

Linguagem de Programação

Linguagem de Programação C
Estruturas Heterogêneas

Sumário

1 Definição

2 Entrada e Saída

3 Manipulação com Vetores

4 TAD

5 Estruturas Aninhadas

6 Exemplos

7 Tópicos Avançados

Estruturas são grupos de variáveis relacionadas entre si algumas vezes chamadas agregadas sob um nome.

(Deitel; Deitel, 2011)

Em C, uma estrutura é uma coleção de variáveis referenciadas por um nome, fornecendo uma maneira conveniente de se ter informações relacionadas agrupadas.

(Schildt, 1996)

```
1 typedef struct
2 {
3     int id;
4     char cpf[12];
5     char nome[20];
6     char sobrenome[20];
7     int idade;
8     char data_nascimento[11];
9     float altura;
10 } Pessoa;
```

```
1 #include <stdio.h>
2 struct Pessoa{
3     int id;
4     char cpf[12];
5     char nome[20];
6     char sobrenome[20];
7     int idade;
8     char data_nascimento[11];
9     float altura;
10 };
```

Código 1: Definição de struct.

```
1 struct Pessoa p;
```

```
1 struct p;
```

Código 2: Declaração de struct.

```
1 int main(int argc, char const *argv[])
2 {
3     Pessoa fulana;
4     int id = 0;
5
6     fulana.id = id;
7     printf("CPF:\n");
8     scanf("%s", fulana.cpf);
9     printf("Nome:\n");
10    scanf("%s", fulana.nome);
11    printf("Sobrenome:\n");
12    scanf("%s", fulana.sobrenome);
13    printf("Idade:\n");
14    scanf("%d", &fulana.idade);
15    printf("Data de Nascimento:\n");
16    scanf("%s", fulana.data_nascimento);
17    printf("Altura:\n");
18    scanf("%f", &fulana.altura);
19    id++;
20    return 0;
21 }
```

Código 3: Leitura dos campos da estrutura.

```
1 int main(int argc, char const *argv[])
2 {
3     Pessoa fulana;
4
5     printf("ID: %d\n", fulana.id);
6     printf("CPF: %s\n", fulana.cpf);
7     printf("Nome: %s\n", fulana.nome);
8     printf("Sobrenome: %s\n", fulana.sobrenome);
9     printf("Idade: %d\n", fulana.idade);
10    printf("Data de Nascimento: %s\n", fulana.data_nascimento);
11    printf("Altura: %.2f\n", fulana.altura);
12    return 0;
13 }
```

Código 4: Exibindo valores armazenados na estrutura.

Manipulação com Vetores

```

1 int main(int argc, char const *argv[])
2 {
3     Pessoa fulana[N];
4     for (int i = 0; i < N; i++)
5     {
6         fulana[i].id = i;
7         printf("CPF:\n");
8         scanf("%s", fulana[i].cpf);
9         printf("Nome:\n");
10        scanf("%s", fulana[i].nome);
11        printf("Sobrenome:\n");
12        scanf("%s", fulana[i].sobrenome);
13        printf("Idade:\n");
14        scanf("%d", &fulana[i].idade);
15        printf("Data de Nascimento:\n");
16        scanf("%s", fulana[i].data_nascimento);
17        printf("Altura:\n");
18        scanf("%f", &fulana[i].altura);
19    }
20    return 0;
21 }

```

Código 5: Uso de estruturas dentro de vetores.

TAD

- Um **TAD (Tipo Abstrato de Dados)** é uma forma de organizar dados e operações de forma abstrata, separando a interface da implementação.
- Ele define **quais operações** podem ser realizadas sobre os dados, sem expor **como essas operações são implementadas**.
- Permite ao programador pensar em termos de **o que o dado representa e o que pode ser feito com ele**, sem se preocupar com detalhes internos.
- A utilização de TADs facilita a **manutenção, reutilização e segurança** do código.

```
1 void inserir(Pessoa pessoas[]){
2     for (int i = 0; i < N; i++)
3     {
4         pessoas[i].id = i;
5         printf("CPF:\n");
6         scanf("%s", pessoas[i].cpf);
7         printf("Nome:\n");
8         scanf("%s", pessoas[i].nome);
9         printf("Sobrenome:\n");
10        scanf("%s", pessoas[i].sobrenome);
11        printf("Idade:\n");
12        scanf("%d", &pessoas[i].idade);
13        printf("Data de Nascimento:\n");
14        scanf("%s", pessoas[i].data_nascimento);
15        printf("Altura:\n");
16        scanf("%f", &pessoas[i].altura);
17    }
18 }
```

Código 6: Inserindo registros.

```

1 void listar(Pessoa pessoas[]){
2     for (int i = 0; i < N; i++)
3     {
4         if(pessoas[i].id != VAZIO){
5             printf("ID: %d\n", pessoas[i].id);
6             printf("CPF: %s\n", pessoas[i].cpf);
7             printf("Nome: %s\n", pessoas[i].nome);
8             printf("Sobrenome: %s\n", pessoas[i].sobrenome);
9             printf("Idade: %d\n", pessoas[i].idade);
10            printf("Data de Nascimento: %s\n", pessoas[i].data_nascimento);
11            printf("Altura: %.2f\n", pessoas[i].altura);
12        }
13    }
14 }

```

Código 7: Listando todos os registros.

```

1 int buscar(Pessoa pessoas[], char cpf[]){
2     for (int i = 0; i < N; i++)
3     {
4         if(!strcmp(pessoas[i].cpf, cpf)){
5             printf("ID: %d\n", pessoas[i].id);
6             printf("CPF: %s\n", pessoas[i].cpf);
7             printf("Nome: %s\n", pessoas[i].nome);
8             printf("Sobrenome: %s\n", pessoas[i].sobrenome);
9             printf("Idade: %d\n", pessoas[i].idade);
10            printf("Data de Nascimento: %s\n", pessoas[i].data_nascimento);
11            printf("Altura: %.2f\n", pessoas[i].altura);
12            return 0;
13        }
14    }
15    printf("Nao encontrado!\n");
16    return 1;
17 }

```

Código 8: Buscando registro único.

```
1 int atualizar(Pessoa pessoas[], char cpf[]){
2     for (int i = 0; i < N; i++)
3     {
4         if(!strcmp(pessoas[i].cpf, cpf)){
5             printf("CPF:\n");
6             scanf("%s", pessoas[i].cpf);
7             printf("Nome:\n");
8             scanf("%s", pessoas[i].nome);
9             printf("Sobrenome:\n");
10            scanf("%s", pessoas[i].sobrenome);
11            printf("Idade:\n");
12            scanf("%d", &pessoas[i].idade);
13            printf("Data de Nascimento:\n");
14            scanf("%s", pessoas[i].data_nascimento);
15            printf("Altura:\n");
16            scanf("%f", &pessoas[i].altura);
17            return 0;
18        }
19    }
20    printf("Nao encontrado!\n");
21    return 1;
22 }
```

Código 9: Atualizando registro.

```
1 int excluir(Pessoa pessoas[], char cpf[]){
2     for (int i = 0; i < N; i++)
3     {
4         if(!strcmp(pessoas[i].cpf, cpf)){
5             pessoas[i].id = VAZIO;
6             strcpy(pessoas[i].cpf, " ");
7             strcpy(pessoas[i].nome, " ");
8             strcpy(pessoas[i].sobrenome, " ");
9             pessoas[i].idade = -1;
10            strcpy(pessoas[i].data_nascimento, " ");
11            pessoas[i].altura = -1.0;
12            printf("Excluido com sucesso!\n");
13            return 0;
14        }
15    }
16    printf("Nao encontrado!\n");
17    return 1;
18 }
```

Código 10: Excluindo registro.

Estruturas Aninhadas

```
1 typedef struct{
2     char rua[100];
3     int numero;
4     char bairro[20];
5     char cep[10];
6     char cidade[100];
7     char uf[3];
8 }Endereço;
9 typedef struct{
10     char cpf[12];
11     char nome_completo[100];
12     Endereço endereco;
13 }Pessoa;
14 int main(){
15     Pessoa pessoa;
16     pessoa.endereco.numero = 1024;
17     return 0;
18 }
```

Código 11: Exemplo de *struct* Pessoa e Endereço.

Exemplos

```
1 typedef struct
2 {
3     char nome[50];
4     float nota1;
5     float nota2;
6     float nota3;
7 }Aluno;
```

Código 12: Exemplo simples de uso de nota com struct.

```
1 typedef struct
2 {
3     char nome[50];
4     float notas[3];
5 }Aluno;
```

Código 13: Exemplo com vetor de notas.

```
1 typedef struct
2 {
3     char nome[50];
4     float notas[3][2]; //notas por disciplina.
5 }Aluno;
```

Código 14: Exemplo com matriz de notas usando struct.

Tópicos Avançados

Uma *união* é um tipo composto similar a *struct*, mas ***todos os membros compartilham o mesmo espaço de memória***. Ou seja, em uma *union*, apenas ***um membro por vez*** pode armazenar um valor válido.

```

1 #include <stdio.h>
2 union Numero {
3     int i;
4     float f;
5     char c;
6 };
7 int main() {
8     union Numero n;
9     n.i = 10;
10    printf("i = %d | Endereço: %p\n", n.i, &n.i);
11    n.f = 3.14;
12    printf("f = %.2f | Endereço: %p\n", n.f, &n.f);
13    printf("i agora = %f (valor alterado)\n", n.i);
14    return 0;
15 }

```

Código 15: Exemplo básico de union

**Atenção!**

Modificar um membro da union sobrescreve os dados de outros membros. O tamanho da union é igual ao tamanho do seu **maior membro**.

- › Economizar memória quando apenas **um membro por vez** é necessário.
- › Representar tipos de dados variantes, como em protocolos de rede, arquivos binários ou árvores sintáticas.

- O **alinhamento** determina em quais endereços da memória um tipo de dado pode ser armazenado, geralmente múltiplos de seu tamanho ou exigência da arquitetura.
- O **padding** é o espaço extra que o compilador insere para garantir que os membros de uma **struct** fiquem corretamente alinhados. Mesmo que a soma dos tamanhos dos membros seja pequena, `sizeof(struct)` pode ser maior devido ao **padding** adicionado pelo compilador.

```
1 #include <stdio.h>
2 #include <stdalign.h>
3 struct A {
4     char c1;      // 1 byte
5     int i;        // 4 bytes
6     char c2;      // 1 byte
7 };
8
9 struct B {
10     char c1; //1 byte
11     char c2; //1 byte
12     int i;   // 4 bytes
13 };
14
```

```
15 int main() {  
16     struct A a;  
17     struct B b;  
18     printf("Struct A:\n");  
19     printf("    alignof(A) = %zu\n", alignof(struct A));  
20     printf("    sizeof(A)  = %zu\n", sizeof(struct A));  
21     printf("\nStruct B:\n");  
22     printf("    alignof(B) = %zu\n", alignof(struct B));  
23     printf("    sizeof(B)  = %zu\n", sizeof(struct B));  
24     return 0;  
25 }
```

Código 16: Exemplo de alinhamento.

Alinhamento e Padding IV

char c = 'c'; → 1 byte
ASCII 'c' → 01100011

padding

int n = 270; → 4 bytes

0000 0000
0000 0000
0000 1111
1111 1111

0110 0011

0000 0000

0000 0000

0000 0000

0000 0000

0000 0000

0000 1111

1111 1111

› Dicas para otimizar structs:

- ›› Ordenar membros do maior para o menor tipo.
- ›› Agrupar membros do mesmo tipo.
- ›› Evitar structs pequenas dentro de structs grandes sem planejar o alinhamento.

Nota:

Pensar no *padding* não muda a lógica do programa, mas pode reduzir o consumo de memória e melhorar a eficiência de acesso.

Um `enum` é um tipo definido pelo usuário que representa um conjunto de constantes inteiras nomeadas. Ele melhora a legibilidade e a manutenção do código, evitando o uso de números “mágicos”.

```
1 #include <stdio.h>
2
3 typedef enum { DOMINGO, SEGUNDA, TERCA, QUARTA, QUINTA, SEXTA, SABADO } Dia;
4
5 int main() {
6     Dia hoje = TERCA;
7     if (hoje == DOMINGO || hoje == SABADO)
8         printf("Dia de descanso!\n");
9     else
10         printf("Dia útil!\n");
11     return 0;
12 }
```

Código 17: Exemplo de uso de enum

Atenção!

Por padrão, o primeiro valor do enum é 0, e os subsequentes incrementam 1 automaticamente. É possível atribuir valores explícitos aos elementos.

Nota:

Sempre utilize enums quando um valor pode assumir **conjunto limitado de estados**, ao invés de usar números ou caracteres diretamente.

(Deitel; Deitel, 2011) - Capítulo 10



DE OLIVEIRA, J.F.; MANZANO, J.A.N.G. **Algoritmos: Lógica para Desenvolvimento de Programação de Computadores**. 16. ed. São Paulo: Editora Érica, 2004.

DE SOUZA, M.A.F. *et al.* **Algoritmos e Lógica de Programação**. São Paulo: Thomson Learning, 2004.

DEITEL, Paul; DEITEL, Harvey. **C: Como Programar**. 6. ed. São Paulo: Pearson Universidades, 2011.

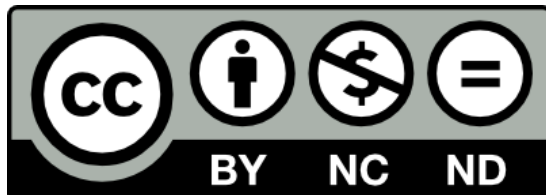
MEDINA, M.; FERTIG, C. **Algoritmos e Programação – Teoria e Prática**. São Paulo: Novatec, 2005.

SCHILDT, Herbert. **C Completo e Total: O Guia Definitivo para Programação em C**. 3. ed. São Paulo: Makron Books, 1996. ISBN 978-8534606928.



SCHILDT, Herbert. **C Completo e Total**. 4ª. São Paulo: McGraw-Hill Brasil, 2015.

Estes slides estão protegidos por uma licença Creative Commons



Este modelo foi adaptado de Maxime Chupin.

Linguagem de Programação

Linguagem de Programação C
Estruturas Heterogêneas