8. Tipos estruturados

W. Celes e J. L. Rangel

Na linguagem C, existem os tipos básicos (`char`, `int`, `float`, etc.) e seus respectivos ponteiros que podem ser usados na declaração de variáveis. Para estruturar dados complexos, nos quais as informações são compostas por diversos campos, necessitamos de mecanismos que nos permitam agrupar tipos distintos. Neste capítulo, apresentaremos os mecanismos fundamentais da linguagem C para a estruturação de tipos.

8.1. O tipo estrutura

Em C, podemos definir um tipo de dado cujos campos são compostos de vários valores de tipos mais simples. Para ilustrar, vamos considerar o desenvolvimento de programas que manipulam pontos no plano cartesiano. Cada ponto pode ser representado por suas coordenadas *x* e *y*, ambas dadas por valores reais. Sem um mecanismo para agrupar as duas componentes, teríamos que representar cada ponto por duas variáveis independentes.

```
float x;
float y;
```

No entanto, deste modo, os dois valores ficam dissociados e, no caso do programa manipular vários pontos, cabe ao programador não misturar a coordenada *x* de um ponto com a coordenada *y* de outro. Para facilitar este trabalho, a linguagem C oferece recursos para agruparmos dados. Uma estrutura, em C, serve basicamente para agrupar diversas variáveis dentro de um único contexto. No nosso exemplo, podemos definir uma estrutura `ponto` que contenha as duas variáveis. A sintaxe para a definição de uma estrutura é mostrada abaixo:

```
struct ponto {
   float x;
   float y;
};
```

Desta forma, a estrutura `ponto` passa a ser um tipo e podemos então declarar variáveis deste tipo.

```
struct ponto p;
```

Esta linha de código declara `p` como sendo uma variável do tipo `struct ponto`. Os elementos de uma estrutura podem ser acessados através do operador de acesso "ponto" (**.**). Assim, é válido escrever:

```
ponto.x = 10.0;
ponto.y = 5.0;
```

Manipulamos os elementos de uma estrutura da mesma forma que variáveis simples. Podemos acessar seus valores, atribuir-lhes novos valores, acessar seus endereços, etc.

Exemplo: Capturar e imprimir as coordenadas de um ponto.

Para exemplificar o uso de estruturas em programas, vamos considerar um exemplo simples em que capturamos e imprimimos as coordenadas de um ponto qualquer.

```
/* Captura e imprime as coordenadas de um ponto qualquer */

#include <stdio.h>

struct ponto {
   float x;
   float y;
};

int main (void)
{
   struct ponto p;

   printf("Digite as coordenadas do ponto(x y): ");
   scanf("%f %f", &p.x, &p.y);
   printf("O ponto fornecido foi: (%.2f,%.2f)\n", p.x, p.y);
   return 0;
}
```

A variável `p`, definida dentro de `main`, é uma variável local como outra qualquer. Quando a declaração é encontrada, aloca-se, na pilha de execução, um espaço para seu armazenamento, isto é, um espaço suficiente para armazenar todos os campos da estrutura (no caso, dois números reais). Notamos que o acesso ao endereço de um campo da estrutura é feito da mesma forma que com variáveis simples: basta escrever `&(p.x)`, ou simplesmente `&p.x`, pois o operador de acesso ao campo da estrutura tem precedência sobre o operador "endereço de".

Ponteiro para estruturas

Da mesma forma que podemos declarar variáveis do tipo estrutura:

```
struct ponto p;
```

podemos também declarar variáveis do tipo ponteiro para estrutura:

```
struct ponto *pp;
```

Se a variável `pp` armazenar o endereço de uma estrutura, podemos acessar os campos dessa estrutura indiretamente, através de seu ponteiro:

```
(*pp).x = 12.0;
```

Neste caso, os parênteses são indispensáveis, pois o operador "conteúdo de" tem precedência menor que o operador de acesso. O acesso de campos de estruturas é tão comum em programas C que a linguagem oferece outro operador de acesso, que permite acessar campos a partir do ponteiro da estrutura. Este operador é composto por um traço seguido de um sinal de maior, formando uma seta (`->`). Portanto, podemos reescrever a atribuição anterior fazendo:

```
pp->x = 12.0;
```

Em resumo, se temos uma variável estrutura e queremos acessar seus campos, usamos o operador de acesso ponto (`p.x`); se temos uma variável ponteiro para estrutura, usamos o operador de acesso seta (`pp->x`). Seguindo o raciocínio, se temos o ponteiro e queremos acessar o endereço de um campo, fazemos `&pp->x`!

Passagem de estruturas para funções

Para exemplificar a passagem de variáveis do tipo estrutura para funções, podemos reescrever o programa simples, mostrado anteriormente, que captura e imprime as coordenadas de um ponto qualquer. Inicialmente, podemos pensar em escrever uma função que imprima as coordenadas do ponto. Esta função poderia ser dada por:

```
void imprime (struct ponto p)
{
   printf("O ponto fornecido foi: (%.2f,%.2f)\n", p.x, p.y);
}
```

A passagem de estruturas para funções se processa de forma análoga à passagem de variáveis simples, porém exige uma análise mais detalhada. Da forma como está escrita no código acima, a função recebe uma estrutura inteira como parâmetro. Portanto, faz-se uma cópia de toda a estrutura para a pilha e a função acessa os dados desta cópia. Existem dois pontos a serem ressaltados. Primeiro, como em toda passagem por valor, a função não tem como alterar os valores dos elementos da estrutura original (na função `imprime` isso realmente não é necessário, mas seria numa função de leitura). O segundo ponto diz respeito à eficiência, visto que copiar uma estrutura inteira para a pilha pode ser uma operação custosa (principalmente se a estrutura for muito grande). É mais conveniente passar apenas o ponteiro da estrutura, mesmo que não seja necessário alterar os valores dos elementos dentro da função, pois copiar um ponteiro para a pilha é muito mais eficiente do que copiar uma estrutura inteira. Um ponteiro ocupa em geral 4 bytes, enquanto uma estrutura pode ser definida com um tamanho muito grande. Desta forma, uma segunda (e mais adequada) alternativa para escrevermos a função `imprime` é:

```
void imprime (struct ponto* pp)
{
   printf("O ponto fornecido foi: (%.2f,%.2f)\n", pp->x, pp->y);
}
```

Podemos ainda pensar numa função para ler a hora do evento. Observamos que, neste caso, obrigatoriamente devemos passar o ponteiro da estrutura, caso contrário não seria possível passar ao programa principal os dados lidos:

```
void captura (struct ponto* pp)
{
   printf("Digite as coordenadas do ponto(x y): ");
   scanf("%f %f", &p->x, &p->y);
}
```

Com estas funções, nossa função `main` ficaria como mostrado abaixo.

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

   captura(&p);
   imprime(&p);

   return 0;
}
```

Exercício: Função para determinar a distância entre dois pontos.
Considere a implementação de uma função que tenha como valor de retorno a distância entre dois pontos. O protótipo da função pode ser dado por:

```
float distancia (struct ponto *p, struct ponto *q);
```

Nota: A distância entre dois pontos é dada por: $d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$

Alocação dinâmica de estruturas

Da mesma forma que os vetores, as estruturas podem ser alocadas dinamicamente. Por exemplo, é válido escrever:

```
struct ponto* p;
p = (struct ponto*) malloc (sizeof(struct ponto));
```

Neste fragmento de código, o tamanho do espaço de memória alocado dinamicamente é dado pelo operador `sizeof` aplicado sobre o tipo estrutura (`sizeof(struct ponto)`). A função `malloc` retorna o endereço do espaço alocado, que é então convertido para o tipo ponteiro da estrutura `ponto`.

Após uma alocação dinâmica, podemos acessar normalmente os campos da estrutura, através da variável ponteiro que armazena seu endereço:

```
...
p->x = 12.0;
...
```

8.2. Definição de "novos" tipos

A linguagem C permite criar nomes de tipos. Por exemplo, se escrevermos:

```
typedef float Real;
```

podemos usar o nome `Real` como um mnemônico para o tipo `float`. O uso de `typedef` é muito útil para abreviarmos nomes de tipos e para tratarmos tipos complexos. Alguns exemplos válidos de `typedef`:

```
typedef unsigned char UChar;
typedef int* PInt;
typedef float Vetor[4];
```

Neste fragmento de código, definimos `UChar` como sendo o tipo `char` sem sinal, `PInt` como um tipo ponteiro para `int`, e `Vetor` como um tipo que representa um vetor de quatro elementos. A partir dessas definições, podemos declarar variáveis usando estes mnemônicos:

```
Vetor v;
...
v[0] = 3;
...
```

Em geral, definimos nomes de tipos para as estruturas com as quais nossos programas trabalham. Por exemplo, podemos escrever:

```
struct ponto {
   float x;
   float y;
};

typedef struct ponto Ponto;
```

Neste caso, `Ponto` passa a representar nossa estrutura de ponto. Também podemos definir um nome para o tipo ponteiro para a estrutura.

```
typedef struct ponto *PPonto;
```

Podemos ainda definir mais de um nome num mesmo `typedef`. Os dois `typedef` anteriores poderiam ser escritos por:

```
typedef struct ponto Ponto, *PPonto;
```

A sintaxe de um `typedef` pode parecer confusa, mas é equivalente à da declaração de variáveis. Por exemplo, na definição abaixo:

```
typedef float Vector[4];
```

se omitíssemos a palavra `typedef`, estaríamos declarando a variável `Vector` como sendo um vetor de 4 elementos do tipo `float`. Com `typedef`, estamos definindo um nome que representa o tipo vetor de 4 elementos `float`. De maneira análoga, na definição:

```
typedef struct ponto Ponto, *PPonto;
```

se omitíssemos a palavra `typedef`, estaríamos declarando a variável `Ponto` como sendo do tipo `struct ponto` e a variável `PPonto` como sendo do tipo ponteiro para `struct ponto`.

Por fim, vale salientar que podemos definir a estrutura e associar mnemônicos para elas em um mesmo comando:

```
typedef struct ponto {
   float x;
   float y;
} Ponto, *PPonto;
```

É comum os programadores de C usarem nomes com as primeiras letras maiúsculas na definição de tipos. Isso não é uma obrigatoriedade, apenas um estilo de codificação.

8.3. Vetores de estruturas

Já discutimos o uso de vetores para agrupar elementos dos tipos básicos (vetores de inteiros, por exemplo)Nesta seção, vamos discutir o uso de vetores de estruturas, isto é, vetores cujos elementos são estruturas. Para ilustrar a discussão, vamos considerar o cálculo da área de um polígono plano qualquer delimitado por uma seqüência de *n* pontos. A área desse polígono pode ser calculada somando-se as áreas dos trapézios formados pelos lados do polígono e o eixo *x*, conforme ilustra a Figura 8.1.

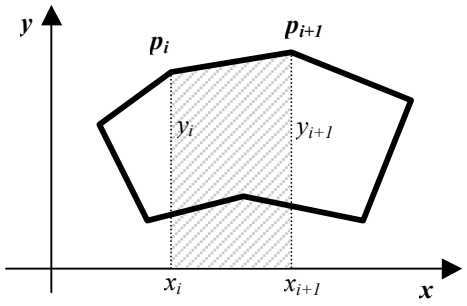

Figura 8.1: Cálculo da área de um polígono.

Na figura, ressaltamos a área do trapézio definido pela aresta que vai do ponto p_i ao ponto p_{i+1}. A área desse trapézio é dada por: $a = (x_{i+1} - x_i)(y_{i+1} + y_i)/2$. Somando-se as "áreas" (algumas delas negativas) dos trapézios definidos por todas as arestas chega-se a área do polígono (as áreas externas ao polígono são anuladas). Se a seqüência de pontos que define o polígono for dada em sentido anti-horário, chega-se a uma "área" de valor negativo. Neste caso, a área do polígono é o valor absoluto do resultado da soma.

Um vetor de estruturas pode ser usado para definir um polígono. O polígono passa a ser representado por um seqüência de pontos. Podemos, então, escrever uma função para calcular a área de um polígono, dados o número de pontos e o vetor de pontos que o representa. Uma implementação dessa função é mostrada abaixo.

```
float area (int n, Ponto* p)
{
   int i, j;
   float a = 0;
   for (i=0; i<n; i++) {
      j = (i+1) % n;        /* próximo índice (incremento circular) */
      a += (p[j].x-p[i].x)*(p[i].y + p[j].y)/2;
   }

   if (a < 0)
     return -a;
   else
     return a;
}
```

Um exemplo de uso dessa função é mostrado no código abaixo:

```
int main (void)
{
  Ponto p[3] = {{1.0,1.0},{5.0,1.0},{4.0,3.0}};
  printf("area = %f\n",area (3,p));
  return 0;
}
```

Exercício: Altere o programa acima para capturar do teclado o número de pontos que delimitam o polígono. O programa deve, então, alocar dinamicamente o vetor de pontos, capturar as coordenadas dos pontos e, chamando a função `area`, exibir o valor da área.

8.4. Vetores de ponteiros para estruturas

Da mesma forma que podemos declarar vetores de estruturas, podemos também declarar vetores de ponteiros para estruturas. O uso de vetores de ponteiros é útil quando temos que tratar um conjunto elementos complexos. Para ilustrar o uso de estruturas complexas, consideremos um exemplo em que desejamos armazenar uma tabela com dados de alunos. Podemos organizar os dados dos alunos em um vetor. Para cada aluno, vamos supor que sejam necessárias as seguintes informações:

- *nome*: cadeia com até 80 caracteres
- *matricula*: número inteiro
- *endereço*: cadeia com até 120 caracteres
- *telefone*: cadeia com até 20 caracteres

Para estruturar esses dados, podemos definir um tipo que representa os dados de um aluno:

```
struct aluno {
   char nome[81];
   int mat;
   char end[121];
   char tel[21];
};

typedef struct aluno Aluno;
```

Vamos montar a tabela de alunos usando um vetor global com um número máximo de alunos. Uma primeira opção é declarar um vetor de estruturas:

```
#define MAX 100
Aluno tab[MAX];
```

Desta forma, podemos armazenar nos elementos do vetor os dados dos alunos que queremos organizar. Seria válido, por exemplo, uma atribuição do tipo:

```
...
tab[i].mat = 9912222;
...
```

No entanto, o uso de vetores de estruturas tem, neste caso, uma grande desvantagem. O tipo `Aluno` definido acima ocupa pelo menos `227` (=`81+4+121+21`) bytes[1]. A declaração de um vetor desta estrutura representa um desperdício significativo de memória, pois provavelmente estaremos armazenando de fato um número de alunos bem inferior ao máximo estimado. Para contornar este problema, podemos trabalhar com um vetor de ponteiros.

```
typedef struct aluno *PAluno;

#define MAX 100
PAluno tab[MAX];
```

Assim, cada elemento do vetor ocupa apenas o espaço necessário para armazenar um ponteiro. Quando precisarmos alocar os dados de um aluno numa determinada posição

[1] Provavelmente o tipo ocupará mais espaço, pois os dados têm que estar alinhados para serem armazenados na memória.

do vetor, alocamos dinamicamente a estrutura `Aluno` e guardamos seu endereço no vetor de ponteiros.

Considerando o vetor de ponteiros declarado acima como uma variável global, podemos ilustrar a implementação de algumas funcionalidades para manipular nossa tabela de alunos. Inicialmente, vamos considerar uma função de inicialização. Uma posição do vetor estará vazia, isto é, disponível para armazenar informações de um novo aluno, se o valor do seu elemento for o ponteiro nulo. Portanto, numa função de inicialização, podemos atribuir `NULL` a todos os elementos da tabela, significando que temos, a princípio, uma tabela vazia.

```
void inicializa (void)
{
   int i;
   for (i=0; i<MAX; i++)
      tab[i] = NULL;
}
```

Uma segunda funcionalidade que podemos prever armazena os dados de um novo aluno numa posição do vetor. Vamos considerar que os dados serão fornecidos via teclado e que uma posição onde os dados serão armazenados será passada para a função. Se a posição da tabela estiver vazia, devemos alocar uma nova estrutura; caso contrário, atualizamos a estrutura já apontada pelo ponteiro.

```
void preenche (int i)
{
   if (tab[i]==NULL)
      tab[i] = (PAluno)malloc(sizeof(Aluno));

   printf("Entre com o nome:");
   scanf(" %80[^\n]", tab[i]->nome);
   printf("Entre com a matricula:");
   scanf("%d", &tab[i]->mat);
   printf("Entre com o endereco:");
   scanf(" %120[^\n]", tab[i]->end);
   printf("Entre com o telefone:");
   scanf(" %20[^\n]", tab[i]->tel);
}
```

Podemos também prever uma função para remover os dados de um aluno da tabela. Vamos considerar que a posição da tabela a ser liberada será passada para a função:

```
void remove (int i)
{
   if (tab[i] != NULL)
   {
      free(tab[i]);
      tab[i] = NULL;
   }
}
```

Para consultarmos os dados, vamos considerar uma função que imprime os dados armazenados numa determinada posição do vetor:

```
void imprime (int i)
{
   if (tab[i] != NULL)
   {
      printf("Nome: %s\n", tab[i]->nome);
      printf("Matrícula: %d\n", tab[i]->mat);
      printf("Endereço: %s\n", tab[i]->end);
      printf("Telefone: %s\n", tab[i]->tel);
   }
}
```

Por fim, podemos implementar uma função que imprima os dados de todos os alunos da tabela:

```
void imprime_tudo (void)
{
   int i;
   for (i=0; i<MAX; i++)
      imprime(i);
}
```

Exercício. Faça um programa que utilize as funções da tabela de alunos escritas acima.

Exercício. Re-escreva as funções acima sem usar uma variável global.
Sugestão: Crie um tipo `Tabela` e faça as funções receberem este tipo como primeiro parâmetro.

8.5. Tipo união**

Em C, uma *união* é uma localização de memória que é compartilhada por diferentes variáveis, que podem ser de tipos diferentes. As uniões são usadas quando queremos armazenar valores heterogêneos num mesmo espaço de memória. A definição de uma união é parecida com a de uma estrutura:

```
union exemplo
{
   int i;
   char c;
}
```

Análogo à estrutura, este fragmento de código não declara nenhuma variável, apenas define o tipo união. Após uma definição, podemos declarar variáveis do tipo união:

```
union exemplo v;
```

Na variável `v`, os campos `i` e `c` compartilham o mesmo espaço de memória. A variável ocupa pelo menos o espaço necessário para armazenar o maior de seus campos (um inteiro, no caso).

O acesso aos campos de uma união é análogo ao acesso a campos de uma estrutura. Usamos o operador ponto (`.`) para acessá-los diretamente e o operador seta (`->`) para acessá-los através de um ponteiro da união. Assim, dada a declaração acima, podemos escrever:

```
v.i = 10;
```

ou

```
v.c = 'x';
```

Salientamos, no entanto, que apenas um único elemento de uma união pode estar armazenado num determinado instante, pois a atribuição a um campo da união sobrescreve o valor anteriormente atribuído a qualquer outro campo.

8.6. Tipo enumeração**

Uma enumeração é um conjunto de constantes inteiras com nomes que especifica os valores legais que uma variável daquele tipo pode ter. É uma forma mais elegante de organizar valores constantes. Como exemplo, consideremos a criação de um tipo booleano. Variáveis deste tipo podem receber os valores 0 (FALSE) ou 1 (TRUE).

Poderíamos definir duas constantes simbólicas dissociadas e usar um inteiro para representar o tipo booleano:

```
#define FALSE  0
#define TRUE   1

typedef int Bool;
```

Desta forma, as definições de `FALSE` e `TRUE` permitem a utilização destes símbolos no código, dando maior clareza, mas o tipo booleano criado, como é equivalente a um inteiro qualquer, pode armazenar qualquer valor inteiro, não apenas `FALSE` e `TRUE`, que seria mais adequado. Para validarmos os valores atribuídos, podemos enumerar os valores constantes que um determinado tipo pode assumir, usando `enum`:

```
enum bool {
   FALSE,
   TRUE
};

typedef enum bool Bool;
```

Com isto, definimos as constantes `FALSE` e `TRUE`. Por *default*, o primeiro símbolo representa o valor 0, o seguinte o valor 1, e assim por diante. Poderíamos explicitar os valores dos símbolos numa enumeração, como por exemplo:

```
enum bool {
   TRUE = 1,
   FALSE = 0,
};
```

No exemplo do tipo booleano, a numeração *default* coincide com a desejada (desde que o símbolo `FALSE` preceda o símbolo `TRUE` dentro da lista da enumeração).

A declaração de uma variável do tipo criado pode ser dada por:

```
Bool resultado;
```

onde `resultado` representa uma variável que pode receber apenas os valores `FALSE` (`0`) ou `TRUE` (`1`).