Struct e Uniões

Prof. Marcelo Costa, MSc
marcelo.costa@uva.edu.br

Agenda

- Como funcionam Structs
- Declarando e utilizando ponteiros
- Ponteiros e Vetores

Definição de Structs

- As estruturas de dados consistem em criar apenas um dado que contém vários membros, que nada mais são do que outras variáveis.
- De uma forma mais simples, é como se uma variável tivesse outras variáveis dentro dela.
- A vantagem em se usar estruturas de dados é que podemos agrupar de forma organizada vários tipos de dados diferentes, por exemplo, dentro de uma estrutura de dados podemos ter juntos tanto um tipo float, um inteiro, um char ou um double.
- As variáveis que ficam dentro da estrutura de dados são chamadas de membros.

Criando uma estrutura de dados com STRUCT

- No seguinte exemplo declara um registro dma com três campos que podem ser usados para armazenar datas:

```
struct dma {
    int dia;
    int mes;
    int ano;
};
struct dma x;  /* um registro x do tipo dma */
struct dma y;  /* um registro y do tipo dma */
```

Como referenciar um Struct

- Para se referir a um campo de um registro, basta escrever o nome do registro e o nome do campo separados por um ponto:
- x.dia = 31;
- x.mes = 12;
- x.ano = 2015;

Exemplo com tipo de variável struct

```
#include <stdio.h>

#include <string.h>

struct tipo_endereco

{
    char rua [50];
    int numero;
    char bairro [20];
    char cidade [30];
    char sigla_estado [3];
    long int CEP;

};

struct ficha_pessoal

{
    char nome [50];
    long int telefone;
    struct tipo_endereco endereco;

};
```

```
main (void)
{
    struct ficha_pessoal ficha;
    strcpy (ficha.nome,"Luiz Osvaldo Silva");
    ficha.telefone=4921234;
    strcpy (ficha.endereco.rua,"Rua das Flores");
    ficha.endereco.numero=10;
    strcpy (ficha.endereco.bairro,"Cidade Velha");
    strcpy (ficha.endereco.cidade,"Belo Horizonte");
    strcpy (ficha.endereco.sigla_estado,"MG");
    ficha.endereco.CEP=31340230;
    return 0;
}
```

Explicação do exemplo

- Assim, para acessar o campo telefone de ficha, escrevemos:

ficha.telefone = 4921234;

- Como a struct ficha pessoal possui um campo, endereco, que também é uma struct, podemos fazer acesso aos campos desta struct interna da seguinte maneira:

ficha.endereco.numero = 10;

ficha.endereco.CEP=31340230;

- Desta forma, estamos acessando, primeiramente, o campo endereco da struct ficha e, dentro deste campo, estamos acessando o campo numero e o campo CEP.

Matriz de Estruturas

- Um estrutura é como qualquer outro tipo de dado no C. Podemos, portanto, criar matrizes de estruturas. Vamos ver como ficaria a declaração de um vetor de 100 fichas pessoais:

Ex: struct ficha_pessoal fichas [100];

- Poderíamos então acessar a segunda letra da sigla de estado da décima terceira ficha fazendo:

Ex: fichas[12].endereco.sigla_estado[1];

Atribuição de estruturas

Podemos atribuir duas estruturas que sejam do *mesmo* tipo. O C irá, neste caso, copiar uma estrutura, campo por campo, na outra. Veja o programa abaixo:

```
    struct est1 {
    int i;
    float f;
};
void main()
{
    struct est1 primeira, segunda; /* Declara primeira e segunda como structs do tipo est1 */
    primeira.i = 10;
    primeira.f = 3.1415;
    segunda = primeira; /* A segunda struct e' agora igual a primeira */
    printf(" Os valores armazenasdos na segunda struct sao : %d e %f ", segunda.i , segunda.f);
}
```

Comentários

- São declaradas duas estruturas do tipo *est1,* uma chamada *primeira* e outra chamada *segunda*. Atribuem-se valores aos dois campos da struct primeira. Os valores de *primeira* são copiados em *segunda* apenas com a expressão de atribuição:

  segunda = primeira;

- Todos os campos de primeira serão copiados na segunda. Note que **isto é diferente do que acontecia em vetores**, onde, para fazer a cópia dos elementos de um vetor em outro, tínhamos que copiar elemento por elemento do vetor.

Atribuição de structs que contenham campos ponteiros

```
#include <stdio.h>

#include <string.h>

#include <stdlib.h>

struct tipo_end

{
    char *rua; /* A struct possui um campo que é um ponteiro */
    int numero;

};
```

Exercício

- Escreva um programa fazendo o uso de struct's. Você deverá criar uma struct chamada Ponto, contendo apenas a posição x e y (inteiros) do ponto.
- Declare 2 pontos, leia a posição (coordenadas x e y) de cada um e calcule a distância entre eles. Apresente no final a distância entre os dois pontos.
- Distância é a raiz quadrada da soma dos quadrados da diferença entre as coordenadas x e y do ponto 2 para as coordenadas x e y do ponto 1

```
void main()
{
    struct tipo_end end1, end2;
     char buffer[50];
     printf("\nEntre o nome da rua:");
     gets(buffer); /* Le o nome da rua em uma string de buffer */
     end1.rua = (char *) malloc((strlen(buffer)+1)*sizeof(char)); /* Aloca a quantidade de memoria suficiente para armazenar a string
     */
     strcpy(end1.rua, buffer); /* Copia a string */
     printf("\nEntre o numero:");
     scanf("%d", &end1.numero);
```

```
end2 = end1; /* ERRADO end2.rua e end1.rua estao apontando para a mesma regiao de memoria */

printf("Depois da atribuicao:\n Endereco em end1 %s %d \n Endereco em end2 %s %d", end1.rua,end1.numero,end2.rua,
end2.numero);

strcpy(end2.rua, "Rua Mesquita"); /* Uma modificacao na memoria apontada por end2.rua causara' a modificacao do que e'
apontado por end1.rua

end2.numero = 1100; /* Nesta atribuicao nao ha problemas */

printf(" \n\nApos modificar o endereco em end2:\n Endereco em end1 %s %d \n Endereco em end2 %s %d", end1.rua,
end1.numero, end2.rua, end2.numero);

}
```

Passando para funções

- No exemplo apresentado no ítem usando, vimos o seguinte comando:

  strcpy (ficha.nome,"Luiz Osvaldo Silva");

- Neste comando um elemento de uma estrutura é passado para uma função. Este tipo de operação pode ser feita sem maiores considerações.
- Podemos também passar para uma função uma estrutura inteira. Veja a seguinte função:

  ```
  void PreencheFicha (struct ficha_pessoal ficha)
  {
  ...
  }
  ```

Considerações

- Como vemos acima é fácil passar a estrutura como um todo para a função. Devemos observar que, como em qualquer outra função no C, a passagem da estrutura é feita por valor.
- A estrutura que está sendo passada, vai ser copiada, campo por campo, em uma variável local da função PreencheFicha.
- Isto significa que alterações na estrutura dentro da função não terão efeito na variável fora da função.
- Mais uma vez podemos contornar este pormenor usando ponteiros e passando para a função um ponteiro para a estrutura.

Ponteiros e Estrutura

- Podemos ter um ponteiro para uma estrutura. Vamos ver como poderia ser declarado um ponteiro para as estruturas de ficha que estamos usando nestas seções:

  struct ficha_pessoal *p;

- Os ponteiros para uma estrutura funcionam como os ponteiros para qualquer outro tipo de dados no C. Para usá-lo, haveria duas possibilidades. A primeira é apontá-lo para uma variável struct já existente, da seguinte maneira:

  ```
  struct ficha_pessoal ficha;
  struct ficha_pessoal *p;
  p = &ficha
  ```

- A segunda é alocando memória para ficha_pessoal usando, por exemplo, malloc():

```
#include <stdlib.h>

main()

{
    struct ficha_pessoal *p;
    int a = 10; /* Faremos a alocacao dinamica de 10 fichas pessoais */
    p = (struct ficha_pessoal *) malloc (a * sizeof(struct ficha_pessoal));
    p[0].telefone = 3443768; /* Exemplo de acesso ao campo telefone da primeira ficha apontada por p */
    free(p);

}
```

Structs e ponteiros

- Cada registro tem um endereço na memória do computador. (Você pode imaginar que o endereço de um registro é o endereço de seu primeiro campo.) É muito comum usar um ponteiro para guardar o endereço de um registro. Dizemos que um tal ponteiro aponta para o registro. Por exemplo,

- data *p; /* p é um ponteiro para registros dma */
- data x;
- p = &x; /* agora p aponta para x */
- (*p).dia = 31; /* mesmo efeito que x.dia = 31 */

Ponteiros referenciando structs

- (Se apontarmos o ponteiro **p** para uma estrutura qualquer (como fizemos em p = &ficha;) e quisermos acessar um elemento da estrutura poderíamos fazer:

  (*p).nome

- Os parênteses são necessários, porque o operador . tem precedência maior que o operador * . Porém, este formato não é muito usado. O que é comum de se fazer é acessar o elemento **nome** através do operador seta, que é formado por um sinal de "menos" (-) seguido por um sinal de "maior que" (>), isto é: **->** .

- Assim faremos:

  p->nome

- A declaração acima é muito mais fácil e concisa. Para acessarmos o elemento **CEP** dentro de **endereco** faríamos:

  p->endereco.CEP

Exercício

- Seja a seguinte struct que é utilizada para descrever os produtos que estão no estoque de uma loja :

```
struct Produto {
char nome[30]; /* Nome do produto */
int codigo; /* Codigo do produto */
double preco; /* Preco do produto */
};
```

a) Escreva uma instrução que declare uma matriz de Produto com 10 itens de produtos;

b) Atribua os valores "Pe de Moleque", 13205 e R$0,20 aos membros da posição 0 e os valores "Cocada Baiana", 15202 e R$0,50 aos membros da posição 1 da matriz anterior;

c) Faça as mudanças que forem necessárias para usar um ponteiro para Produto ao invés de uma matriz de Produtos. Faça a alocação de memória de forma que se possa armazenar 10 produtos na área de memória apontada por este ponteiro e refaça as atribuições da letra b;

d) Escreva as instruções para imprimir os campos que foram atribuídos na letra c.

Unions

- Uma declaração **union** determina uma *única* localização de memória onde podem estar armazenadas várias variáveis diferentes. A declaração de uma união é semelhante à declaração de uma estrutura:

  union nome_do_tipo_da_union

  {

  tipo_1 nome_1;

  tipo_2 nome_2;

  ...

  tipo_n nome_n;

  } variáveis_union;

Unions

- Como exemplo, vamos considerar a seguinte união:

```
union angulo
{
float graus;
float radianos;
};
```

- Nela, temos duas variáveis (**graus** e **radianos**) que, apesar de terem nomes diferentes, ocupam o *mesmo* local da memória. Isto quer dizer que só gastamos o espaço equivalente a um único **float**.
- Uniões podem ser feitas também com variáveis de diferentes tipos. Neste caso, a memória alocada corresponde ao tamanho da maior variável no union.

Exemplos

```
#include <stdio.h>

#define GRAUS 'G'

#define RAD 'R'

union angulo

{

   int graus;

    float radianos;

};
```

```
void main()
{
    union angulo ang;
    char op;
     printf("\nNumeros em graus ou radianos? (G/R):");
     scanf("%c",&op);
     if (op == GRAUS)
     {
        ang.graus = 180;
        printf("\nAngulo: %d\n",ang.graus);
      }
   else if (op == RAD)
     {
       ang.radianos = 3.1415;
        printf("\nAngulo: %f\n",ang.radianos);
     }
    else printf("\nEntrada invalida!!\n");
}
```

Cuidado ao usar Union

Temos que tomar o maior cuidado pois poderíamos fazer:

```
#include <stdio.h>
union numero
{
    char Ch;
    int I;
    float F;
};
main (void)
{
   union numero N;
    N.I = 123;
    printf ("%f",N.F); /* Vai imprimir algo que nao e' necessariamente 123 ...*/
    return 0;
}
```

O programa acima é muito perigoso pois você está lendo uma região da memória, que foi "gravada" como um inteiro, como se fosse um ponto flutuante. Tome cuidado! O resultado pode não fazer sentido.

Diferença entre Union e Struct

- O tamanho da estrutura será a soma do total de variáveis pelos tipos, ou seja, 4bytes para int e float e 1byte para char, no final teremos 9 bytes.
- No caso de uma union o que acontece é que as variáveis são armazenadas com base no tamanho do maior tipo dentro da união, ou seja, se a estrutura acima fosse uma union o tamanho alocado na memoria seria o de 4bytes e nada mais nada menos que isso.
- A Union acessa apenas uma variável por vez. Dessa forma você deve saber o que está sendo armazenado na sua união para que não receba como retorno um comportamento indefinido.

Utilização do typedef

- Registros podem ser tratados como um novo tipo-de-dados. Depois da seguinte definição, por exemplo, podemos passar a dizer data no lugar de struct dma:

- typedef struct dma data;
- data x, y;

Definição de Typedef

- Em C e C++ podemos redefinir um tipo de dado dando-lhe um novo nome.
- Essa forma de programação ajuda em dois sentidos: 1º. Fica mais simples entender para que serve tal tipo de dado; 2º. É a única forma de conseguirmos referenciar uma estrutura de dados dentro de outra (struct dentro de struct).
- Para redefinirmos o nome de um tipo de dado usamos o comando typedef, que provém de type definition.
- Typedef faz o compilador assumir que o novo nome é um certo tipo de dado, e então, passamos a usar o novo nome da mesma forma que usaríamos o antigo.

Utilizando o typedef

- Por exemplo, podemos definir que, ao invés de usarmos int, agora usaremos inteiro ou, ao invés de usarmos float, usaremos decimal.
- Typedef deve sempre vir antes de qualquer programação que envolva procedimentos (protótipo de funções, funções, função main, structs, etc.) e sua sintase base é:
- typedef nome_antigo nome novo;
  - typedef int inteiro;
  - typedef float decimal;

```
#include <iostream>
#include <stdlib>
typedef int inteiro;
typedef float decimal;
int main (){
    inteiro x = 1;
    decimal y = 1.5;
    printf (“X = %d e Y = %f”, x, y);
    system ("pause");
    return 0;
}
```

Nota importante

- O uso de typedef para redefinir nomes de tipos primitivos (como int, float, char) é altamente desencorajado por proporcionar uma menor legibilidade do código.
- Portanto, devemos utilizar typedef apenas em momentos oportunos (como por exemplo, definir o nome de uma estrutura de dados complexa - struct).

Definindo nomes para estruturas de dados

- Uma vantagem muito grande que typedef nos proporciona é definir um nome para nossa estrutura de dados (struct).
- Graças a isso, somos capazes de auto-referenciar a estrutura, ou seja, colocar um tipo de dado struct dentro de outro struct.
- Podemos definir o nome de uma estrutura de dados (struct) de duas maneiras:
- Definindo o nome da estrutura e só depois definir a estrutura; ou definir a estrutura ao mesmo tempo que define o nome.

Exercício

- Escreva uma função que receba um número inteiro que representa um intervalo de tempo medido em minutos e devolva o correspondente número de horas e minutos (por exemplo, converte 131 minutos em 2 horas e 11 minutos). Use uma struct como a seguinte:

```
struct hm {
   int horas;
   int minutos;
};
```