Capítulo 3

Tipos Abstratos de Dados

A linguagem C possui diversos tipos de dados nativos e disponíveis ao programador como int, float, double, long, char, entre outros. Um programador pode utilizar estes tipos de dados na forma de variáveis para representar informações em seus programas. Em algumas situações, porém, os tipos de dados nativos não representam adequadamente as informações que o programador precisa representar.

Tipos Abstratos de dados são tipos de dados que podem ser criados pelo próprio programador C para conseguir representar informações de uma forma mais direta que utilizando apenas os tipos nativos da linguagem C. Existem várias formas de criar tipos abstratos de dados, as quais serão estudadas em partes.

3.1 Enumeração

Uma enumeração em C é um tipo de dado que suporta apenas um conjunto finito de valores. A forma de uma enumeração é

```
1 enum diasemana {domingo, segunda, terca, quarta, quinta, sexta,sabado}; // definicao da enumeracao
2 enum mes {janeiro, fevereiro, marco, abril, maio, junho, julho, agosto, setembro, outubro, novembro,
      dezembro};
3 enum cor {branca, amarela, azul, verde, vermelha, preta};
```

Neste exemplo, criou-se três enumerações diasemana, mes e cor. As enumerações criadas podem ser utilizadas para criar variáveis que somente aceitem como valores os elementos que foram listados entre chaves. No exemplo abaixo, utilizou-se uma enumeração para criar uma variável e depois atribuir um determinado valor a esta variável. Mais tarde o valor da variável é testado para saber se vale sexta. Tanto sexta quanto segunda são valores válidos pois foram listados entre chaves na definição da enumeração. Internamente uma enumeração é representada por um número. Ou seja, no exemplo da primeira enumeração a palavra domingo vale a mesma coisa que 0 (valor padrão), segunda vale 1, terça 2 e assim por diante. Quando atribuimos um valor `Minha_Variavel = segunda` na realidade ela está recebendo o valor 1. Já na comparação, estamos comparando com o valor 5.

```
1 enum diasemana Minha_Variavel; // uso da enumeracao para criar uma variavel
2
3 Minha_Variavel = segunda;
4
5 if (Minha_Variavel==sexta)
6 {
7 }
8 else {
9 }
```

Perceba que para criar uma variável, é necessário utilizar a palavra `enum` seguinda do nome da enumeração criada e o nome da variável. Felizmente, existe uma forma de tornar o uso de uma enumeração semelhante ao de um tipo nativo da linguagem C, através do `typedef`.

```
1 enum diasemana {domingo, segunda, terca=5, quarta, quinta, sexta=22,sabado}; // definicao da
      enumeracao
2 enum mes {janeiro, fevereiro, marco, abril, maio, junho, julho, agosto, setembro, outubro, novembro,
      dezembro};
```

```
3 enum cor {branca, amarela, azul, verde, vermelha, preta};
4
5 typedef enum diasemana diasemana;
6
7 diasemana Minha_Variavel;
8
9 Minha_Variavel = segunda;
```

Os valores não precisam ser necessariamente sequenciais iniciando-se em 0. Podemos ajustar o valor de um dos dias da semana para ser digamos 5. Todos os elementos seguintes ao elemento que foi ajustado terão valores uma unidade maior e a sequencia continua. No exemplo, domingo vale 0, segunda vale 1, terça vale 5, quarta vale 6, quinta vale 7, sexta vale 22 e sábado 23.

3.2 Estruturas não homogêneas

As vezes precisamos trabalhar com um conjunto de informações como se fosse apenas uma. Vamos pensar os registros de pessoas numa agenda telefônica. Um registro da agenda descreve uma pessoa. O registro será composto tipicamente por nome, endereço e telefone.

Na linguagem C podemos criar um registro com a palavra reservada `struct`, tal como mostrado abaixo. Podemos criar variáveis utilizando posteriormente a palavra struct o nome da estrutura que foi criada pelo programador e o nome da variável. Agora temos uma variável chamada `Variavel_V1` que internamente possui 3 campos (nome, endereco e telefone) que podem ser acessados utilizando-se um ponto.

```
1 struct Registro_de_pessoa {
2    char nome[20];
3    char endereco[20];
4    int telefone;
5 };
6
7 struct Registro_de_pessoa Variavel_V1;
8
9 Variavel_V1.telefone=324567;
10 strcpy(Variavel_V1.nome, "Ana Maria");
11 strcpy(Variavel_V1.endereco, "getulio Vargas 1234");
12
13
14 printf("Mostra os dados completos\n");
15 printf("Nome=%s  Endereco=%s  Telefone= %d \n\n", Variavel_V1.nome, Variavel_V1.endereco, Variavel_V1.
     telefone);
```

No exemplo utilizamos o ponto para acessar o campo telefone e como telefone é um inteiro, conseguimos atribuir diretamente um valor para o mesmo. Já para o caso do campo nome e endereço que são strings, na linguagem C não é possível atribuir diretamente, precisamos utilizar uma função de copia de string `strcpy()` para este fim. Posteriormente, mostramos na tela os valores dos campos usando um `printf`.

A forma de declarar variáveis que são uma estrutura é semelhante ao enum no sentido de ser necessário escrever a palavra reservada novamente. Podemos adotar a mesma solução que antes e usar o typedef para criar um nome novo para o `struct Registro_de_pessoa`. Agora, a forma de declarar variáveis ficou semelhante ao uso dos tipos nativos da linguagem C

```
1 struct Registro_de_pessoa {
2    char nome[20];
3    char endereco[20];
4    int telefone;
5 };
6 typedef struct Registro_de_pessoa Registro_de_pessoa;
7
8 // tipo de dado   nome da variavel
9 Registro_de_pessoa Variavel_V1;
10 int               xxx;
11
12 Variavel_V1.telefone=324567;
13 strcpy(Variavel_V1.nome, "Ana Maria");
```

```
14 strcpy(Variavel_V1.endereco, "getulio Vargas 1234");
15
16
17 printf("Mostra os dados completos\n");
18 printf("Nome=%s  Endereco=%s  Telefone= %d \n\n", Variavel_V1.nome, Variavel_V1.endereco, Variavel_V1.
       telefone);
```

Ao criarmos os campos de uma struct podemos utilizar os tipos nativos da linguagem C e também outros tipos que tenham sido criados pelo programador. Veja o exemplo.

```
1
2  struct Tipo_Telefone {
3      int DDD;
4      int numero;
5  }
6  typedef struct Tipo_Telefone Tipo_Telefone;
7
8  struct Tipo_Endereco {
9      char rua[20]
10     int numero_da_casa;
11     char cidade[20];
12     char estado[3];
13 }
14 typedef struct Tipo_Endereco Tipo_Endereco;
15
16
17 struct Registro_de_pessoa {
18    char nome[20];
19    Tipo_Endereco endereco[20];
20    Tipo_Telefone telefone;
21 };
22 typedef struct Registro_de_pessoa Registro_de_pessoa;
23
24 // tipo de dado   nome da variavel
25 Registro_de_pessoa Variavel_V1;
26 int                xxx;
27
28 Variavel_V1.telefone.DDD=48;
29 Variavel_V1.telefone.numero=321234;
30
31 strcpy(Variavel_V1.nome, "Ana Maria");
32 strcpy(Variavel_V1.endereco.rua, "getulio Vargas");
33 Variavel_V1.endereco.numero_da_casa=1234;
34 strcpy(Variavel_V1.endereco.cidade, "Araranguа");
35 strcpy(Variavel_V1.endereco.ESTADO, "SC");
36
37
38 printf("Mostra os dados completos\n");
39 printf("Nome=%s  Endereco=%s %d Telefone= (%d)- %d \n\n", Variavel_V1.nome, Variavel_V1.endereco.rua,
       Variavel_V1.endereco.numero_da_casa,
40 Variavel_V1.telefone.DDD, Variavel_V1.telefone.numero);
```

3.2.1 Estruturas não homogêneas - typedef struct

Podemos combinar a o `struct` e o `typedef` vistos anteriormente para criar um novo tipo de dado, veja o exemplo:

```
1
2  typedef struct XXXX {
3     char Nome[5];
4     int Codigo;
5     char Sexo;
6     int c;
```

```
7     char Opcao;
8  } Dados_Cliente;
9
10 void main (void)
11 {
12    Dados_Cliente Cliente;
13
14    Cliente.Codigo=12345; Cliente.Sexo= F ;
15    printf("Codigo:%d\n",Cliente.Codigo);
16    printf("Sexo :%c\n",Cliente.Sexo);
17 }
```

Repare que XXXX pode ser qualquer coisa (é opcional), o que importa mesmo é o nome `Dados_Cliente` que aparece antes do ponto e vírgula.

3.2.2 Inicialização estática de estruturas

É possível inicializar uma estrutura no mesmo momento no qual uma variável é criada. No exemplo abaixo, `Dados_cliente` é composto por 2 campos. A variável cliente é criada como sendo do tipo `Dados_cliente` mas seu valor não é explicitado. Ja a variável Pedro é criada e o valor de seus dois campos é explicitado na mesma ordem em que foram declarados dentro do `struct`.

```
1  #include <stdio.h>
2  #include <string.h>
3
4  typedef int Meu_Inteiro;
5  typedef char String[100];
6
7  typedef struct {
8     char Nome[5];
9     int Codigo;
10 }Dados_Cliente;
11
12 void main (void)
13 {
14    Meu_Inteiro xx;
15    String a;
16    Dados_Cliente Cliente;
17    Dados_Cliente Pedro={"teste",123};
18 }
```

3.3 Uniões

Uniões são similares a `structs` no sentido de que podemos criar um registro composto por várias variáveis. Mas existe uma grande diferença entre registros e uniões em relação a como os campos ficam localizados na memória. Ao utilizar uniões, os campos que aparecem dentro da união ocuparão a mesma posição de memória. ex:

```
1  union {
2       float numero_float;
3       int   numero_inteiro;
4  }Tipo_Variavel;
5
6  void main () {
7     Tipo_Variavel x;
8     x.numero_float = 3.1415;
9     printf("%f\n",x.numero_float);
10    x.numero_inteiro = 45;
11    printf("%d\n",x.numero_inteiro);
12 }
```

3.4 Modificando campos de uma estrutura

Sabemos que para acessar um campo de uma estrutura é necessário usar o operador **.**. Ocorre que se tivermos uma função e o parâmetro desta função é uma estrutura, cuidados especiais devem ser tomados. Veja um exemplo que NÃO FUNCIONA.

```
1  #include <stdio.h>
2  struct Teste {
3    int codigo;
4    int CPF;
5  };
6  // O programador acredita que assim, modificara o campo CPF.
7  // Mas para modificar o parametro da funcao, este precisa ser passado por REFERENCIA
8  // e NAO por VALOR....
9  void muda_CPF ( struct Teste xx) {
10   xx.CPF = 100;
11 }
12 void main (void)
13 {
14     struct Teste T;
15
16     T.codigo=123;
17     T.CPF=34;
18     muda_CPF(T);
19     printf("CPF=%d\n",T.CPF);
20 }
```

Agora veja o próximo exemplo, onde o parâmetro da função muda_CPF é passado por referência (passa-se o endereço onde a variável está na memória). Para alterar um campo dessa estrutura, precisamos usar a notação **(*variável_estrutura).nome_do_campo=VALOR;** Como mostrado no próximo exemplo.

```
1  #include <stdio.h>
2  struct Teste {
3    int codigo;
4    int CPF;
5  };
6  void muda_CPF ( struct Teste *xx) {
7    (*xx).CPF= 100;
8  }
9  void main (void)
10 {
11     struct Teste T;
12
13     T.codigo=123;
14     T.CPF=34;
15     muda_CPF(&T);
16     printf("CPF=%d\n",T.CPF);
17 }
```

A notação `(*xx).CPF` assusta algumas pessoas, por isso, existe uma notação equivalente `xx->CPF`, ou seja, ao invés de `.` usa-se `->` (seta). Em resumo: Quando você precisa alterar campos de uma estrutura DENTRO DE UMA FUNÇÃO a estrutura deve ser passada por REFERÊNCIA e os campos devem ser acessados usando o `->`.

```
1  #include <stdio.h>
2  struct Teste {
3    int codigo;
4    int CPF;
5  };
6  void muda_CPF ( struct Teste *xx) {
7    xx->CPF= 100;
8    // Aqui usa o operador seta para modificar o campo CPF
9  }
10 void main (void)
```

```
{
    struct Teste T;

    // coloca algum conteudo dentro da estrutura, usa o ponto pois aqui so apenas os campos de uma
        varivel
    T.codigo=123;
    T.CPF=34;
    muda_CPF(&T); // Neste ponto o programador deseja alterar a estrutura, entao deve passa-la por
        referencia
    printf("CPF=%d\n",T.CPF);
}
```

3.5 Vetores de estruturas

Podemos criar vetores de estruturas onde cada uma das posições do vetor será uma estrutura composta por campos. Abaixo, mostra-se como criar um vetor estático de 100 posições onde cada uma destas posições terá os campos código e CPF.

```
#include <stdio.h>
struct Teste {
  int codigo;
  int CPF;
};
typedef struct Teste Teste;
Teste vetor[100];

void main (void) {
  vetor[0].CPF=100;
  vetor[0].codigo=1212;
  vetor[1].CPF=1233;
  vetor[1].codigo=345;
}
```

3.5.1 Função com vetores de estruturas como parâmetros

Quando utilizamos vetores de estruturas como parâmetros, por ser um vetor, seu nome é o endereço onde o mesmo está na memória. Assim, internamente na função podemos modificar o vetor pois é uma passagem de parâmetro por referência (mesmo que não sejam utilizados os símbolos & e asterisco).

```
#include <stdio.h>
struct Teste {
  int codigo;
  int CPF;
};
typedef struct Teste Teste;
Teste vetor[100]; int contador=0; // contador comeca em zero

void insere_dados (Teste v[], int codigo, int CPF) {
  v[contador].codigo=codigo; // usa uma variavel GLOBAL contador para controlar
  v[contador].CPF=CPF ;      // a posicao onde deve colocar os dados
  contador++;                // na proxima vez insere na proxima posicao
}
void mostra_todos (Teste v[]) {
   int x;
   for (x=0;x<contador;x++) {
      printf("%d %d\n",v[x].codigo, v[x].CPF);
   }
}
void main (void) {
  insere_dados(vetor, 100, 123);
  insere_dados(vetor, 200, 456);
```

```
23   insere_dados(vetor, 300, 789);
24   insere_dados(vetor, 400, 98);
25   mostra_todos (vetor);
26 }
```

3.5.2 Vetores de estruturas

O exemplo anterior apresentou um vetor de estruturas e uma variável global para controlar a posição que o mesmo deve ser acessado. Ocorre que usar variável global não é uma boa forma de manter o código bem organizado e gerenciado, ainda mais se tivermos vários vetores de estruturas para controlar.

Uma forma mais adequada seria "Empacotar"a variável contadora dentro de uma estrutura

```
1  #include <stdio.h>
2  struct Teste {
3    int codigo;
4    int CPF;
5  };
6  typedef struct Teste Teste;
7  struct Minha_Estrutura {
8    Teste vetor[100];
9    int contador;
10 };
11 typedef struct Minha_Estrutura Minha_Estrutura;
12 void insere_dados (Minha_Estrutura *v, int codigo, int CPF) {
13   v->vetor[ v->contador ].codigo = codigo;
14   v->vetor[ v->contador ].CPF = CPF;
15   v->contador++;
16 }
17 void mostra_todos (Minha_Estrutura v) {
18    int x;      for (x=0;x<v.contador;x++) printf("%d %d \n",v.vetor[x].codigo, v.vetor[x].CPF);
19 }
20 int main (void) {
21   Minha_Estrutura M; // N e' a variavel do tipo estrutura que tem um vetor e um contador;
22   M.contador = 0; // inicializa-se o contador para marcar zero
23   insere_dados (&M, 100, 200); // usa-se & pois agora M nao e' um vetor como no exemplo anterior
24   insere_dados (&M, 100, 300);
25   mostra_todos (M);
26 }
```