Fundamentos de Programação
Registros (*structs*) - parte 2

Dainf - UTFPR

Profa. Leyza B. Dorini
Prof. Bogdan T. Nassu

Utilizando structs

Lembre-se: a `struct` é um tipo!!

Portanto, lembre-se: podemos usar a `struct` definida para criar variáveis/*arrays*, representar entradas e retornos de funções, etc. Iremos discutir algumas dessas coisas neste material!

Vetores e `structs`

Vetor de registros: tamanho estático

Assim como é possível declarar vetores de inteiros ou de `floats`, podemos declarar vetores do tipo definido por uma `struct` (por exemplo, para cadastrar todos os alunos de uma mesma turma).

Declarando vetores: `tipo nome[TAM];`

```
Estudante v[5];
```

Acessando vetores: `v[indice].campo;`

```
v[0].mat = 109898;
strcpy(v[0].nome, "Foolano Detal");
```

Obviamente, você também pode declarar matrizes do tipo `Estudante`!

Exemplo: vetor de registros (estático)

```
1  #define N 5
2  int main(){
3
4      Estudante v[N];
5      int i;
6
7      for (i = 0; i < TAM; i++) {
8        printf ("Mat e cr do %dº aluno: ", i+1);
9        scanf ("%d %f", &v[i].mat, &v[i].cr);
10     }
11
12     printf("Lista de matriculas e CRs: \n");
13     for (i = 0; i < TAM; i++)
14       printf ("%d -- %f", v[i].mat, v[i].cr);
15
16     return 0;
17 }
```

Declaração do vetor

Observe a sintaxe: primeiro acessa o índice específico do vetor, e depois o campo!

Alocando dinamicamente um vetor de registros

Fazemos exatamente da mesma forma que com um vetor de inteiros! Fique atento(a) aos tipos do `sizeof()` e do *casting*!

```
1  int main(){
2     Estudante *v;
3     int n;
4
5     printf("Qual o tamanho do vetor?");
6     scanf("%d", &n);
7
8     v = (Estudante*) malloc (n * sizeof(Estudante));
9
10    printf("Digite as %d matrículas: \n", n);
11    for (i = 0; i < n; i++)
12      scanf("%d", &v[i].mat);
13
14    free(v);
15    return 0;
16 }
```

Registros e funções

Registros em funções

Registros podem ser passados como parâmetros de uma função ou retornados por ela, como qualquer outro tipo.

- O parâmetro formal recebe uma cópia do registro, da mesma forma que em uma atribuição envolvendo registros.
- Quando a função retorna um registro, ele é novamente copiado como resultado da expressão.

Dado que, ao copiar um regitro para outro todos os campos são copiados, não precisamos nos preocupar com isso!

Exemplo: registros em funções (parte 1/2)

```
1  //Funcao que imprime os campos da struct
2  void imprime(Estudante x){
3      printf ("%d tirou média %f\n", x.mat, x.cr);
4  }
5
6  //Funcao que preenche os campos da struct
7  Estudante preenche(int m, float cr, char n[]){
8
9    Estudante aux;
10
11   aux.mat = m;
12   aux.cr = cr;
13   strcpy(aux.nome, n);
14
15   return aux;
16
17 }
```

Exemplo: registros em funções (parte 2/2)

Para invocar as funções anteriores, poderíamos fazer:

```
1  int main(){
2
3    Estudante x;
4
5    x = preenche(1029010, 8.8, "Foolano Detal");
6    imprime(x);
7
8    return 0;
9  }
```

E se fosse necessário passar `x` por referência para a função `preencheCadastro()`? Seria preciso acessar `structs` via ponteiros!

Uma esquisitice...

O que acontece na chamada da função do código abaixo?

```
1  typedef struct
2  {
3    int vetor [4];
4
5  } UmaStruct;
6
7  void fooFunction ( int um_vetor [4], UmaStruct s )
8  {
9    int i;
10   for (i = 0; i < 4; i++)
11   {
12     s.vetor [i] = i;
13     um_vetor [i] = i;
14   }
15 }
16
17
```

Em C, vetores são sempre passados por referência...

... mas o vetor dentro de s está sendo copiado.

Registros e ponteiros

Ponteiros para `struct` são muito, muito comuns. Copiar todo o conteúdo de uma `struct` grande é ineficiente.

Acessando registros via ponteiros

Uma vez que definimos um tipo T baseado em uma struct, podemos usar variáveis deste tipo da mesma forma que um tipo primitivo básico, ou seja, também podemos passar variáveis do tipo T por referência.

Ponteiros para struct

Ponteiros para struct são tão comuns que existe uma notação especial para acessar os campos de uma variável apontada.

```
variavel->campo
```

Importante: esta notação é equivalente a: `(*variavel).campo`

Acessando registros via ponteiros

Considere o exemplo abaixo:

```
1  int main(){
2
3    Estudante *px, x;
4
5    px = &x;
6
7    (*px).mat = 10;
8
9    px->cr = 8.8;
10
11   return 0;
12  }
```

Acessamos: (1) o conteúdo apontado pelo ponteiro, (2) no campo específico. Os () são necessários pq o . tem maior precedência que o *

Uma sintaxe alternativa (mais simples) consiste em acessar o campo utilizando o operador ->!

Exemplo 01: registros em funções (passagem de parâmetros por referência)

```
1  void preenche(Estudante *aux, int m, float cr, char n[]){
2    aux->mat = m;
3    aux->cr = cr;
4    strcpy(aux->nome, n);
5  }
6
7  int main(){
8    Estudante x;
9
10   preenche(&x, 1029010, 8.8, "Foolano Detal");
11   imprime(x);
12
13   return 0;
14 }
```

Observe como é realizado o acesso aos campos: utilizando o ponteiro aux!

Passamos o endereço de x por parâmetro!

Exemplo 02: alocação dinâmica e passagem de parâmetros por referência

```
1
2  typedef struct estudante{
3    int mat;
4    char *nome;
5    float cr;
6  }Estudante;
7
8  void preenche(Estudante *aux, int m, float cr, char n[]){
9
10   aux->mat = m;
11   aux->cr = cr;
12   aux->nome = (char*)malloc(strlen(n)*sizeof(char));
13   strcpy(aux->nome, n);
14 }
```

Na definição da `struct`, não temos um vetor, mas sim um ponteiro

Ao preencher os campos da variável (do tipo `Estudante`), é alocada apenas a memória necessária para armazenar a string recebida por parâmetro!

Considerações finais

- A representação de tipos de forma mais abstrata é o primeiro passo em direção à Orientação a Objeto.
- Em materiais de apoio e aulas de outros professores, você pode encontrar uma forma diferente de declarar structs.
  - A forma que usamos, com typedef, define a struct como um tipo. Esta é a forma mais próxima do modo de usar classes em linguagens como Java e C++!
  - Se precisar entender como se usam structs sem o typedef, consulte o livro ou outra referência da linguagem!

Lista!

Agora é contigo: faça a lista de exercícios!