MAKRON
Books

7

Estruturas, Uniões, Enumerações e Tipos Definidos pelo Usuário

A linguagem C permite criar tipos de dados definíveis pelo usuário de cinco formas diferentes. O primeiro é a *estrutura*, que é um agrupamento de variáveis sob um nome e é chamado tipo de dado *agregado* (ou, às vezes, *conglomerado*). O segundo tipo definido pelo usuário é o *campo de bit*, que é uma variação da estrutura que permite o fácil acesso aos bits dentro de uma palavra. O terceiro é a *união*, que permite que a mesma porção da memória seja definida por dois ou mais tipos diferentes de variáveis. Um quarto tipo de dado definível pelo usuário é a *enumeração*, que é uma lista de símbolos. O último tipo definido pelo usuário é criado através do uso de **typedef** e define um novo nome para um tipo existente.

Estruturas

Em C, uma estrutura é uma coleção de variáveis referenciadas por um nome, fornecendo uma maneira conveniente de se ter informações relacionadas agrupadas. Uma *definição de estrutura* forma um modelo que pode ser usado para criar variáveis de estruturas. As variáveis que compreendem a estrutura são chamadas membros da estrutura. (Os membros da estrutura são comumente chamados *elementos* ou *campos*.)

Geralmente, todos os elementos na estrutura são logicamente relacionados. Por exemplo, a informação de nome e endereço em uma lista postal seria normalmente representada em uma estrutura. O fragmento de código seguinte mostra como criar um modelo de estrutura que define os campos de nome e

endereço. A palavra-chave **struct** informa ao compilador que um modelo de estrutura está sendo definido.

```
struct addr
{
  char name[30];
  char street[40];
  char city[20];
  char state[3];
  unsigned long int zip;
};
```

Observe que a definição termina com um ponto-e-vírgula. Isso ocorre porque uma definição de estrutura é um comando. Além disso, o identificador (tag) da estrutura **addr** indica essa estrutura de dados particular e é o seu especificador de tipo.

Nesse ponto do código, *nenhuma variável foi de fato declarada.* Apenas a forma dos dados foi definida. Para declarar uma variável de tipo **addr**, escreva

```
struct addr addr_info;
```

Isso declara uma variável do tipo **struct addr** chamada **addr_info**. Quando você define uma estrutura, está essencialmente definindo um tipo complexo de variável, não uma variável. Não existe uma variável desse tipo até que ela seja realmente declarada.

Quando uma variável de estrutura (como **addr_info** é declarada, o compilador C aloca automaticamente memória suficiente para acomodar todos os seus membros. A Figura 7.1 mostra como **addr_info** aparece na memória, assumindo caracteres de 1 byte e inteiros longos de 4 bytes.

Você também pode declarar uma ou mais variáveis ao definir a estrutura. Por exemplo,

```
struct addr {
  char name[30];
  char street[40];
  char city[20];
  char state[3];
  unsigned long int zip;
} addr_info, binfo, cinfo;
```

define uma estrutura chamada **addr** e declara as variáveis **addr_info**, **binfo** e **cinfo** desse tipo.

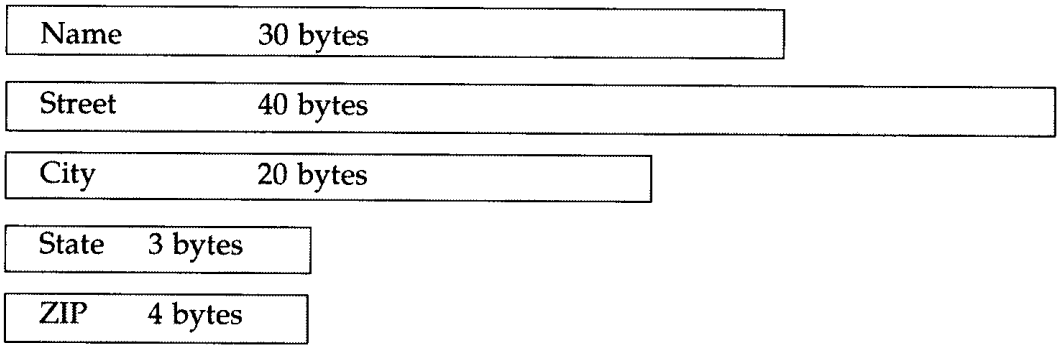

Figura 7.1 A estrutura **addr_info** na memória.

Se você precisa de apenas uma variável estrutura, o nome da estrutura não é necessário. Isso significa que

```
struct {
  char name[30];
  char street[40];
  char city[20];
  char state[3];
  unsigned long int zip;
} addr_info;
```

declara uma variável chamada **addr_info** como definido pela estrutura que a precede.

A forma geral de uma definição de estrutura é

struct *identificador* {
  tipo nome_da_variável;
  tipo nome_da_variável;
  tipo nome_da_variável;
    .
    .
    .
} *variáveis_estrutura;*

onde *identificador* ou *variáveis_estrutura* podem ser omitidos, mas não ambos.

Referenciando Elementos de Estruturas

Elementos individuais de estruturas são referenciados por meio do operador (algumas vezes chamado de *operador ponto*). Por exemplo, o código seguinte atribui o CEP 12345 ao campo **zip** da variável estrutura **addr_info** declarada anteriormente:

```
addr_info.zip = 12345;
```

O nome da variável estrutura seguido por um ponto e pelo nome do elemento referencia esse elemento individual da estrutura. A forma geral para acessar um elemento de estrutura é

nome_da_estrutura.nome_do_elemento

Assim, para imprimir o CEP na tela, escreva

```
printf("%d", addr_info.zip);
```

Isso imprime o código do CEP contido na variável **zip** da variável estrutura **addr_info**.

Do mesmo modo, a matriz de caracteres **addr_info.name** pode ser usada para chamar **gets()**, como mostrado aqui:

```
gets(addr_info.name);
```

Isso passa um ponteiro de caracteres para o início do elemento **name**.

Para acessar os elementos individuais de **addr_info.name**, você pode indexar **name**. Por exemplo, você pode imprimir o conteúdo de **addr_info.name**, um caractere por vez, usando o seguinte código:

```
register int t;

for(t=0; addr_info.name[t]; ++t)

  putchar(addr_info.name[t]);
```

Atribuição de Estruturas

Se seu compilador C é compatível com o padrão C ANSI, a informação contida em uma estrutura pode ser atribuída a outra estrutura do mesmo tipo. Isto é, em lugar de ter de atribuir os valores de todos os elementos separadamente, você pode empregar um único comando de atribuição. O programa seguinte ilustra atribuições de estruturas:

```
#include <stdio.h>

void main(void)
{
```

```
  struct {
    int a;
    int b;
  } x, y;

  x.a = 10;

  y = x; /* atribui uma estrutura a outra */

  printf("%d", y.a);
}
```

Após a atribuição, **y.a** conterá o valor 10.

Matrizes de Estruturas

Talvez o uso mais comum de estruturas seja em matriz de estruturas. Para declarar uma matriz de estruturas, você deve primeiro definir uma estrutura e, então, declarar uma variável matriz desse tipo. Por exemplo, para declarar uma matriz de estruturas com 100 elementos do tipo **addr**, que foi definido anteriormente, deve-se escrever

```
struct addr addr_info[100];
```

Isso cria 100 conjuntos de variáveis que estão organizados como definido na estrutura **addr**.

Para acessar uma estrutura específica, deve-se indexar o nome da estrutura. Por exemplo, para imprimir o código do CEP da estrutura 3, escreva

```
printf("%d", addr_info[2].zip);
```

Como todas as outras matrizes, matrizes de estruturas começam a indexação em 0.

Um Exemplo de Lista Postal

Para ilustrar como estruturas e matrizes de estruturas são usadas, esta seção desenvolve um programa simples de lista postal que usa uma estrutura para guardar as informações de endereço. Nesse exemplo, a informação armazenada inclui nome, rua, cidade, estado e CEP.

Para definir a estrutura básica de dados, **addr**, que contém essa informação, escreva

```
struct addr {
  char name[30];
  char street[40];
  char city[20];
  char state[3];
  unsigned long int zip;
} addr_info[MAX];
```

Observe que o campo de CEP é um inteiro longo sem sinal. Isso ocorre porque os CEPs maiores que 64000 — como 94564 — não podem ser representados em um inteiro de 2 bytes. Nesse exemplo, um inteiro possui o código do CEP para ilustrar um elemento de estrutura numérico. Porém, a prática mais comum é usar uma string de caracteres para acomodar códigos postais com letras além de números (como usado no Canadá e em outros países). O valor de **MAX** pode ser definido para satisfazer necessidades específicas.

A primeira função necessária para o programa é **main()**.

```
/* Um exemplo simples de lista postal usando uma
   matriz de estruturas. */
#include <stdio.h>
#include <stdlib.h>

#define MAX 100

struct addr {
  char name[30];
  char street[40];
  char city[20];
  char state[3];
  unsigned long int zip;
} addr_info[MAX];

void init_list(void), enter(void);
void delete(void), list(void);
int menu_select(void), find_free(void);

void main(void)
{
  char choice;

  init_list(); /* inicializa a matriz de estruturas */

  for(;;) {
```

```
    choice=menu_select();
    switch(choice) {
      case 1: enter();
        break;
      case 2: delete();
        break;
      case 3: list();
        break;
      case 4: exit(0);
    }
  }
}
```

Primeiro, a função **init_list()** prepara a matriz de estruturas para ser usada, colocando um caractere nulo no primeiro byte do campo **nome**. O programa assume que uma variável estrutura não está sendo usada se **nome** estiver vazio. A função **init_list()** é mostrada aqui:

```
/* Inicializa a lista. */
void init_list(void)
{
  register int t;

  for(t=0; t<MAX; ++t) addr_info[t].name[0] = '\0';
}
```

A função **menu_select()** apresenta as mensagens de opção e devolve a seleção do usuário.

```
/* Obtém a seleção. */
menu_select(void)
{
  char s[80];
  int c;

  printf("1. Inserir um nome\n");
  printf("2. Excluir um nome\n");
  printf("3. Listar o arquivo\n");
  printf("4. Sair\n");

  do {
    printf("\nDigite sua escolha: ");
    gets(s);
    c = atoi(s);
```

```
  } while(c<0 || c>4);
  return c;
}
```

A função **enter()** espera pela entrada do usuário e coloca a informação recebida na próxima estrutura livre. Se a matriz estiver cheia, então a mensagem **lista cheia** será escrita na tela. A função **find_free()** procura um elemento não usado na matriz de estruturas.

```
/* Insere os endereços na lista. */
void enter(void)
{
  int slot;
  char s[80];

  slot = find_free();
  if(slot==-1) {
    printf("\nLista cheia");
    return;
  }

  printf("Digite o nome: ");
  gets(addr_info[slot].name);

  printf("Digite a rua: ");
  gets(addr_info[slot].street);

  printf("Digite a cidade: ");
  gets(addr_info[slot].city);

  printf("Digite o estado: ");
  gets(addr_info[slot].state);

  printf("Digite o cep: ");
  gets(s);
  addr_info[slot].zip = strtoul(s, '\0', 10);
}

/* Encontra uma estrutura não usada. */
find_free(void)
{
  register int t;
  for(t=0; addr_info[t].name[0] && t<MAX; ++t);

  if(t==MAX) return -1; /* nenhum elemento livre */
```

```
  return t;
}
```

Observe que **find_free()** devolve -1 se toda a matriz de estruturas está sendo usada. Esse é um número seguro porque não pode haver um elemento -1 em uma matriz.

A função **delete()** simplesmente pede ao usuário para especificar o número do endereço que precisa ser excluído. A função, então, põe um caractere nulo no primeiro caractere do campo **name**.

```
/* Apaga um endereço. */
void delete(void)
{
  register int slot;
  char s[80];

  printf("Digite o registro #: ");
  gets(s);
  slot = atoi(s);
  if(slot>=0 && slot < MAX)
       addr_info[slot].name[0] = '\0';
}
```

A última função de que o programa precisa é **list()**, que escreve a lista postal inteira na tela. O padrão C não define uma função que envie a saída para a impressora, em virtude da grande variação entre ambientes. Porém, todos os compiladores C fornecem alguns significados para executar isto. No entanto, você pode querer acrescentar essa capacidade ao programa de lista postal por sua conta.

```
/* Mostra a lista na tela. */
void list(void)
{
  register int t;

  for(t=0; t<MAX; ++t) {
    if(addr_info[t].name[0]) {
      printf("%s\n", addr_info[t].name);
      printf("%s\n", addr_info[t].street);
      printf("%s\n", addr_info[t].city);
      printf("%s\n", addr_info[t].state);
      printf("%lu\n", addr_info[t].zip);
    }
  }
```

```
  printf("\n\n");
}
```

O programa completo de lista postal é mostrado aqui. Se você ainda tem qualquer dúvida sobre estruturas, digite esse programa em seu computador e estude a sua execução, fazendo alterações e observando seus efeitos.

```
/* Um exemplo simples de lista postal usando uma
   matriz de estruturas. */
#include <stdio.h>
#include <stdlib.h>

#define MAX 100

struct addr {
  char name[30];
  char street[40];
  char city[20];
  char state[3];
  unsigned long int zip;
} addr_info[MAX];

void init_list(void), enter(void);
void delete(void), list(void);
int menu_select(void), find_free(void);

void main(void)
{
  char choice;

  init_list(); /* inicializa a matriz de estruturas */
  for(;;) {
    choice=menu_select();
    switch(choice) {
      case 1: enter();
        break;
      case 2: delete();
        break;
      case 3: list();
        break;
      case 4: exit(0);
    }
  }
}
```

```
/* Inicializa a lista. */
void init_list(void)
{
  register int t;

  for(t=0; t<MAX; ++t) addr_info[t].name[0] = '\0';
}

/* Obtém a seleção. */
menu_select(void)
{
  char s[80];
  int c;

  printf("1. Inserir um nome\n");
  printf("2. Excluir um nome\n");
  printf("3. Listar o arquivo\n");
  printf("4. Sair\n");
  do {
    printf("\nDigite sua escolha: ");
    gets(s);
    c = atoi(s);
  } while(c<0 || c>4);
  return c;
}

/* Insere os endereços na lista. */
void enter(void)
{
  int slot;
  char s[80];

  slot = find_free();
  if(slot==-1) {
    printf("\nlista cheia");
    return;
  }

  printf("Digite o nome: ");
  gets(addr_info[slot].name);

  printf("Digite a rua: ");
  gets(addr_info[slot].street);
```

```
  printf("Digite a cidade: ");
  gets(addr_info[slot].city);

  printf("Digite o estado: ");
  gets(addr_info[slot].state);

  printf("Digite o cep: ");
  gets(s);
  addr_info[slot].zip = strtoul(s, '\0', 10);
}

/* Encontra uma estrutura não usada. */
find_free(void)
{
  register int t;

  for(t=0; addr_info[t].name[0] && t<MAX; ++t);

  if(t==MAX) return -1; /* nenhum elemento livre */
  return t;
}

/* Apaga um endereço */
void delete(void)
{
  register int slot;
  char s[80];

  printf("Digite o registro #: ");
  gets(s);
  slot = atoi(s);
  if(slot>=0 && slot < MAX)
    addr_info[slot].name[0] = '\0';
}

/* Mostra a lista na tela. */
void list(void)
{
  register int t;

  for(t=0; t<MAX; ++t) {
    if(addr_info[t].name[0]) {
      printf("%s\n", addr_info[t].name);
      printf("%s\n", addr_info[t].street);
      printf("%s\n", addr_info[t].city);
```

```
        printf("%s\n", addr_info[t].state);
        printf("%lu\n", addr_info[t].zip);
      }
    }
    printf("\n\n");
}
```

Passando Estruturas para Funções

Esta seção discute a passagem de estruturas e seus elementos para funções.

Passando Elementos de Estrutura para Funções

Quando você passa um elemento de uma variável estrutura para uma função, está, de fato, passando o valor desse elemento para a função. Assim, você está passando uma variável simples (a menos, é claro, que o elemento seja complexo, como uma matriz de caracteres). Por exemplo, considere esta estrutura:

```
struct fred
{
  char x;
  int y;
  float z;
  char s[10];
} mike;
```

A seguir são mostrados exemplos de cada elemento sendo passado para uma função:

```
func(mike.x);  /* passa o valor do caractere de x */
func2(mike.y); /* passa o valor inteiro de y */
func3(mike.z); /* passa o valor float de z */
func4(mike.s); /* passa o endereço da string s */
func(mike.s[2]); /* passa o valor do caractere de s[2] */
```

Porém, se você quiser passar o endereço de um elemento individual da estrutura, ponha o operador **&** antes do nome da estrutura. Por exemplo, para passar o endereço dos elementos da estrutura **mike**, escreva

```
func(&mike.x);  /* passa o endereço do caractere x */
```

```
func2(&mike.y); /* passa o endereço do inteiro y */
func3(&mike.z); /* passa o endereço do float z */
func4(mike.s); /* passa o endereço da string s */
func(&mike.s[2]); /* passa o endereço do caractere s[2] */
```

Lembre-se de que o operador **&** precede o nome da estrutura, não o nome do elemento individual. Note também que o elemento string **s** já significa um endereço, de forma que o **&** não é necessário.

Passando Estruturas Inteiras para Funções

Quando uma estrutura é usada como um argumento para uma função, a estrutura inteira é passada usando o método padrão de chamada por valor. Obviamente, isso significa que quaisquer alterações podem ser feitas no conteúdo da estrutura dentro da função para a qual ela é passada sem afetar a estrutura usada como argumento.

NOTA: Em algumas versões antigas de C, estruturas não podiam ser passadas para funções. Em vez disso, elas eram tratadas como matrizes, e apenas um ponteiro para a estrutura era passado. Tenha isso em mente se você alguma vez usar um compilador C antigo.

Quando usar uma estrutura como um parâmetro, lembre-se de que o tipo de argumento deve coincidir com o tipo de parâmetro. Por exemplo, neste programa, tanto o argumento **arg** como o parâmetro **parm** são declarados como o mesmo tipo de estrutura.

```
#include <stdio.h>

/* Define um tipo de estrutura. */
struct struct_type {
  int a, b;
  char ch;
} ;
void f1(struct struct_type parm);

void main(void)
{
  struct struct_type arg;

  arg.a = 1000;

  f1(arg);
```

```
}

void f1(struct struct_type parm)
{
  printf("%d", parm.a);
}
```

Como este programa ilustra, se você declarar parâmetros que são estruturas, deverá tornar a declaração do tipo de estrutura global, para que todas as partes do seu programa possam usá-la. Por exemplo, se **struct_type** tivesse sido declarada dentro de **main()** (por exemplo), então não seria visível a **f1()**.

Como acabamos de enunciar, ao passar estruturas, o tipo do argumento deve coincidir com o tipo do parâmetro. Não é suficiente que eles sejam fisicamente semelhantes; os nomes dos seus tipos devem coincidir. Por exemplo, a versão seguinte do programa anterior é incorreta e não compilará porque o nome do tipo do argumento usado para chamar **f1()** difere do nome do tipo de seu parâmetro.

```
/* Este programa está errado e não poderá ser compilado. */
#include <stdio.h>

/* Define um tipo de estrutura. */
struct struct_type {
  int a, b;
  char ch;
} ;

/* Define uma estrutura similar a struct_type,
   mas com outro nome. */
struct struct_type2 {
  int a, b;
  char ch;
} ;

void f1(struct struct_type2 parm);

void main(void)
{
  struct struct_type arg;

  arg.a = 1000;

  f1(arg); /* erro de tipos */
```

```
}

void f1(struct struct_type2 parm)
{
  printf("%d", parm.a);
}
```

Ponteiros para Estruturas

C permite ponteiros para estruturas exatamente como permite ponteiros para outros tipos de variáveis. No entanto, há alguns aspectos especiais de ponteiros de estruturas que você deve conhecer.

Declarando um Ponteiro para Estrutura

Como outros ponteiros, você declara ponteiros para estrutura colocando * na frente do nome da estrutura. Por exemplo, assumindo a estrutura previamente definida **addr**, o código seguinte declara **addr_pointer** como um ponteiro para dados daquele tipo.

```
struct addr *addr_pointer;
```

Usando Ponteiros para Estruturas

Há dois usos primários para ponteiros de estrutura: gerar uma chamada por referência para uma função e criar listas encadeadas e outras estruturas de dados dinâmicas usando o sistema de alocação de C. Este capítulo cobre o primeiro uso. O segundo uso é coberto detalhadamente na Parte 3.

Há um prejuízo maior em passar todas as estruturas, exceto as mais simples, para funções: o tempo extra necessário para colocar (e tirar) todos os elementos da estrutura na pilha. Em estruturas simples, com poucos elementos, esse tempo extra não é tão grande. Se vários elementos são usados, porém, ou se alguns dos elementos são matrizes, a performance pode ser reduzida a níveis inaceitáveis. A solução para esse problema é passar apenas um ponteiro para uma função.

Quando um ponteiro para uma estrutura é passado para uma função, apenas o endereço da estrutura é colocado (e tirado) da pilha. Isso contribui para chamadas muito rápidas a funções. Uma segunda vantagem, em alguns casos, é

quando a função precisa referenciar o argumento real em lugar de uma cópia. Passando um ponteiro, é possível alterar o conteúdo dos elementos reais da estrutura usada na chamada.

Para encontrar o endereço da variável estrutura, deve-se colocar o operador & antes do nome da estrutura. Por exemplo, dado o seguinte fragmento:

```
struct bal {
  float balance;
  char name[80];
} person;

struct bal *p; /* declara um ponteiro para estrutura */
```

então

```
p = &person;
```

põe o endereço da estrutura **person** no ponteiro **p**.

Para acessar os elementos de uma estrutura usando um ponteiro para a estrutura, você deve usar o operador ->. Por exemplo, isso referencia o campo **balance**:

```
p->balance
```

O -> é normalmente chamado de *operador seta*, e consiste no sinal de subtração seguido pelo sinal de maior. A seta é usada no lugar do operador ponto quando se está acessando um elemento de estrutura por meio de um ponteiro para a estrutura.

Para ver como um ponteiro para estrutura pode ser usado, examine este programa simples, que escreve as horas, minutos e segundos na tela usando um relógio (*timer*) por software.

```
/* Mostra um relógio por software. */
#include <stdio.h>

#define DELAY 128000

struct my_time {
  int hours;
  int minutes;
  int seconds;
} ;
```

```
void display(struct my_time *t);
void update(struct my_time *t);
void delay(void);

void main(void)
{
  struct my_time systime;

  systime.hours = 0;
  systime.minutes = 0;
  systime.seconds = 0;

  for(;;) {
    update(&systime);
    display(&systime);
  }
}

void update(struct my_time *t)
{
  t->seconds++;
  if(t->seconds==60) {
    t->seconds = 0;
    t->minutes++;
  }

  if(t->minutes==60) {
    t->minutes = 0;
    t->hours++;
  }

  if(t->hours==24) t->hours = 0;
  delay();
}

void display(struct my_time *t)
{
  printf("%02d:", t->hours);
  printf("%02d:", t->minutes);
  printf("%02d\n", t->seconds);
}

void delay(void)
{
```

```
    long int t;

    /* mude isto como necessário */
    for(t=1; t<DELAY; ++t) ;
}
```

A temporização desse programa é ajustada alterando-se o **DELAY**.

Como você pode ver, uma estrutura global chamada **my_time** foi definida, mas nenhuma variável foi declarada. Dentro de **main()**, a estrutura **systime** foi declarada e inicializada em 00:00:00. Isso significa que **systime** é conhecido diretamente apenas na função **main()**.

O endereço de **systime** é passado às duas funções **update()** (que modifica o tempo) e **display()** (que imprime a hora). Nas duas funções, o argumento é declarado como um ponteiro para a estrutura **my_time**.

Dentro de **update()** e **display()**, cada elemento de **systime** é acessado via um ponteiro. Como **update()** recebe um ponteiro para a estrutura **systime**, ele pode atualizar seu valor. Por exemplo, para ajustar a hora de volta a 0, quando se atinge 24:00:00, **update()** contém esta linha de código:

```
if(t->hours==24) t->hours = 0;
```

Essa linha de código informa ao compilador para tomar o endereço de **t** (que aponta para **systime** em **main()**) e atribuir zero a seu elemento **hours**.

Lembre-se de usar o operador ponto para acessar elementos de estruturas quando estiver operando na própria estrutura. Quando você tem um ponteiro para a estrutura, use o operador seta.

Matrizes e Estruturas Dentro de Estruturas

Um elemento de estrutura pode ser simples ou complexo. Um elemento simples é qualquer dos tipos de dados intrínsecos, como um caractere ou inteiro. Você já viu um elemento complexo: a matriz de caracteres usada em **addr**. Outros tipos de dados complexos são matrizes unidimensionais e multidimensionais e outros tipos de dados e estruturas.

Um elemento de estrutura que é uma matriz é tratada como você poderia esperar a partir dos exemplos anteriores. Por exemplo, considere esta estrutura:

```
struct x {
```

```
  int a[10][10]; /* matriz de 10 x 10 itens */
  float b;
} y;
```

Para referenciar o inteiro 3,7 em **a** da estrutura **y**, escreva

```
y.a[3][7]
```

Quando um elemento de uma estrutura é um elemento de outra estrutura, ela é chamada estrutura *aninhada*. Por exemplo, a estrutura **address** é aninhada em **emp** neste exemplo:

```
struct emp {
  struct addr address; /* estrutura aninhada */
  float wage;
} worker;
```

Aqui, a estrutura **emp** foi definida como tendo dois elementos. O primeiro elemento é a estrutura do tipo **addr**, que contém o endereço de um empregado. O outro é **wage**, que contém o salário do empregado. O seguinte fragmento de código atribui 93456 ao elemento **zip** de **address**.

```
worker.address.zip = 93456;
```

Como você pode ver, os elementos de cada estrutura são referenciados do mais externo ao mais interno. O padrão ANSI C especifica que as estruturas podem ser aninhadas até 15 níveis. A maioria dos compiladores permite mais.

Campos de Bits

Ao contrário da maioria das linguagens de computador, C tem um método intrínseco para acessar um único bit dentro de um byte. Isso pode ser útil por um certo número de razões:

- Se o armazenamento é limitado, você pode armazenar diversas variáveis *booleanas* (verdadeiro/falso) em um byte.
- Certos dispositivos transmitem informações codificadas nos bits.
- Certas rotinas de criptografia precisam acessar os bits dentro de um byte.

Embora essas tarefas possam ser realizadas usando os operadores bit a bit, um campo de bit pode acrescentar mais estrutura (e possivelmente eficiência) ao seu código.

Para acessar os bits, C usa um método baseado na estrutura. Um campo de bits é, na verdade, apenas um tipo de elemento de estrutura que define o comprimento, em bits, do campo. A forma geral de uma definição de campo de bit é

```
struct identificador{
  tipo nome1 : comprimento;
  tipo nome2 : comprimento;
    .
    .
    .
  tipo nomeN : comprimento;
} lista_de_variáveis;
```

Um campo de bits deve ser declarado como **int**, **unsigned** ou **signed**. Campos de bits de comprimento 1 devem ser declarados como **unsigned**, porque um único bit não pode ter um sinal. (Alguns compiladores só permitem campos de bit **unsigned**.) O número de bits no campo de bits é especificado por *comprimento*.

Campos de bits são freqüentemente usados quando se analisa a entrada de um dispositivo de hardware. Por exemplo, o estado da porta do adaptador de comunicações seriais poderia retornar um byte de estado organizado desta forma:

Bit	Significado quando ligado
0	Alteração na linha clear-to-send
1	Alteração em data-set-ready
2	Borda de subida da portadora detectada
3	Alteração na linha de recepção
4	Clear-to-send
5	Data-set-ready
6	Chamada do telefone
7	Sinal recebido

Você pode representar a informação em um byte de estado usando o seguinte campo de bits:

```
struct status_type {
  unsigned delta_cts:      1;
  unsigned delta_dsr:      1;
  unsigned tr_edge:  1;
```

```
  unsigned delta_rec:     1;
  unsigned cts:           1;
  unsigned dsr:           1;
  unsigned ring:          1;
  unsigned rec_line:      1;
} status;
```

Você pode usar uma rotina semelhante a esta, mostrada aqui, para permitir que um programa determine quando pode enviar ou receber dados.

```
status = get_port_status();
if(status.cts) printf("livre para enviar");
if(status.dsr) printf("dados prontos");
```

Para atribuir um valor a um campo de bits, simplesmente use a forma que você usaria para qualquer outro tipo de elemento de estrutura. Por exemplo, este fragmento de código limpa o campo **ring**:

```
status.ring = 0;
```

Como você pode ver, a partir destes exemplos, cada campo de bits é acessado com o operador ponto. Porém, se a estrutura é referenciada por meio de um ponteiro, você deve usar o operador ->.

Não é necessário dar um nome a todo campo de bits. Isso torna fácil alcançar o bit que você quer, contornando os não usados. Por exemplo, se apenas **cts** e **dsr** importam, você poderia declarar a estrutura **status_type** desta forma:

```
struct status_type {
  unsigned :          4;
  unsigned cts:       1;
  unsigned dsr:       1;
} status;
```

Além disso, note que os bits após **dsr** não precisam ser especificados se não são usados.

É válido misturar elementos normais de estrutura com elementos de campos de bit. Por exemplo,

```
struct emp {
  struct addr address;
  float pay;
  unsigned lay_off:  1; /* ocioso ou ativo */
```

```
  unsigned hourly:   1; /* pagamento por horas ou salário */
  unsigned deduction:3; /* deduções de imposto */
};
```

define um registro de empregado que utiliza apenas 1 byte para segurar três pedaços de informação: o estado do empregado se contratado ou assalariado, e o número de deduções. Sem o campo de bit, essa informação usaria 3 bytes.

Variáveis de campo de bits têm certas restrições. Você não pode obter o endereço de uma variável de campo de bits. Variáveis de campo de bits não podem ser organizadas em matrizes. Você não pode ultrapassar os limites de um inteiro. Não pode saber, de máquina para máquina, se os campos estarão dispostos da esquerda para direita ou da direita para a esquerda. Em outras palavras, qualquer código que use campos de bits pode ter algumas dependências da máquina.

Uniões

Em C, uma *union* é uma posição de memória que é compartilhada por duas ou mais variáveis diferentes, geralmente de tipos diferentes, em momentos diferentes. A definição de uma **union** é semelhante à definição de estrutura. Sua forma geral é

union *identificador* {
  tipo nome_da_variável;
  tipo nome_da_variável;
  tipo nome_da_variável;
    .
    .
    .
} *variáveis_união;*

Por exemplo,

```
union u_type {
  int i;
  char ch;
};
```

Essa definição não declara quaisquer variáveis. Você pode declarar uma variável colocando seu nome no final da definição ou usando um comando de declaração separado. Para declarar um variável **union cnvt** do tipo **u_type**, usando a definição dada há pouco, escreva

```
union u_type cnvt;
```

Na **union cnvt**, tanto o inteiro **i** como o caractere **ch** compartilham a mesma posição de memória. (Obviamente, **i** ocupa 2 bytes e **ch** usa apenas 1.) A Figura 7.2 mostra como **i** e **ch** compartilham o mesmo endereço. A qualquer momento, você pode referir-se ao dado armazenado em **cnvt** como um inteiro ou um caractere.

Quando uma **union** é declarada, o compilador cria automaticamente uma variável grande o bastante para conter o maior tipo de variável da **union**. Por exemplo (supondo inteiros de 2 bytes), **cnvt** tém dois bytes de comprimento para poder armazenar **i**, embora **ch** exija somente um byte.

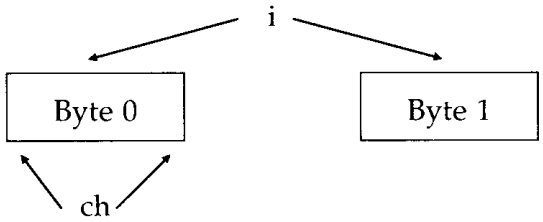

Figura 7.2 Como **i** e **ch** utilizam a **union cnvt** (supondo um inteiro de 2 bytes).

Para acessar um elemento da **union**, deve-se usar a mesma sintaxe que seria usada para estruturas: os operadores ponto e seta. Se você está operando na **union** diretamente, use o operador ponto. Se a variável **union** é acessada por meio de um ponteiro, use o operador seta. Por exemplo, para atribuir o inteiro 10 ao elemento **i** de **cnvt**, escreva

```
cnvt.i = 10;
```

No próximo exemplo, um ponteiro para **cnvt** é passado para uma função:

```
void func1(union u_type *un)
{
  un->i = 10; /* atribui 10 a cnvt usando uma função */
}
```

Usar uma **union** pode ajudar na produção de código independente da máquina (portável). Como o compilador não perde o tamanho real das variáveis que perfazem a união, nenhuma dependência da máquina é produzida. Você não precisa preocupar-se com o tamanho de **int**, **long**, **float** ou o que quer que seja.

Unions são usadas freqüentemente quando conversões de tipo são necessárias, porque você pode referenciar os dados contidos na union de maneiras diferentes. Por exemplo, você pode usar uma **union** para manipular os bytes que constituem um **double**, a fim de alterar sua precisão ou para realizar um tipo de arredondamento incomum.

Para ter uma idéia da utilidade de uma **union** quando conversões não padrão de tipos são necessárias, considere o problema de escrever um inteiro em um arquivo de disco. A biblioteca padrão de C não contém funções projetadas para especificamente escrever um inteiro em um arquivo. Embora você possa escrever qualquer tipo de dado (incluindo um inteiro) em um arquivo, usar **fwrite()** é muito para uma operação tão simples.

Contudo, usando uma **union**, você pode criar facilmente uma função chamada **putw()**, a qual escreve a representação binária de um inteiro em um arquivo um byte por vez. Para ver como, primeiro crie uma **union** consistindo de um inteiro e uma matriz de caractere de 2 bytes:

```
union pw {
  int i;
  char ch[2];
};
```

Agora, **putw()** pode ser escrita desta forma:

```
putw(union pw word, FILE *fp)
{
  putc(word->ch[0], fp); /* escreve a primeira metade */
  putc(word->ch[1], fp); /* escreve a segunda metade */
}
```

Embora possa ser chamada com um inteiro, **putw()** ainda pode usar a função **putc()** para escrever um inteiro em um arquivo em disco um byte por vez.

Enumerações

Uma enumeração é uma extensão da linguagem C acrescentada pelo padrão ANSI. Uma *enumeração* é um conjunto de constantes inteiras que especifica todos os valores legais que uma variável desse tipo pode ter. Enumerações são comuns na vida cotidiana. Por exemplo, uma enumeração das moedas usadas nos Estados Unidos é

penny, nickel, dime, quarter, half-dollar, dollar

Enumerações são definidas de forma semelhante a estruturas; a palavra-chave **enum** assinala o início de um tipo de enumeração. A forma geral para enumeração é

enum *identificador* { *lista de enumeração* } *lista_de_variáveis*;

Aqui, tanto o identificador da enumeração quanto a lista de variáveis são opcionais. Análogo às estruturas, o identificador da enumeração é usado para declarar variáveis daquele tipo. O fragmento de código seguinte define uma enumeração chamada **coin** e declara **money** como sendo desse tipo:

```
enum coin {penny, nickel, dime, quarter,
           half_dollar, dollar};
enum coin money;
```

Dada essa definição e declaração, os tipos de comandos seguintes são perfeitamente válidos:

```
money = dime;
if(money==quarter) printf("Money é um quarto\n");
```

O ponto-chave para o entendimento de uma enumeração é que cada símbolo representa um valor inteiro. Dessa forma, eles podem ser usados em qualquer lugar em que um inteiro pode ser usado. A cada símbolo é dado um valor maior em uma unidade do precedente. O valor do primeiro símbolo da enumeração é 0. Assim,

```
printf("%d %d", penny, dime);
```

mostra **0 2** na tela.

Você pode especificar o valor de um ou mais dos símbolos usando um inicializador. Isso é feito colocando-se um sinal de igual e um valor inteiro após o símbolo. Os símbolos que aparecem após os inicializadores recebem valores maiores que o da inicialização precedente. Por exemplo, o código seguinte atribui o valor 100 a **quarter**:

```
enum coin {penny, nickel, dime, quarter=100,
           half_dollar, dollar};
```

Agora, os valores destes símbolos são

```
penny        0
nickel       1
dime         2
quarter    100
half_dollar 101
dollar     102
```

Uma suposição comum porém errônea sobre enumerações é que os símbolos podem ser enviados para a saída e recebidos da entrada diretamente. Isso não acontece. Por exemplo, o fragmento de código seguinte não executa como desejado:

```
/* isso não funcionará */
money = dollar;
printf("%s", money);
```

Lembre-se de que **dollar** é simplesmente um nome para um inteiro, não é uma string. Pela mesma razão, você não pode usar esse código para alcançar os resultados desejados:

```
/* esse código está errado */
strcpy(money, "dime");
```

Isto é, uma string que contém o nome de um símbolo não é automaticamente convertida naquele símbolo.

De fato, criar um código para inserir e retirar símbolos de uma enumeração é um tanto tedioso (a menos que você esteja querendo determinar seus valores inteiros). Por exemplo, você precisa do seguinte código para mostrar em palavras o tipo de moeda que **money** contém:

```
switch(money) {
  case penny:  printf("penny");
    break;
  case nickel:  printf("nickel");
    break;
  case dime:  printf("dime");
    break;
  case quarter: printf("quarter");
    break;
  case half_dollar:  printf("half_dollar");
    break;
  case dollar:  printf("dollar");
}
```

Algumas vezes, você pode declarar uma matriz de strings e usar o valor da enumeração como um índice para traduzir um valor da enumeração na sua string correspondente. Por exemplo, este código também mostra a string apropriada:

```
char name[] [12] = {
  "penny",
  "nickel",
  "dime",
  "quarter",
  "half_dollar",
  "dollar"
};
printf("%s", name[money]);
```

Logicamente, isso funcionará apenas quando nenhum símbolo for inicializado, porque a matriz de strings deve ser indexada começando por 0.

Uma vez que os valores de uma enumeração têm de ser convertidos manualmente em suas strings legíveis para nós, uma E/S do console, eles são mais úteis em rotinas que não fazem essas conversões. Uma enumeração é freqüentemente usada para definir uma tabela de símbolos de um compilador, por exemplo. As enumerações também são usadas para ajudar a provar a validade de um programa, produzindo uma verificação redundante em tempo de compilação, confirmando que apenas valores válidos sejam atribuídos a uma variável.

Usando sizeof para Assegurar Portabilidade

Você viu que estruturas, uniões e enumerações podem ser usadas para criar variáveis de diferentes tamanhos e que o tamanho real dessas variáveis pode mudar de máquina para máquina. O operador unário **sizeof** calcula o tamanho de qualquer variável ou tipo e pode ajudar a eliminar códigos dependendes da máquina de seus programas. Este operador é especialmente útil onde as estruturas ou uniões dizem respeito.

Por exemplo, assuma uma implementação de C, comum a muitos compiladores C para microcomputadores, que tem os tamanhos dos tipos de dados mostrados aqui:

Tipo	Tamanho em bytes
char	1
int	2
float	4

Portanto, o seguinte código escreverá os números 1, 2 e 4 na tela:

```
char ch;
int i;
float f;

printf("%d", sizeof(ch));

printf("%d", sizeof(i));

printf("%d", sizeof(f));
```

O tamanho de uma estrutura é igual a *ou maior que* a soma dos tamanhos dos seus componentes. Por exemplo,

```
struct s {
  char ch;
  int i;
  float f;
} s_var;
```

Aqui, **sizeof(s_var)** vale pelo menos 7 (4 + 2 + 1). No entanto, o tamanho de **s_var** pode ser maior porque é permitido ao compilador "preencher" uma estrutura para obter alinhamento de palavras ou parágrafos. (Um parágrafo são 16 bytes.) Como o tamanho de uma estrutura pode ser maior que a soma dos tamanhos dos seus componentes, você deve usar **sizeof** sempre que precisar saber o tamanho de uma estrutura.

Como **sizeof** é um operador avaliado durante a compilação, toda a informação necessária para determinar o tamanho de qualquer variável é conhecida em tempo de compilação. Isto é especialmente importante para as **union**s, já que o tamanho de uma **union** é sempre igual ao tamanho do seu maior componente. Por exemplo, considere

```
union u {
  char ch;
  int i;
  float f;
} u_var;
```

Aqui, o **sizeof(u_var)** é 4. No tempo de execução, não importa o que a união **u_var** está realmente guardando. Tudo o que importa é o tamanho da maior variável que pode ser armazenada porque a **union** tem de ser do tamanho do seu maior elemento.

typedef

C permite que você defina explicitamente novos nomes aos tipos de dados, utilizando a palavra-chave **typedef**. Você não está realmente *criando* uma nova classe de dados, mas, ao contrário, definindo um novo nome para um tipo já existente. Esse processo pode ajudar a tornar programas dependentes da máquina um pouco mais portáveis. Se você definir seu próprio nome de tipo para cada tipo de dados dependente de máquina usado pelo seu programa, apenas os comandos **typedef** teriam de ser mudados quando você compilar em um novo ambiente. Também pode auxiliar numa autodocumentação do seu código, permitindo nomes mais descritivos aos tipos de dados padrões. A forma geral de um comando **typedef** é

typedef *tipo novonome;*

onde *tipo* é qualquer tipo de dados permitido e *novonome* é o novo nome para esse tipo. O novo nome que você define é uma opção, não uma substituição, ao nome do tipo existente.

Por exemplo, você poderia criar um novo nome para **float** usando

```
typedef float balance;
```

Esse comando diz ao compilador para reconhecer **balance** como outro nome para **float**. A seguir, você poderia criar uma variável **float**, usando **balance**:

```
balance over_due;
```

Aqui, **over_due** é uma variável de ponto flutuante do tipo **balance**, que é uma outra palavra para **float**.

Agora que **balance** foi definido, ele pode ser usado no lado direito de um outro **typedef**. Por exemplo,

```
typedef balance overdraft;
```

diz ao compilador para reconhecer **overdraft** como um outro nome para **balance**, que é um outro nome para **float**.

Existe uma aplicação de **typedef** que você pode achar especialmente útil: **typedef** pode ser usada para simplificar a declaração de variáveis estrutura, **union** ou de enumeração. Por exemplo, considere a seguinte declaração:

```
struct mystruct {
  unsigned x;
  float f;
};
```

Para declarar uma variável do tipo **mystruct**, você deve usar uma declaração como esta:

```
struct mystruct s;
```

Embora certamente não haja nada de errado com esta declaração, ela exige o uso de dois identificadores: **struct** e **mystruct**. No entanto, se você aplicar **typedef** à declaração de **mystruct**, como exibido aqui,

```
typedef struct_mystruct {
  unsigned x;
  float f;
} mystruct;
```

então você pode declarar variáveis deste tipo de estrutura usando a seguinte declaração:

```
mystruct s;
```

O ponto é que através do uso de **typedef** na declaração de **mystruct**, você cria um novo identificador de tipo composto de um único nome. Embora isto seja tecnicamente apenas uma questão de conveniência, certamente vale a pena se você for declarar uma quantidade razoável de variáveis estrutura. Esta mesma técnica pode ser aplicada a **unions** e enumerações.

O uso de **typedef** pode tornar seu código mais fácil de ler e mais fácil de portar para um novo equipamento. Mas lembre-se, você não está criando qualquer tipo de dados novo.