Programação Estruturada

Aula 20 - Structs

Yuri Malheiros (yuri@ci.ufpb.br)

Introdução

- Até agora, a única estrutura de dados que vimos foi o array
- Elementos de um array têm o mesmo tipo e para acessá-los usamos um índice numérico
- Os elementos (membros) de uma `struct` não precisam ter o mesmo tipo
- Para acessar um membro, usamos o seu nome

Declarando uma struct

- Suponha que vamos armazenar dados sobre uma pessoa
- Vamos armazenar o seu nome, idade, peso e altura

```
struct {
  char nome[100];
  int idade;
  float peso;
  float altura;
} pessoa1, pessoa2;
```

- `struct {...}` define um tipo
- `pessoa1` e `pessoa2` definem variáveis
- Os membros da `struct` são armazenados na memória na ordem que foram declarados

Declarando uma struct

- Cada `struct` representa um novo escopo
  - Podemos ter membros com nomes iguais em `struct`s diferentes
  - Os nomes dos membros podem ser iguais ao de variáveis

Declarando uma struct

```
struct {
  char nome[100];
  int idade;
  float peso;
  float altura;
} pessoa1, pessoa2;
```

```
struct {
  char nome[100];
  float peso;
  float preco;
} produto1, produto2;
```

Inicializando uma struct

- Uma `struct` pode ser inicializada na sua declaração

```
struct {
  char nome[100];
  int idade;
  float peso;
  float altura;
} pessoa1 = {"Maria da Silva", 25, 60.0, 1.65},
  pessoa2 = {"Joao da Silva", 20, 65.0, 1.70};
```

- Os valores na inicialização precisam aparecer na mesma ordem dos membros da `struct`

Inicializadores designados

- Com um inicializador designado, podemos rotular cada um dos valores com o nome do seu membro correspondente

```
pessoa1 = {.nome = "Maria da Silva", .idade = 25, .peso = 60.0, .altura = 1.65}
```

- A ordem não importa:

```
pessoa1 = {.altura= 1.65, .nome = "Maria da Silva", .peso = 60, .idade = 25}
```

Inicializadores designados

- Usar inicializadores designados tem muitas vantagens
- Facilita a leitura
- Como a ordem não importa, o programador não precisa lembrar da ordem declarada na `struct`
- Se a ordem dos membros mudar em alterações futuras do código, o programador não precisa se preocupar

Operações

- Para acessar um membro de uma `struct`, nós usamos o seu nome

```
printf("%s\n", pessoa1.nome);
printf("%d\n", pessoa1.idade);
printf("%f\n", pessoa1.peso);
printf("%f\n", pessoa1.altura);
```

Operações

- Podemos atribuir novos valores aos membros

```
pessoa1.altura = 1.70;
pessoa1.idade++;
```

Operações

- O `.` na verdade é um operador
- Ele tem precedência igual ao do `++` e `--` pós-fixados
- `scanf("%d", &pessoa1.idade)` computa o endereço para `pessoa1.idade`

Operações

- Ao atribuir `pessoa1 = pessoa2`
- O compilador copia `pessoa2.nome` para `pessoa1.nome`, `pessoa2.idade` para `pessoa1.idade` e assim sucessivamente

Tipos struct

- Existem casos em que novas variáveis `struct` precisam ser declaradas depois da definição da `struct`
- Poderíamos fazer numa parte do programa:

```
struct {
  char nome[100];
  float peso;
  float preco;
} produto1;
```

- E em outra parte:

```
struct {
  char nome[100];
  float peso;
  float preco;
} produto2;
```

Tipos struct

- Isso pode causar muitos problemas:
- Repetição de código
- Modificar o programa será um desafio
    - Precisamos ter certeza que todas as declarações de variáveis estão consistentes
- Para o C, `produto1` e `produto2` não possuem o mesmo tipo
    - Não podemos atribuir `produto1` a `produto2` ou vice-versa
- Como não temos um nome para o tipo de `produto1` e `produto2` também não podemos passá-los como argumentos na chamada de uma função

Tipos struct

- Para evitar esses problemas, precisamos definir um nome para representar o tipo da `struct`

```
struct produto {
  char nome[100];
  float peso;
  float preco;
};
```

- `produto` identifica a `struct` definida
- Podemos declarar variáveis assim: `struct produto produto1, produto2;`

Tipos struct

- Podemos dar um nome a `struct` e declarar variáveis de uma só vez:

```
struct produto {
  char nome[100];
  float peso;
  float preco;
} produto1, produto2;
```

Tipos struct

- Uma alternativa para nomear uma `struct` é usar `typedef`

```
typedef struct produto {
  char nome[100];
  float peso;
  float preco;
} Produto;
```

- Podemos declarar variáveis assim: `Produto produto1, produto2;`

Structs como argumento e como retorno

- Funções podem receber `struct`s como argumentos e podem retornar `struct`s
- Para receber como argumentos, temos:

```
void print_produto(struct produto p) { ... }
```

- Para chamar a função: `print_produto(produto1);`

Structs como argumento e como retorno

- Para retornar uma `struct`:

```
struct produto build_produto(char *nome, float peso, float preco) {
  struct produto p;

  strcpy(p.nome, nome);
  p.peso = peso;
  p.preco = preco;

  return p;
}
```

- Para chamar a função: `build_produto("Joao da Silva", 20, 65.0, 1.70);`

Structs aninhandas

- Um membro de uma `struct` pode ser do tipo de uma `struct`

```c
struct nome_pessoa {
  char primeiro_nome[30];
  char sobrenome[30];
};

struct aluno {
  struct nome_pessoa nome;
  int matricula;
  int idade;
} aluno1, aluno2;
```

- Para acessar o primeiro nome do aluno, temos: `aluno1.nome.primeiro_nome`

Arrays de structs

- Um array pode ter elementos que são `structs`

```
struct produto estoque[100];
```

- Para acessar um produto: `estoque[i]`
- Para acessar um membro: `estoque[i].preco`

Arrays de structs

- Dada a `struct`

```
struct ddi {
  char pais[50];
  int code;
};
```

Arrays de structs

- Podemos inicializar o array de `structs` assim:

```
struct ddi codigo_paises[] =
{{"Argentina", 54}, {"Brasil", 55}, {"China", 86}, {"França", 33}, {"Alemanha", 49}}
```