10. - ESTRUTURAS

10.1 - Características Básicas

Coleção de uma ou mais variáveis que podem ser de diferentes tipos, agrupadas sob um mesmo nome. Também chamadas de registros.

Exemplo típico:

Folha de pagamento, cada elemento é um funcionário.

Funcionário é descrito através de um conjunto de atributos:

nome → string
endereço → string
CPF → string
código do depto. → char
salário → float
tempo de serviço → int (em meses)
etc

Sintaxe:

```
struct  < nome >  {            /*   definição da estrutura   */
        tipo1  < var1 >;
        tipo2  < var2 >;
        .
        .
        };
```

struct < nome > var;

↓ ↓

novo tipo declaração da variável do novo tipo

var é uma variável do tipo struct <nome> e tem vários campos:

var1, var2, etc ..., que são acessados da seguinte maneira:

var.var1
var.var2
↓ ↓
variável campo da variável

Exemplo 1:

```
struct ponto {
        int x;              /*   ou   int x, y;   */
        int y;
        };
struct ponto P;
struct ponto M;         /*  ou   struct ponto P, M;   */
```

Podemos fazer:

P.x = 2;
P.y = 3;

ou ainda

scanf (“% d % d”, &P.x, &P.y);

Exemplo 2: Definição e declaração podem ser feitas juntas.

```
struct facil                    /*   declaração do tipo   */
        {   int  num;
            char  ch;           /*   define facil   */
        } x1, x2;               /*   declaração das variáveis x1 e x2 */
```

- podemos fazer:

```
        x1.num = 13;
        x1.ch = 'c';
        x1.ch++;
```

- Como x1 e x2 são do mesmo tipo, podemos fazer a seguinte atribuição:

```
x2 = x1;
printf ("% d   % c \n", x2.num, x2.ch);
        /* resulta:  13  d */
```

Este tipo de atribuição não pode ser usado com matrizes e vetores.

- Caso o novo tipo criado seja usado só uma vez, podemos "omitir" seu nome (deixá-lo sem nome):

```
struct {
        int num;
        char ch;
        } x1, x2;           /*   variáveis com 2 campos: num e ch */
```

Exemplo 3:

```
struct livro {
        char titulo [30];
        int registro;
        char estilo [10];
        }
struct livro livro1;            /*   declaração   */
```

Algumas operações:

```
printf (“\n Digite título: “);
        gets (livro1.titulo);
printf (“\n Digite registro: “);
        scanf (“%d”, &livro1.registro);
printf (“\n Digite estilo: “);
        scanf (“%s”, livro1.estilo);
```

```
struct livro livro2 = {“Iracema”, 102, “romance”};
                        /*   declaração de livro2 e atribuição   */
```

Exemplo 4: Podemos usar estruturas dentro de estruturas.

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

struct retangulo {
        struct ponto pt1;
        struct ponto pt2;
        }

struct retangulo R1, R2;
```

R1 e R2 têm 4 campos cada um:
R1.pt1.x, R1.pt1.y, R1.pt2.x, R1.pt2.y
idem para R2.

Supor que pt1 e pt2 sejam vértices opostos de um retângulo.

Leitura dos vértices:

```
printf ("Leitura do 1o. Retângulo: \n");
printf ("Digite vértice 1: \n");
scanf ("%d   %d", &R1.pt1.x, &R1.pt1.y);
printf ("Digite vértice 2: \n");
scanf ("%d   %d", &R1.pt2.x, &R1.pt2.y);
```

Cálculo da área de R1:

```
printf ("\n Area de R1 = %d \n",
            (R1.pt2.x - R1.pt1.x) * (R1.pt2.y - R1.pt1.y));
        /*                  Δx              Δy                  */
```

Obs: Estruturas não podem ser comparadas.

```
Ex:   if (R1 = = R2)
        .
        .
```

ERRADO!!!

Exemplo 5: Definindo um tipo struct com typedef

```
typedef  struct  ponto  {      /*  definição de Ponto  */
          int x, y;
        }
ponto  P, M;                          /*  declarações  */
```

(Não precisa escrever struct antes. Sintaxe mais simples. É bom quando várias declarações de variáveis desse tipo são feitas!)

Exemplo 6:

```
typedef  struct  ponto        /*  definição de tipo  */
        {  int x, y;
        } P, M;               /*  declaração de variáveis  */
ponto S;                      /*  declaração de variável  */
```

Exemplo 7:

```
typedef  struct  {
        int x;
        int y;
        } ponto;          /*  definição de tipo: nome no final  */
```

Obs: Se não tivesse a palavra typedef , Ponto seria variável!!!

```
ponto P, M, S;    /*  declaração de variáveis do tipo Ponto */
```

Exemplo 8: Valores de estruturas podem ser passados como argumentos para funções, e podem ser retornados. Estruturas não são como vetores! Elas são passadas como valor.

```
#define TAM 50
struct endereco {
   char rua[TAM];
   char cidade[TAM];
};

struct endereco obtem_endereco(void);
void imprime_endereco(struct endereco);

struct endereco obtem_endereco(void)
{
    struct endereco e;
    printf("\t Entre rua: ");
    gets(e.rua);
    printf("\t Entre cidade: ");
    gets(e.cidade);
    return e;
}

void imprime_endereco(struct endereco e)
{
    printf("\t %s\n", e.rua);
    printf("\t %s\n", e.cidade);
}

main()
{   struct endereco residencia;

    printf("Entre seu endereco residencial:\n");
    residencia = obtem_endereco();

    printf("\nSeu endereco eh:\n");
    imprime_endereco(residencia);
}
```

Exemplo 9: Arrays de estruturas.

```
#define TAM 50
#define NUM 10
struct endereco {
        char rua[TAM];
        char cidade[TAM];
    };

void obtem_endereco(struct endereco [], int);
void imprime_endereco(struct endereco);

void obtem_endereco(struct endereco ae [], int index)
{   printf("Entre rua: ");
    gets(ae[index].rua);
         /* acesso ao campo rua da estrutura na posicao index */
    printf("Entre cidade: ");
    gets(ae[index].cidade);
 }

 void imprime_endereco(struct endereco e)
 {   printf("%s\n", e.rua);
     printf("%s\n", e.cidade);
  }

main()
{
  struct endereco residencias[NUM];
  int i;

  for (i = 0; i < NUM; i++)
  {   printf("Entre o endereco da pessoa %d:\n", i);
      obtem_endereco(residencias,i);
  }

  for (i = 0; i < NUM; i++)
  {   printf("endereco da pessoa %d:\n", i);
      imprime_endereco(residencias[i]);
  }
}
```

10.3 - Ponteiros para Estruturas

Podemos ter ponteiros para estruturas e acessar os campos da estrutura através desses ponteiros.

Exemplo 1:

```
typedef  struct  Ponto
        {  int x;
           int y;
        };
Ponto P, *ap;          /*   P: variável do tipo Ponto
                      *ap : var. do tipo ponteiro para Ponto */


int a, b;
scanf ("%d   %d", &a, &b);
ap = &P;
ap → x = a;                    /*   igual  a   P.x = a;   */
ap → y = b;                    /*   igual  a  P.y = b;   */
```

Temos então, neste caso, duas maneiras de acessar os elementos da estrutura.

Obs: ap é igual a &P, que é igual a &(ap → x), que é igual a &(P.x)

(endereço do 1o. elemento).

Um ponteiro para uma estrutura também pode ser declarado da seguinte forma:

```
struct {
        int x;
        int y;
        } *ap;        /*   tipo sem nome,  ap é pont. p/ este tipo   */
```

Exemplo 2:

```
struct { int tam;
         char *cad;
       } *p, S;
char aux;
```

Executando alguns comandos:

. e → têm a mesma prioridade equivalente a de ( ) e [ ]

```
p = &S;
scanf ("%d   %c", &(p → tam), &aux);

p→ cad = &aux;
printf ("\n %c",  *(p→ cad));

+ + p →  tam;          /* eq. a   + + S.tam   */
*p → cad = 'f';        /*  eq. a  aux = 'f'  */
```

Exemplo 3: Ponteiros para estruturas como argumentos de funções

```
#define TAM 50
struct endereco {
        char rua[TAM];
        char cidade[TAM];
 };

void obtem_endereco(struct endereco *);
void imprime_endereco(struct endereco);

void obtem_endereco(struct endereco *p_e)
{
   printf("Entre rua: ");
   gets(p_e->rua);
   printf("Entre cidade: ");
   gets(p_e->cidade);
 }

 void imprime_endereco(struct endereco e)
 {
    printf("%s\n", e.rua);
    printf("%s\n", e.cidade);
  }

  main()
  {
     struct endereco residencia;
     printf("Entre seu endereco residencial:\n");
     obtem_endereco(&residencia);
     printf("\nSeu endereco:\n");
      imprime_endereco(residencia);
   }
```

Não é necessário passar um ponteiro para a estrutura se seu valor não será mudado. De um modo geral, é melhor passar ponteiros para estruturas ao invés de passar e retornar valores de estruturas. Embora as duas abordagens sejam equivalentes e funcionem, o programa irá apenas passar ponteiros ao invés de toda uma estrutura que pode ser particularmente grande, implicando em um tempo final de processamento maior.

Veja! Arrays e estruturas são diferentes.