma função de posicionamento do cursor. Porém, virtualmente, todo compilador C fornece uma, embora seu nome possa variar. O programa a seguir usa a função **gotoxy()**, do Borland, para posicionar o cursor. (Ela exige o cabeçalho **conio.h**.)

```
/* Versão Borland. */
#include <stdio.h>
#include <conio.h> /* arquivo de cabeçalho não-padrão */
#include <string.h>

void converge(int line, char *message);
```

```
void main (void)
{
  converge(10, "Isto é um teste de converge().");
}

/* Essa função mostra uma string iniciando do lado esquerdo da
   linha especificada. Ela escreve caracteres de ambas as
   extremidades, convergindo para o centro. */
void converge(int line, char *message)
{
  int i, j;

  for(i=1, j=strlen(message); i<j; i++, j--) {
    gotoxy(i, line); printf("%c", message[i-1]);
    gotoxy(j, line); printf("%c", message[j-1]);
  }
}
```

No Microsoft C, o equivalente a **gotoxy()** é **_settextposition()**, que usa o arquivo de cabeçalho **graph.h**. O programa anterior, recodificado para o Microsoft C, é mostrado aqui:

```
/* Versão para Microsoft. C */
#include <stdio.h>
#include <graph.h> /* arquivo de cabeçalho não-padrão */
#include <string.h>

void converge(int line, char *message);

void main(void)
{
  converge(10, "Isto é um teste de converge().");
}

/* Essa função mostra uma string iniciando do lado esquerdo
   da linha especificada. Ela escreve caracteres de ambas as
   extremidades, convergindo para o centro.  */
void converge(int line, char *message)
{
  int i, j;

  for(i=1, j=strlen(message); i<j; i++, j--) {
    _settextposition(line, i);
    printf("%c", message[i-1]);
```

```
    _settextposition(line, j);
    printf("%c", message[j-1]);
  }
}
```

Se você usa um compilador C diferente, será necessária uma verificação nos seus manuais do usuário para saber o nome da função de posicionamento do cursor.

Nas duas versões de **converge()**, o laço **for** usa duas variáveis de controle do laço, **i** e **j**, para indexar a string pelas extremidades. Quando o laço repete, **i** aumenta e **j** diminui. O laço parará quando **i** for igual a **j**, garantindo, assim, que todos os caracteres sejam escritos.

A expressão condicional não precisa envolver um teste com a variável que controla o laço e algum valor final. Na realidade, a condição pode ser qualquer sentença relacional ou lógica. Isso significa que você pode testar diversas possíveis formas de término.

Por exemplo, você poderia usar a seguinte função para conectar um usuário a um sistema remoto. O usuário tem três tentativas para entrar com a senha. O laço termina quando as três tentativas forem utilizadas ou o usuário entrar com a senha correta.

```
void sign_on(void)
{
  char str[20] ;

  int x;

  for(x=0; x<3 && strcmp(str, "senha"); ++x) {
    printf("Digite a senha por favor:");
    gets(str);
  }

  if (x==3) return;
  /* else conecte o usuário ... */
}
```

Essa função usa **strcmp()**, a função da biblioteca padrão que compara duas strings e retorna zero se forem iguais.

Lembre-se de que cada uma das três seções pode consistir em qualquer expressão válida em C. As expressões não precisam ter relação alguma com os usos mais comuns das seções. Com isso em mente, considere o seguinte exemplo:

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

```
int sqrnum(int num);
int readnum(void);
int prompt(void);

void main(void)
{
  int t;

  for(prompt(); t=readnum(); prompt())
    sqrnum(t);
}

prompt(void)
{
  printf("Digite um número: ");
  return 0;
}

readnum(void)
{
  int t;
  scanf("%d", &t);
  return t;
}

sqrnum(int num)
{
  printf("%d\n", num*num);
  return num*num;
}
```

Olhe atentamente o laço **for** em **main()**. Note que cada parte do laço **for** é composta de chamadas a funções que interagem com o usuário e lêem um número inserido pelo teclado. Se o número inserido for 0, o laço terminará, porque a expressão condicional será falsa. Caso contrário, o número é elevado ao quadrado. Assim, esse laço **for** usa as porções de inicialização e incremento de uma maneira não tradicional, mas completamente válida.

Uma outra característica interessante do laço **for** é que partes da definição do laço não precisam existir. De fato, não há necessidade de que uma expressão esteja presente em nenhuma das seções — as expressões são opcionais. Por exemplo, esse laço será executado até que o usuário insira o número **123**:

```
for(x=0; x!=123; ) scanf("%d", &x);
```

Note que a porção de incremento da definição do **for** está vazia. Isso significa que cada vez que o laço se repetir, **x** será testado para verificar se é igual a 123, mas nenhuma atitude adicional será tomada. Se você digitar **123** no teclado, porém, a condição do laço se tornará falsa e o laço terminará.

Freqüentemente a inicialização é feita fora do comando **for**. Isso geralmente acontece quando a condição inicial da variável de controle do laço for calculada por algum método complexo, como neste exemplo:

```
gets(s);  /* lê uma string para s */
if(*s) x = strlen(s); /* obtém o comprimento da string */
else x = 10;

for( ;x<10; ) {
  printf("%d",x);
  ++x;
}
```

A seção de inicialização foi deixada em branco e **x** é inicializado antes que o laço comece a ser executado.

O Laço Infinito

Embora você possa usar qualquer comando de laço para criar um laço infinito, o **for** é tradicionalmente usado para esse fim. Já que nenhuma das três expressões que formam o laço **for** é obrigatória, você pode fazer um laço sem fim deixando a expressão condicional vazia, como aqui:

```
for (;;) printf("Esse laço será executado para sempre.\n");
```

Você pode ter uma inicialização e uma expressão de incremento, mas programadores C usam mais usualmente a construção **for(;;)** para significar um laço infinito.

De fato, a construção **for(;;)** não garante um laço infinito porque o comando **break** de C, encontrado em qualquer lugar dentro do corpo de um laço, provoca um término imediato (**break** será discutido mais adiante neste capítulo). O controle do programa, então, continua no código seguinte ao laço, como mostrado aqui:

```
ch='\0';

for( ; ; ) {
  ch = getchar();  /* obtém um caractere */
```

```
  if(ch=='A') break; /* sai do laço */
}

printf("você digitou um A");
```

Esse laço será executado até que o usuário digite um A.

Laços for sem Corpos

Como definido pela sintaxe de C, um comando pode ser vazio. Isso significa que o corpo do laço **for** (ou qualquer outro laço) também pode ser vazio. Você pode usar esse fato para aumentar a eficiência de certos algoritmos e para criar laços para atraso de tempo.

Remover espaços de uma "stream" de entrada é uma tarefa comum de programação. Por exemplo, um programa de banco de dados pode permitir uma requisição do tipo "mostre todos os saldos menores que 400". O banco de dados precisa ser alimentado com cada palavra separadamente, sem espaços. Isto é, o processador de entrada do banco de dados reconhece "**show**" mas não "**show**". O seguinte laço remove os primeiros espaços da stream apontada por **str**.

```
for( ; *str == ' '; str++) ;
```

Como você pode ver, esse laço não tem corpo — e nem precisa de um.

Os laços para *atraso de tempo* são muito usados em programas. O seguinte código mostra como criar um usando **for**:

```
for(t=0; t<ALGUM_VALOR; t++) ;
```

O Laço while

O segundo laço disponível em C é o laço **while**. A sua forma geral é

while(*condição*) *comando*;

onde *comando* é um comando vazio, um comando simples ou um bloco de comandos. A *condição* pode ser qualquer expressão, e verdadeiro é qualquer valor não-zero. O laço se repete quando a condição for verdadeira. Quando a condição for falsa, o controle do programa passa para a linha após o código do laço.

O seguinte exemplo mostra uma rotina de entrada pelo teclado, que simplesmente se repete até que o usuário digite **A**:

```
wait_for_char(void)
```

```
{
  char ch;

  ch = '\0';  /* inicializa ch */
  while(ch != 'A')  ch = getchar();
  return ch;
}
```

Primeiro, **ch** é inicializado com nulo. Como uma variável local, seu valor não é conhecido quando **wait_for_char()** é executado. O laço **while** verifica se **ch** não é igual a **A**. Como **ch** foi inicializado com nulo, o teste é verdadeiro e o laço começa. Cada vez que o usuário pressiona uma tecla, o teste é executado novamente. Uma vez digitado **A**, a condição se torna falsa, porque **ch** fica igual a **A**, e o laço termina.

Como os laços **for**, os laços **while** verificam a condição de teste no início do laço, o que significa que o código do laço pode não ser executado. Isso elimina a necessidade de se efetuar um teste condicional antes do laço. A função **pad()** fornece uma boa ilustração disso. Ela adiciona espaços ao final de uma string até um comprimento predefinido. Se a string já é do tamanho desejado, nenhum espaço é adicionado.

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

void pad(char *s, int length);
void main(void)
{
  char str[80];

  strcpy(str, "isto é um teste");
  pad(str, 40);
  printf("%d", strlen(str));
}

/* Acrescenta espaços ao final da string. */
void pad(char *s, int length)
{
  int l;

  l = strlen(s);  /* encontra o comprimento */

  while(l<length) {
    s[l] = ' ';  /* insere um espaço */
```

```
    l++;
  }
  s[l] = '\0';  /* strings precisam terminar com um nulo */
}
```

Os dois argumentos de **pad()** são **s**, um ponteiro para a string a ser ajustada, e **length**, o número de caracteres que **s** deve ter. Se a string **s** já é igual ou maior que **length**, o código dentro do laço **while** não será executado. Se **s** é menor que **length**, **pad()** adiciona o número necessário de espaços. A função **strlen()**, parte integrante da biblioteca padrão, devolve o tamanho de uma string.

Se diversas condições forem necessárias para terminar um laço **while**, normalmente uma única variável forma a expressão condicional. O valor dessa variável será alterado em vários pontos internos ao laço. Neste exemplo

```
void func1(void)
{
  int working;

  working = 1;  /* i.e., verdadeiro */

  while(working) {
    working = process1();
    if(working)
      working = process2();
    if(working)
      working = process3();
  }
}
```

qualquer uma das três rotinas pode retornar falso e fazer com que o laço termine.

Não é necessário haver nenhum comando no corpo do laço **while**. Por exemplo,

```
while((ch=getchar()) != 'A') ;
```

será simplesmente executado até que o usuário digite **A**. Se você se sente desconfortável colocando a atribuição dentro da expressão condicional do **while**, lembre-se de que o sinal de igual é apenas um operador que calcula o valor do operando do lado direito.

O Laço do-while

Ao contrário dos laços **for** e **while**, que testam a condição do laço no começo, o laço **do-while** verifica a condição ao final do laço. Isso significa que um laço **do-while** sempre será executado ao menos uma vez. A forma geral do laço **do-while** é

```
do{
    comando;
} while(condição);
```

Embora as chaves não sejam necessárias quando apenas um comando está presente, elas são geralmente usadas para evitar confusão (para você, não para o compilador) com o **while**. O laço **do-while** repete até que a *condição* se torne falsa.

O seguinte laço **do-while** lerá números do teclado até que encontre um número menor ou igual a 100.

```
do {
  scanf("%d", &num);
} while(num > 100);
```

Talvez o uso mais comum do laço **do-while** seja em uma rotina de seleção por menu. Quando o usuário entra com uma resposta válida, ela é retornada como o valor da função. Respostas inválidas provocam uma repetição do laço. O seguinte código mostra uma versão melhorada do menu do verificador de ortografia que foi desenvolvido anteriormente neste capítulo:

```
void menu(void)
{
  char ch;

  printf("1. Verificar Ortografia\n");
  printf("2. Corrigir Erros de Ortografia\n");
  printf("3. Mostrar Erros de Ortografia\n");
  printf("     Digite sua escolha:   ");

  do {
    ch = getchar();  /* lê do teclado a seleção */
    switch(ch) {
      case '1':
        check_spelling();
        break;
```

```
        case '2':
          correct_errors();
          break;
        case '3':
          display_errors();
          break;
      }
    } while(ch!='1' && ch!='2' && ch!='3');
}
```

Aqui, o laço **do-while** é uma boa escolha, porque você sempre deseja que uma função do menu execute ao menos uma vez. Depois que as opções forem mostradas, o programa será executado até que uma opção válida seja selecionada.

Comandos de Desvio

C tem quatro comandos que realizam um desvio incondicional: **return**, **goto**, **break** e **continue**. Destes, você pode usar **return** e **goto** em qualquer lugar em seu programa. Você pode usar os comandos **break** e **continue** em conjunto com qualquer dos comandos de laço. Como discutido anteriormente neste capítulo, você também pode usar o **break** com **switch**.

O Comando return

O comando **return** é usado para retornar de uma função. Ele é um comando de desvio porque faz com que a execução retorne (salte de volta) ao ponto em que a chamada à função foi feita. Se **return** tem um valor associado a ele, esse valor é o valor de retorno da função. Se nenhum valor de retorno é especificado, assume-se que apenas lixo é retornado. (Alguns compiladores C irão automaticamente retornar 0 se nenhum valor for especificado, mas não conte com isso.)

A forma geral do comando **return** é

return *expressão*;

Lembre-se de que a *expressão* é opcional. Entretanto, se estiver presente, ela se tornará o valor da função.

Você pode usar quantos comandos **return** quiser dentro de uma função. Contudo, a função parará de executar tão logo ela encontre o primeiro **return**. A } que finaliza uma função também faz com que a função retorne. É o mesmo que um **return** sem nenhum valor especificado.

Uma função declarada como **void** não pode ter um comando **return** que especifique um valor. (Como uma função void não retorna valor, não tem sentido que o comando **return** especifique um valor nas funções **void**).

Veja o Capítulo 6 para mais informações sobre **return**.

O Comando goto

Uma vez que C tem um rico conjunto de estruturas de controle e permite um controle adicional usando **break** e **continue**, há pouca necessidade do **goto**. A grande preocupação da maioria dos programadores sobre o **goto** é a sua tendência de tornar os programas ilegíveis. Entretanto, embora o **goto** tenha sido desencorajado alguns anos atrás, ele tem recentemente polido um pouco a sua imagem manchada. Não há nenhuma situação na programação que necessite do **goto**. Apesar disso, contudo, **goto** é uma conveniência que, se usada prudentemente, pode ser uma vantagem em certas situações na programação. Sendo assim, **goto** não é usado fora desta seção.

O comando **goto** requer um rótulo para sua operação. (Um rótulo é um identificador válido em C seguido por dois-pontos.) O padrão ANSI refere-se a esse tipo de construção como uma sentença de rótulo. Além disso, o rótulo tem de estar na mesma função do **goto** que o usa — você não pode efetuar desvios entre funções. A forma geral do comando **goto** é

goto *rótulo*;
.
.
.
rótulo:

onde *rótulo* é qualquer rótulo válido existente antes ou depois do **goto**. Por exemplo, você poderia criar um laço de 1 até 100 usando **goto** e um rótulo, como mostrado aqui:

```
x = 1;
loop1:
  x++;
  if(x<100) goto loop1;
```

O Comando break

O comando **break** tem dois usos. Você pode usá-lo para terminar um **case** em um comando **switch** (abordado anteriormente na seção sobre o **switch**, neste

capítulo). Você também pode usá-lo para forçar uma terminação imediata de um laço, evitando o teste condicional normal do laço.

Quando o comando **break** é encontrado dentro de um laço, o laço é imediatamente terminado e o controle do programa retorna no comando seguinte ao laço. Por exemplo,

```
#include <stdio.h>

void main(void)
{
  int t;

  for(t=0; t<100; t++) {
    printf("%d ", t);
    if(t==10) break;
  }
}
```

escreve os números de 1 até 10 na tela. Então, o laço termina porque o **break** provoca uma saída imediata do laço, desrespeitando o teste condicional **t<100**.

Os programadores geralmente usam o comando **break** em laços em que uma condição especial pode provocar uma terminação imediata. Por exemplo, aqui, o pressionamento de uma tecla pode parar a execução da função **look_up()**:

```
look_up(char *name)
{
  do {
    /* procura nomes ... */
    if(kbhit()) break;
  } while(!found);
  /* processa a concordância */
}
```

A função **kbhit()** retorna 0 se você não pressionar uma tecla. Caso contrário, ela retorna um valor não-zero. Devido às grandes diferenças entre ambientes computacionais, o padrão ANSI não define **kbhit()**, mas é quase certo que você a tem (ou alguma com um nome um pouco diferente) fornecida com seu compilador.

Um comando **break** provoca uma saída apenas do laço mais interno. Por exemplo,

```
for(t=0; t<100; ++t) {
  count = 1;
  for(;;) {
    printf("%d ", count);
    count++;
    if(count==10) break;
  }
}
```

escreve os números de 1 a 10 na tela 100 vezes. Cada vez que o compilador encontra **break**, transfere o controle de volta para a repetição **for** mais externa.

Um **break** usado em um comando **switch** somente afetará esse **switch**. Ele não afeta qualquer repetição que contenha o **switch**.

A Função exit()

Da mesma forma que você pode sair de um laço, pode sair de um programa usando a função **exit()** da biblioteca padrão. Essa função provoca uma terminação imediata do programa inteiro, forçando um retorno ao sistema operacional. Com efeito, a função **exit()** age como se ela estivesse finalizando o programa inteiro.

A forma geral da função **exit()** é

void exit(int *código_de_retorno*);

O valor de *código_de_retorno* é retornado ao processo chamador, que é normalmente o sistema operacional. O zero é geralmente usado como um código de retorno que indica uma terminação normal do programa. Outros argumentos são usados para indicar algum tipo de erro.

Programadores geralmente usam **exit()** quando uma condição mandatória para a execução do programa não é satisfeita. Por exemplo, imagine um jogo para computador que queira uma placa gráfica colorida. A função **main()** desse jogo poderia se parecer com isto:

```
void main(void)
{
  if(!virtual_graphics()) exit(1);
  play();
}
```

onde **virtual_graphics()** é uma função definida pelo usuário que retorna verdadeiro se a placa colorida está presente. Se a placa não está no sistema, **virtual_graphics()** retorna falso e o programa termina.

Como um outro exemplo, essa versão de **menu()** usa **exit()** para abandonar o programa e retornar ao sistema operacional:

```
void menu(void)
{
  char ch;

  printf("1. Verificar Ortografia\n");
  printf("2. Corrigir Erros de Ortografia\n");
  printf("3. Mostrar Erros de Ortografia\n");
  printf("4. Abandonar\n");
  printf("     Digite sua escolha:   ");

  do {
    ch=getchar();  /* lê do teclado a seleção */
      switch(ch) {
         case '1':
           check_spelling();
           break;
         case '2':
           correct_errors();
           break;
         case '3':
           display_errors();
           break;
         case '4':
           exit(0);  /* retorna ao OS */
    }
  } while(ch!='1' && ch!='2' && ch!='3');
}
```

O Comando continue

O comando **continue** trabalha de uma forma um pouco parecida com a do comando **break**. Em vez de forçar a terminação, porém, **continue** força que ocorra a próxima iteração do laço, pulando qualquer código intermediário. Para o laço **for**, **continue** faz com que o teste condicional e a porção de incremento do laço sejam executados. Para os laços **while** e **do-while**, o controle do programa passa para o teste condicional. Por exemplo, o seguinte programa conta o número de espaços contidos em uma string inserida pelo usuário:

```
/* Conta espaços */
#include <stdio.h>
```

```
void main(void)
{
  char s[80], *str;
  int space;

  printf("Digite uma string: ");
  gets(s);
  str = s;

  for(space=0; *str; str++) {
    if(*str !=' ') continue;
    space++;
  }
  printf("%d espaços\n", space);
}
```

Cada caractere é testado para ver se é um espaço. Se não é, o comando **continue** força o **for** a iterar novamente. Se o caractere é um espaço, **space** é incrementada.

O seguinte exemplo mostra que você pode usar **continue** para apressar a saída de um laço, forçando o teste condicional a ser realizado mais cedo.

```
void code(void)
{
  char done, ch;

  done = 0;
  while(!done) {
    ch = getchar();
    if(ch=='$') {
      done = 1;
      continue;
    }
    putchar(ch+1);  /* desloca o alfabeto uma posição */
  }
}
```

Esta função codifica uma mensagem deslocando todos os caracteres uma letra acima. Por exemplo, um **A** se tornaria um **B**. A função terminará quando um **$** for digitado. Depois que um **$** for inserido, nenhuma saída adicional ocorrerá porque o teste condicional, trazido a efeito pelo **continue**, encontrará **done** como sendo verdadeiro e provocará a saída do laço.

Comandos de Expressões

O Capítulo 2 abrange completamente as expressões de C. Porém, alguns pontos especiais são mencionados aqui. Lembre-se de que um comando de expressão é simplesmente uma expressão válida em C seguida por um ponto-e-vírgula, como em

```
func();  /* uma chamada a uma função */
a = b+c; /* um comando de atribuição */
b+f();   /* um comando válido, que não faz nada */
;        /* um comando vazio */
```

O primeiro comando de expressão executa uma chamada a uma função. O segundo é uma atribuição. O terceiro elemento é estranho, mas é avaliado pelo compilador C, porque a função **f()** pode realizar alguma tarefa necessária. O último exemplo mostra que C permite que um comando seja vazio (algumas vezes chamado de *comando nulo*).

Blocos de Comandos

Blocos de comandos são simplesmente grupos de comandos relacionados que são tratados como uma unidade. Os comandos que constituem um bloco estão logicamente conectados. Um bloco começa com um { e termina com um } correspondente. Programadores normalmente usam blocos de comandos para criar um objeto multicomandos para algum outro comando, como o **if**. No entanto, você pode pôr um bloco de comandos em qualquer lugar onde seja possível a colocação de um outro comando qualquer. Por exemplo, este é um código em C perfeitamente válido (embora não usual):

```
#include <stdio.h>

void main(void)
{
  int i;

  {  /* um bloco de comandos */
     i = 120;
     printf("%d", i);
  }
}
```