

## Structs e Ponteiros

### Estruturas (struct)

Já conhecemos os tipos simples do C: char, int, short, float, double, ...

É possível construir-se um tipo composto de elementos destes tipos simples.

Estrutura (**struct**) é uma coleção de uma ou mais variáveis, possivelmente de tipos diferentes, que podem ser manipuladas em conjunto ou em separado. Alguns exemplos:

- 1) Variáveis x, y, z do tipo estrutura para conter os dados de funcionários da empresa:

```
struct {int numero;
        char nome[40];
        int cpf;
        double salario;
} x, y, z;
```

- 2) Variáveis d1 e d2 do tipo estrutura para conter uma data:

```
struct {int dia;
        int mes;
        int ano;
} d1, d2;
```

- 3) Variável pt do tipo estrutura para conter as coordenadas de um ponto no plano:

```
struct {double x;
        double y;
} pt;
```

### Atribuindo um nome a estrutura

Podemos dar um nome à estrutura e referenciar esse nome na declaração das variáveis:

```
struct funcionario {int numero;
                   char nome[40];
                   int cpf;
                   double salario;
};
/* declara as variáveis x, y e z do tipo funcionário */
struct funcionario x, y, z;

struct data {int dia;
            int mes;
            int ano;
};
/* declara as variáveis d1 e d2 do tipo data */
struct data d1, d2;
```

```
struct ponto {double x;  
              double y;  
};  
/* declara a variável pt do tipo ponto */  
struct ponto pt;
```

### **Outros exemplos:**

Dados de um produto:

```
struct produto {  
    int codigo;  
    char nome[40];  
    double custo;  
    double preco;  
    int quantidade;  
}  
  
/* variáveis do tipo produto */  
struct produto P1,P2;
```

Dados de um livro:

```
struct livro {  
    int codigo;  
    char titulo[60];  
    char autor[40];  
    char editora[30];  
    int ano;  
    int num_exemplares;  
}  
  
/* variáveis do tipo livro */  
struct produto livro1,livro2,livro3;
```

### **Como usar e como referenciar os elementos internos das estruturas?**

Alguns exemplos usando as declarações dos elementos acima:

```
/* uso da variáveis do tipo struct */  
x = y;  
d1 = d2;
```

Como nos exemplos acima, é possível atribuir uma estrutura á outra, desde que sejam do mesmo tipo. Porém, não é possível comparar duas estruturas. Não são permitidos comando do tipo:

```
if (x == y) ...  
while (x > y) ...
```

Para fazer a comparação é necessário o acesso aos elementos internos da estrutura e fazer a comparação campo a campo.

Para se referenciar um dos campos internos dentro de uma variável do tipo estrutura usa-se o operador ponto (.) da seguinte forma:

**<variável>.<campo>**

Alguns exemplos:

```
/* usar os elementos internos das variáveis do tipo struct */  
x.numero = 12345;  
x.salario = 1035.33;  
x.salario = y.salario;  
if (x.cpf == 8888888880) ....  
d1.dia = 1;  
d1.mes = 1;  
d1.ano = 2001;  
if (d1.ano > d2.ano) ...  
pt.x = 2.34;  
pt.y = -56.48;
```

## Vetores de estruturas

A declaração:

```
struct funcionario tabfunc[100];
```

Declara um vetor de 100 elementos, em que cada elemento é uma estrutura do tipo funcionario.

Os seguintes comandos são exemplos de acesso aos elementos de tabfunc:

```
tabfunc[i].numero = 235467;  
tabfunc[k].salario = 333.44;  
  
for (i = 0; i < max_func; i++) {  
    tabfunc[i].numero = -1;  
    tabfunc[i].cpf = 0;  
    tabfunc[i].salario = 0;  
    for (j = 0; j < 40; j++) tabfunc[i].nome[j] = ' ';  
}
```

## Ponteiros e estruturas – O operador ponto (.)

Da mesma forma que outras variáveis, podemos ter ponteiros para estruturas.

```
struct funcionario x, *pf;  
struct data d, *pd;
```

```
pf = &x;  
pd = &d;
```

```
(*pf).numero = 123456; é o mesmo que x.numero = 123456;  
(*pd).ano = 2001; é o mesmo que d.ano = 2001;
```

O motivo do parêntesis em `(*pf)` é a prioridade dos operadores `*` (estrela) e `.` (ponto).

## O operador ->

Outra forma de usar ponteiro com estrutura é usar a abreviatura `p->` para `(*p)`.

Os comandos acima ficariam:

```
pf->numero = 123456; é o mesmo que x.numero = 123456;  
pd->ano = 2001; é o mesmo que d.ano = 2001;
```

A notação `p->` é mais clara e um pouco menos carregada que a notação `(*p)`.  
Também um pouco mais intuitiva.

## Estruturas e alocação dinâmica de memória

Da mesma forma que outras variáveis a alocação de espaço em memória para estruturas pode ser feito dinamicamente com `malloc` e `free`. Isso pode ser especialmente interessante no caso de estruturas, onde cada elemento pode ocupar uma região grande de memória. Considere por exemplo a estrutura `funcionario` acima e a tabela `tabfunc`. Vamos alocá-la com `n` elementos:

```
#include <stdlib.h>  
#include <stdio.h>  
int main(){  
    int i, j, n;  
    struct funcionario {int numero;  
                        char nome[40];  
                        int cpf;  
                        double salario;  
    };  
    struct funcionario *tabfunc;
```

```
scanf("%d", &n); /* le n */
/* aloca elementos para tabfunc */
tabfunc = (struct funcionario *) malloc(n * sizeof(struct
funcionario));
/* inicia os n elementos de tabfunc */
for (i = 0; i < n; i++){
    tabfunc[i].numero = 0;
    tabfunc[i].cpf = 99999999999;
    tabfunc[i].salario = -999999.99;
    for (j = 0; j < 40; j++) tabfunc[i].nome[j] = 'x';
}

/* outra forma de iniciar os n elementos de tabfunc */
for (i = 0; i < n; i++){
    (tabfunc + i) -> numero = 0;
    (tabfunc + i) -> cpf = 99999999999;
    (tabfunc + i) -> salario = -999999.99;
    for (j = 0; j < 40; j++) (tabfunc + i) -> nome[j] = 'x';
}

/* outra forma de iniciar os n elementos de tabfunc */
for (i = 0; i < n; i++){
    (*(tabfunc + i)).numero = 0;
    (*(tabfunc + i)).cpf = 99999999999;
    (*(tabfunc + i)).salario = -999999.99;
    for (j = 0; j < 40; j++) (*(tabfunc + i)).nome[j] = 'x';
}

/* usa os n elementos de tabfunc */
...
...
/* libera os n elementos de tabfunc */
free(tabfunc);
}
```

## Structs e tipo de funções

Uma função pode ter como tipo uma struct.

Isso até resolve o problema de uma função poder devolver em seu retorno (comando return) um só valor. Se ela for do tipo struct, pode devolver todos os valores internos à struct.

Suponha uma função que calcule e devolva as raízes de uma equação do segundo grau: Podemos fazer uma estrutura contendo as duas raízes e usar esta estrutura como tipo da função:

```
struct raizes {
    double x1,x2;
}

struct raizes calculo (double a, double b, double c) {
    /* calcula e devolve as raizes */
    struct raizes r;
    double delta;
    delta=b*b-4*a*c;
    r.x1 = (-b+sqrt(delta))/(2*a);
    r.x2 = (-b-sqrt(delta))/(2*a);
    return r;
}
```

Veja agora o exemplo abaixo. Os pontos no plano são definidos por um par de coordenadas (x, y). Vamos definir uma estrutura struct ponto para este novo tipo de dado e usá-la em algumas funções para manipular pontos.

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

struct ponto {
    double x;
    double y;
};

struct ponto define_ponto (double a, double b) {
    struct ponto t;
    t.x = a; t.y = b;
    return t;
}

struct ponto soma_pontos (struct ponto u, struct ponto v) {
    struct ponto t;
    t.x = u.x+v.x;
    t.y = u.y+v.y;
    return t;
}

double distancia(struct ponto u, struct ponto v) {
    return sqrt((u.x-v.x)*(u.x-v.x)+(u.y-v.y)*(u.y-v.y));
}

int main() {
    struct ponto a,b,c;
    a = define_ponto(0.0, 0.0);
```

```
    b = define_ponto(2.0, 2.0);  
    c = soma_pontos(a,b);  
    printf("\nsoma = (%5.11f,%5.11f)", c.x, c.y);  
    printf("\ndistancia = %5.11f", distancia(a,b));  
}
```

### Exercícios:

- 1) Um número complexo  $a+bi$  também é um par  $(a, b)$ . Defina uma struct complexo e faça as seguintes funções para manipular complexos:

```
struct complexo soma(struct complexo c1,struct complexo c2){  
/* devolve a some de c1 com c2 */
```

```
struct complexo mult(struct complexo c1,struct complexo c2){  
/* devolve o produto dos complexos c1 e c2 */
```

```
double modulo (struct complexo c) {  
/* devolve o modulo do complexo c */
```

- 2) Refaça o exemplo da função que calcula as raízes da equação do segundo grau, tratando os casos particulares, devolvendo dentro da struct um campo a mais como código de retorno:

```
struct raizes {  
    int retorno;  
    /* devolve -1 se delta < 0 - não há raízes reais */  
    /* devolve 0 se a = 0 - não é do 2 grau */  
    /* devolve 1 se delta = 0 - 2 raízes iguais */  
    /* devolve 2 se delta > 0 - 2 raízes diferentes */  
    double x1,x2;  
}
```