CAPÍTULO 7

DADOS ORGANIZADOS

7.1 Estruturas

Estrutura é um conjunto de variáveis, provavelmente de tipos diferentes, agrupadas e descritas por um único nome.

Por exemplo, numa folha de pagamento, desejamos guardar diversos registros (ou estruturas) representando funcionários. Cada funcionário, neste caso, possui alguns atributos, entre eles: nome (string), número do departamento (inteiro), salário (float), além de outros. Uma estrutura define um novo tipo composto, contendo diversos tipos básicos (ou compostos).

Para definir uma estrutura, definem-se os elementos dentro dela:

```
struct etiqueta
{
  declaração da variável membro
  declaração da variável membro
  ...
};
```

Po exemplo, a estrutura definida por:

```
struct Data
{
  int  dia;
  char mes[10];
  int  ano;
};
```

cria um novo tipo, chamado Data. Uma variável do tipo Data contém um dia (inteiro), um mês (string) e um ano (inteiro).

Para declarar variáveis do tipo Data, deve-se utilizar a palavra-chave **struct** antes do tipo. Por exemplo:

```
struct Data hoje;
```

declara uma variável de nome *hoje*, do tipo **struct** Data.

7.1.1 Acessando dados membro

Para acessar cada membro da estrutura separadamente, é utilizado o operador seleção (**.**). Por exemplo,

```
hoje.Ano = 2001;
hoje.Dia++;
strcpy(hoje.Mes, "Abril");
int iDia = hoje.Dia;
```

Cada membro da estrutura é equivalente a uma variável simples de seu tipo. O exemplo a seguir ilustra a utilização de estruturas.

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

void main(void)
{
  struct Data
  {
    int Dia;
    char Mes[10];
    int Ano;
  };

  struct Data hoje;

  hoje.Dia = 4;
  strcpy(hoje.Mes, "Outubro");
  hoje.Ano = 2001;
  printf("%d de %s de %d\n", hoje.Dia, hoje.Mes, hoje.Ano);
}
```

7.1.2 Estruturas dentro de estruturas

Estruturas podem conter outras estruturas em seu interior. Por exemplo, em uma agenda telefônica podemos guardar, além do telefone de cada pessoa, outros dados como o endereço e a data de nascimento.

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

void main(void)
{
  struct Data
  {
    int Dia;
    char Mes[10];
    int Ano;
  };

  struct Pessoa
  {
    char Nome[50];
    char Telefone[20];
    char Endereco[50];
    struct Data Nascimento;
  };

  struct Pessoa pessoa1;

  strcpy(pessoa1.Nome, "João");
  strcpy(pessoa1.Telefone, "2222-2222");
  strcpy(pessoa1.Endereco, "Av. Pres. Vargas, 10/1001");
  pessoa1.Nascimento.Dia = 15;
  strcpy(pessoa1.Nascimento.Mes, "Janeiro");
  pessoa1.Nascimento.Ano = 1980;
  printf("%s\n%s\n%s\n%d de %s de %d\n",
    pessoa1.Nome, pessoa1.Telefone, pessoa1.Endereco,
    pessoa1.Nascimento.Dia, pessoa1.Nascimento.Mes, pessoa1.Nascimento.Ano);
}
```

7.1.3 Atribuições entre estruturas

Na versão original do C é impossível atribuir o valor de uma variável estrutura a outra do mesmo tipo, usando uma simples expressão de atribuição. Seus dados membro tinham que ser atribuídos um a um.

Nas versões mais modernas de C, esta forma de atribuição já é possível. Isto é, se pessoa1 e pessoa2 são variáveis estrutura do mesmo tipo, a seguinte expressão pode ser usada:

```
pessoa1 = pessoa2;
```

7.1.4 Passando estruturas para funções

Versões mais recentes de C ANSI permitem que estruturas sejam passadas como argumento de funções. Como exemplo, vamos reescrever o exemplo chamando uma função que imprime os atributos de uma pessoa.

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

struct Data
{
  int Dia;
  char Mes[10];
  int Ano;
};

struct Pessoa
{
  char Nome[50];
  char Telefone[20];
  char Endereco[50];
  struct Data Nascimento;
};

void ImprimePessoa(struct Pessoa p)
{
  printf("%s\n%s\n%s\n%d de %s de %d\n",
    p.Nome, p.Telefone, p.Endereco,
    p.Nascimento.Dia, p.Nascimento.Mes, p.Nascimento.Ano);
}

void main(void)
{
  struct Pessoa pessoa1;

  strcpy(pessoa1.Nome, "João");
  strcpy(pessoa1.Telefone, "2222-2222");
  strcpy(pessoa1.Endereco, "Av. Pres. Vargas, 10/1001");
  pessoa1.Nascimento.Dia = 15;
  strcpy(pessoa1.Nascimento.Mes, "Janeiro");
  pessoa1.Nascimento.Ano = 1980;
  ImprimePessoa(pessoa1);
}
```

Note que, como mais de uma função do programa vai acessar as estruturas, elas são declaradas fora do escopo de qualquer função, o que as torna globais.

7.1.5 Vetores de estruturas

Assim como qualquer tipo básico, podemos ter vetores de estruturas. Estes vetores são definidos da mesma maneira que vetores de tipos básicos.

Vamos modificar o exemplo anterior para que trabalhemos com um vetor de pessoas.

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

struct Data
{
  int Dia;
  char Mes[10];
  int Ano;
};

struct Pessoa
{
  char Nome[50];
  char Telefone[20];
  char Endereco[50];
  struct Data Nascimento;
};

void ImprimePessoa(struct Pessoa p)
{
  printf("%s\n%s\n%s\n%d de %s de %d\n",
    p.Nome, p.Telefone, p.Endereco,
    p.Nascimento.Dia, p.Nascimento.Mes, p.Nascimento.Ano);
}

void main(void)
{
  struct Pessoa pessoa[100];

  strcpy(pessoa[0].Nome, "João");
  strcpy(pessoa[0].Telefone, "2222-2222");
  strcpy(pessoa[0].Endereco, "Av. Pres. Vargas, 10/1001");
  pessoa[0].Nascimento.Dia = 15;
  strcpy(pessoa[0].Nascimento.Mes, "Janeiro");
  pessoa[0].Nascimento.Ano = 1980;
  ImprimePessoa(pessoa[0]);
  strcpy(pessoa[1].Nome, "José");
  strcpy(pessoa[1].Telefone, "2222-2221");
  strcpy(pessoa[1].Endereco, "Av. Pres. Vargas, 10/1002");
  pessoa[1].Nascimento = pessoa[0].Nascimento;
  ImprimePessoa(pessoa[1]);
}
```

No exemplo acima, note que cada elemento do vetor é do tipo da estrutura Pessoa. O vetor é inicialmente dimensionado com 100 elementos, ou seja, podemos utilizar até 100 pessoas diferentes no vetor.

7.1.6 Ponteiros para estruturas

Ponteiros podem apontar para estruturas da mesma maneira que apontam para qualquer variável. Por exemplo, um ponteiro para a estrutura Data seria declarado como:

```
struct Data* pHoje = &hoje;
```

Para acessarmos os dados membro da estrutura através de seu ponteiro, podemos escrever

```
(*pHoje).Dia  /* Os parenteses sao necessarios */
```

ou podemos acessar os dados membro de um ponteiro através do operador (**->**)

```
pHoje->Dia
```

Vamos alterar a função ImprimePessoa, no exemplo anterior, para que receba o endereço da pessoa, ao invés de uma cópia. A vantagem de escrever a função desta maneira é o aumento da velocidade de processamento, já que é necessário copiar apenas o endereço, ao invés de todo o conteúdo da variável.

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

struct Data
{
  int Dia;
  char Mes[10];
  int Ano;
};

struct Pessoa
{
  char Nome[50];
  char Telefone[20];
  char Endereco[50];
  struct Data Nascimento;
};

void ImprimePessoa(struct Pessoa* p)
{
  printf("%s\n%s\n%s\n%d de %s de %d\n",
    p->Nome, p->Telefone, p->Endereco,
    p->Nascimento.Dia, p->Nascimento.Mes, p->Nascimento.Ano);
}

void main(void)
{
  struct Pessoa pessoa[100];

  strcpy(pessoa[0].Nome, "João");
  strcpy(pessoa[0].Telefone, "2222-2222");
  strcpy(pessoa[0].Endereco, "Av. Pres. Vargas, 10/1001");
  pessoa[0].Nascimento.Dia = 15;
  strcpy(pessoa[0].Nascimento.Mes, "Janeiro");
  pessoa[0].Nascimento.Ano = 1980;
  ImprimePessoa(&pessoa[0]);
  strcpy(pessoa[1].Nome, "José");
  strcpy(pessoa[1].Telefone, "2222-2221");
  strcpy(pessoa[1].Endereco, "Av. Pres. Vargas, 10/1002");
```

```
  pessoa[1].Nascimento = pessoa[0].Nascimento;
  ImprimePessoa(&pessoa[1]);
}
```

É recomendável que estruturas muito grandes sejam sempre passadas por endereço. No caso, a estrutura Pessoa ocupa 134 bytes (50 do nome, 20 do telefone, 50 do endereço e mais 14 da estrutura Data - 2 do dia, 10 do mês e mais 2 do ano). Passando o argumento por valor, o programa será obrigado a copiar 134 bytes, ao passo que passando por endereço, apenas os 4 bytes do ponteiro são copiados.

7.2 Uniões

Uma união permite que as mesmas localizações de memória sejam referenciadas de mais de um modo. Sua sintaxe é semelhante à da estrutura.

```
union etiqueta
{
  declaração da variável membro
  declaração da variável membro
  ...
};
```

A diferença é que a estrutura armazena todos os seus dados membro individualmente, enquanto a união utiliza o mesmo espaço de memória para armazenar todos os seus dados membro. Enquanto o espaço ocupado por uma estrutura é a soma do espaço utilizado por seus dados membro, a memória utilizada pela união é a memória ocupada por seu maior dado membro.

Po exemplo, a união definida por:

```
union Mes
{
  int  numero;
  char nome[10];
};
```

ocupa 10 bytes (relativos ao dado *nome*). Podemos acessar seus dados da mesma forma que na estrutura mas, uma vez alterado um dado membro, os outros são alterados também.

Ao trabalhar com uniões, deve-se acessar apenas um dos dados membro, dependendo do tipo de uso. Por exemplo, se definirmos

```
Mes mes;
strcpy(mes.nome, "Janeiro");
```

e tentarmos obter

```
mes.numero
```

o valor retornado é m valor sem sentido, já que o espaço de memória ocupado foi alterado por outra variável.

Não tendo problemas de falta de memória, deve-se optar pela utilização de estruturas ao invés de uniões.

7.3 Enumeração

A linguagem C apresenta um tipo de dado adicional, chamado enumeração. Sua sintaxe é:

```
enum identificacao {enum1, enum2 ...};
```

Por exemplo:

```
enum dias {segunda, terca, quarta, quinta, sexta, sabado, domingo};
```

especifica que *dias* é uma identificação para uma variável do tipo **enum**, que somente pode ter os valores de segunda a domingo.

Podemos declarar variáveis do tipo do enumerado acima, por exemplo:

```
enum dias dia;
```

os tipos de enumeração são representados internamente por inteiros. O primeiro identificador recebe o valor 0, o próximo 1 e assim sucessivamente. Ou seja, na enumeração *dias*, *segunda* vale 0, *terca* vale 1 e assim sucessivamente, até *domingo*, que vale 6.

Um enumerador pode ser declarado alterando os valores correspondentes a cada termo. Por exemplo, se declararmos

```
enum dias {segunda = 2, terca, quarta, quinta, sexta};
```

nesta enumeração, *segunda* vale 2, *terca* vale 3 e assim sucessivamente, até *sexta*, que vale 6. Se declararmos

```
enum dias {segunda = 2, quarta = 4, quinta = 5};
```

definimos separadamente os termos.