Programação em computadores II

Estruturas - material extra

Prof. Dr. Fábio Rodrigues de la Rocha

Definição de uma Estrutura em C

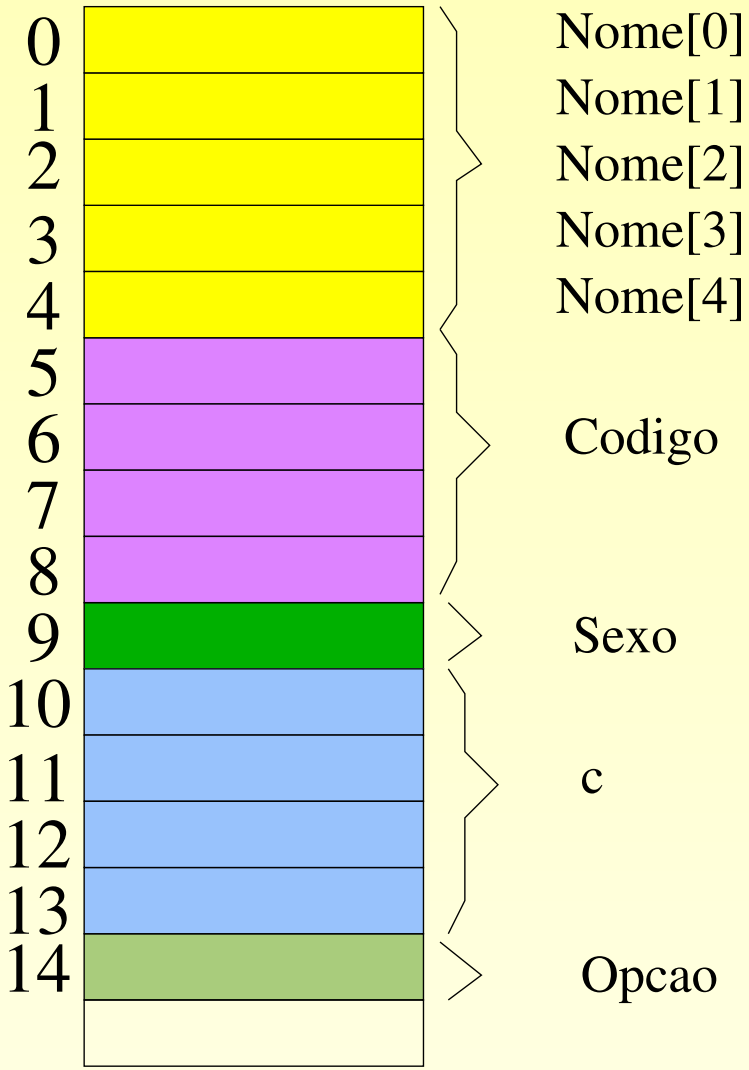
```
1 struct Dados_Cliente {
2       char Nome[5];
3       int Codigo;
4       char Sexo;
5       int c;
6       char Opcao;
7 }XXXX;
```

Repare que XXXX pode ser qualquer coisa (e é opcional), o que importa mesmo é o nome que aparece após a palavra `struct`.

Exemplo completo

```
1  #include <stdio.h>
2  #include <string.h>
3
4  struct Dados_Cliente {
5       char Nome[5];
6       int Codigo;
7       char Sexo;
8       int c;
9       char Opcao;
10 };
11 void main (void) {
12      struct Dados_Cliente Cliente;
13      Cliente.Codigo=12345; Cliente.Sexo='F';
14      printf("Codigo:%d\n",Cliente.Codigo);
15      printf("Sexo :%c\n",Cliente.Sexo);
16 }
```

Estrutura na memória



Definição de tipos

Define-se:

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

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.

Exemplo completo

```
1  #include <stdio.h>
2  #include <string.h>
3
4  typedef struct XXXX {
5       char Nome[5];
6       int Codigo;
7       char Sexo;
8       int c;
9       char Opcao;
10 }Dados_Cliente;
11 void main (void) {
12      Dados_Cliente Cliente;
13      Cliente.Codigo=12345; Cliente.Sexo='F';
14      printf("Codigo:%d\n",Cliente.Codigo);
15      printf("Sexo :%c\n",Cliente.Sexo);
16 }
```

Outro exemplo

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

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.

Uniões

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 }
```

Uniões

Mas quando utilizarei Uniões ?

Estude os exemplos seguintes:

Registro 1

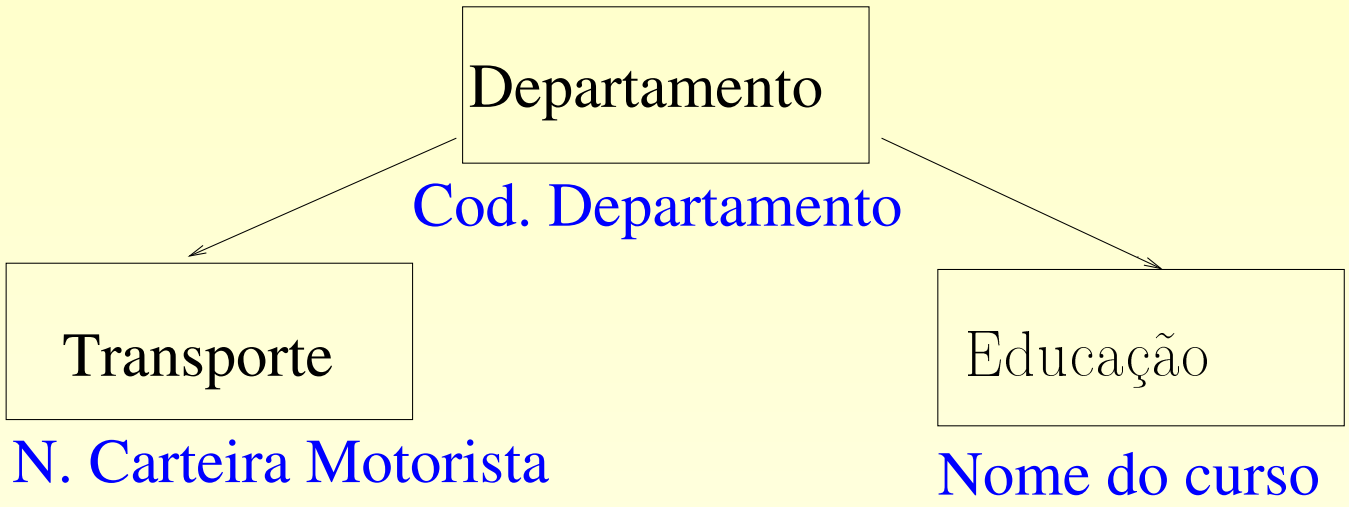
Uma empresa tem um conjunto de funcionários e deseja cadastrá-los num computador. Os dados relevantes são: Nome, endereço e CPF.

```
1 typedef struct {
2       char nome[20];
3       char endereco[20];
4       long CPF;
5
6 }Tipo_funcionario;
```

Sabendo que um `long` tem 8 bytes, realizando a soma temos que o `Tipo_funcionario` ocupa 48 bytes de memória. Compare com o próximo exemplo.

Registro 2

Suponha que uma universidade tenha um cadastro de seus funcionarios. Se o funcionario em questão trabalha no departamento de transportes da universidade, ele deve fornecer seu número de carteira de motorista. Se o funcionário da universidade é da área de educação, deve informar o nome do curso que trabalha.



Se utilizarmos a mesma idéia do Registro 1 e o criarmos utilizando um `typedef struct` teremos:

```
1 typedef struct {
2       int Cod_Dept;
3       long Numero_carteira;
4       char Nome_do_curso[10];
5
6 }Tipo_funcionario;
```

Só que desta forma, como os funcionários requerem campos diferentes nos registros, acabamos por desperdiçar memória fazendo uma soma nos campos dos dois tipos de funcionários.

Utilizando unions para resolver o problema

```
1  typedef struct {
2       int Cod_Dept;
3       union {
4          long Numero_carteira;
5          char Nome_curso[10];
6       }Dados;
7  }Tipo_funcionario;
8    Tipo_funcionario vet[100];
9    // Se o funcionario motorista
10   vet[0].Cod_Dept=10;
11   vet[0].Dados.Numero_carteira=123;
12   // se o funcionario professor
13   vet[1].Cod_Dept=20;
14   strcpy(vet[1].Dados.Nome_curso,"letras");
```

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.

Modificando campos de uma estrutura.

```
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 }
```

Modificando campos de uma estrutura.

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 mostrada no exemplo da próxima página.

```c
#include <stdio.h>
struct Teste {
  int codigo;
  int CPF;
};
void muda_CPF ( struct Teste *xx) {
  (*xx).CPF= 100;
}
void main (void)
{
     struct Teste T;

     T.codigo=123;
     T.CPF=34;
     muda_CPF(&T);
     printf("CPF=%d\n",T.CPF);
}
```

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 `->`.

```c
#include <stdio.h>
struct Teste {
  int codigo;
  int CPF;
};
void muda_CPF ( struct Teste *xx) {
  xx->CPF= 100;
  // Aqui usa o operador seta para modificar o campo CPF
}
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);
}
```

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.

```
1  #include <stdio.h>
2  struct Teste {
3    int codigo;
4    int CPF;
5  };
6  typedef struct Teste Teste;
7  Teste vetor[100];
8
9  void main (void) {
10   vetor[0].CPF=100;
11   vetor[0].codigo=1212;
12   vetor[1].CPF=1233;
13   vetor[1].codigo=345;
14 }
```

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).

```
1  #include <stdio.h>
2  struct Teste {
3    int codigo;
4    int CPF;
5  };
6  typedef struct Teste Teste;
7  Teste vetor[100]; int contador=0; // contador comeca em zero
8
9  void insere_dados (Teste v[], int codigo, int CPF) {
10   v[contador].codigo=codigo; // usa uma variavel GLOBAL contador para controlar
11   v[contador].CPF=CPF ;     // a posicao onde deve colocar os dados
12   contador++;               // na proxima vez insere na proxima posicao
13 }
```

```c
#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);
  insere_dados(vetor, 300, 789);
  insere_dados(vetor, 400, 98);
  mostra_todos (vetor);
}
```

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

```c
#include <stdio.h>
struct Teste {
  int codigo;
  int CPF;
};
typedef struct Teste Teste;
struct Minha_Estrutura {
  Teste vetor[100];
  int contador;
};
typedef struct Minha_Estrutura Minha_Estrutura;
void insere_dados (Minha_Estrutura *v, int codigo, int CPF) {
  v->vetor[ v->contador ].codigo = codigo;
  v->vetor[ v->contador ].CPF = CPF;
  v->contador++;
}
void mostra_todos (Minha_Estrutura v) {
   int x;      for (x=0;x<v.contador;x++) printf("%d %d \n",v.vetor[x].codigo, v.vetor[x].CPF);
}
int main (void) {
  Minha_Estrutura M; // N e' a variavel do tipo estrutura que tem um vetor e um contador;
  M.contador = 0; // inicializa-se o contador para marcar zero
  insere_dados (&M, 100, 200); // usa-se & pois agora M nao e' um vetor como no exemplo anterior
  insere_dados (&M, 100, 300);
  mostra_todos (M);
}
```

Podemos melhorar ainda mais o código anterior para fazer com que o vetor não seja estático mas que possa ser criado dinamicamente.

```c
 #include <stdio.h>
#include <stdlib.h>
struct Teste {
  int codigo;
  int CPF; };
typedef struct Teste Teste;
struct Minha_Estrutura {
  Teste *vetor;
  int contador;
};
typedef struct Minha_Estrutura Minha_Estrutura;
void insere_dados (Minha_Estrutura *v, int codigo, int CPF) {
  v->vetor[ v->contador ].codigo = codigo;
  v->vetor[ v->contador ].CPF = CPF;
  v->contador++;
}
void mostra_todos (Minha_Estrutura v) {
   int x;      for (x=0;x<v.contador;x++) printf("%d %d \n",v.vetor[x].codigo, v.vetor[x].CPF);
}
void inicializa (Minha_Estrutura *v, int tamanho) {
   v->contador=0; // inicializa o contador
   v->vetor = (Teste *) malloc (sizeof (Teste)*tamanho);
}
int main (void) {
  Minha_Estrutura M; // N e' a variavel do tipo estrutura que tem um vetor e um contador;
  inicializa (&M, 100);
  insere_dados (&M, 100, 200); // usa-se & pois agora M nao e' um vetor como no exemplo anterior
  insere_dados (&M, 100, 300);
  mostra_todos (M);
}
```