

SSC0501 - Introdução à Ciência de Computação I

Bem Vindos!

Structs (Estruturas)!

Tipos compostos

- Às vezes, precisamos de **um tipo de dado capaz de armazenar múltiplos valores**, em vez de apenas um único valor simples.
- Podemos agrupar esses tipos compostos em duas categorias:

```
int matrix[3][3]:
```

5	9	3
2	6	8
4	0	7

9	2	3
7	8	5
2	2	3

2	2	0	8	6	1
4	9	6	5	3	0
1	2	8	5	2	0

Tipos compostos

- Às vezes, precisamos de **um tipo de dado capaz de armazenar múltiplos valores**, em vez de apenas um único valor simples.
- Podemos agrupar esses tipos compostos em duas categorias:
 - **Estruturas com dados homogêneos**
Todos os elementos são do mesmo tipo.
 - Exemplo: **Vetores e Matrizes**

Tipos compostos

- Às vezes, precisamos de **um tipo de dado capaz de armazenar múltiplos valores**, em vez de apenas um único valor simples.
- Podemos agrupar esses tipos compostos em duas categorias:
 - **Estruturas com dados homogêneos**
Todos os elementos são do mesmo tipo.
 - Exemplo: **Vetores e Matrizes**
 - **Estrutura com dados heterogêneos**
Os elementos podem ser de tipos diferentes.
 - Exemplo: **Structs (Estruturas)**

Tipos Compostos

O que fazer quando precisamos armazenar na memória informações de diferentes tipos, como:

- Nome (string)
- Idade (inteiro)
- CPF (string ou número)
- Salário (ponto flutuante)

Structs!

O que fazer quando precisamos armazenar na memória informações de diferentes tipos, como:

- Nome (string)
- Idade (inteiro)
- CPF (string ou número)
- Salário (ponto flutuante)

Structs (ou **registros**) são a solução!

- **Structs** são coleções de dados heterogêneos, agrupados em um único elemento de dado.

struct - Sintaxe

A sintaxe de **definição** de uma struct em C é a seguinte:

```
struct nome_da_estrutura {  
    tipo_1 dado_1;  
    tipo_2 dado_2;  
    ...  
    tipo_n dado_n;  
};
```

Para criar **declarar** uma struct, utiliza se a sintaxe:

```
struct nome_da_estrutura nome_da_variavel;
```


struct - Exemplo

Exemplo de **definição** e **declaração** de uma struct:

```
struct dados_pessoa
{
    char nome[60];
    int idade;
    char cpf[12];
    float salario;
};

struct dados_pessoa p1, p2;
```

struct - Exemplo

Exemplo de **definição** e **declaração** de uma struct:

```
struct dados_pessoa  
{  
    char nome[60];  
    int idade;  
    char cpf[12];  
    float salario;  
};
```

```
struct dados_pessoa p1, p2;
```

Duas variáveis p1 e p2
declaradas como sendo do
tipo struct dados_pessoa!

struct - Acesso aos Campos

Assim como outras variáveis compostas, precisamos acessar e manipular **individualmente os campos** de uma struct.

Sintaxe de acesso a um campo individual:

nome_variavel_struct.nome_campo

Exemplo:

```
p1.idade = 30;    // Atribuição
if (p2.salario < 1000.00)
{
    // Faça algo
}
```

struct - Acesso aos Campos

Assim como outras variáveis compostas, precisamos acessar e manipular **individualmente os campos** de uma struct.

Sintaxe de acesso a um campo individual:

nome_variavel_struct.nome_campo

Exemplo:

```
p1.idade = 30;    // Atribuição
if (p2.salario < 1000.00)
{
    // Faça algo
}
```

Use o operador . para acessar campos de uma variável do tipo struct.

struct – Ponteiros para Structs

- A criação de **ponteiros para structs** funciona de forma análoga à de variáveis simples.
- Para acessar os campos por meio de um ponteiro, utilizamos o operador `->`.

Exemplo:

```
struct dados_pessoa
{
    int idade;
};

struct dados_pessoa pessoa, *pp;

pp = &pessoa;           // pp aponta para pessoa

pessoa.idade = 50;      // Acesso direto
pp->idade = 30;         // Acesso via ponteiro
```

struct – Ponteiros para Structs

- A criação de **ponteiros para structs** funciona de forma análoga à de variáveis simples.
- Para acessar os campos por meio de um ponteiro, utilizamos o operador `->`.

Exemplo:

```
struct dados_pessoa
{
    int idade;
};

struct dados_pessoa pessoa, *pp;

pp = &pessoa;          // pp aponta para pessoa

pessoa.idade = 50;      // Acesso direto
pp->idade = 30;         // Acesso via ponteiro
```

struct – Ponteiros para Structs

- A criação de **ponteiros para structs** funciona de forma análoga à de variáveis simples.
- Para acessar os campos por meio de um ponteiro, utilizamos o operador `->`.

Exemplo:

```
struct dados_pessoa
{
    int idade;
};

struct dados_pessoa pessoa, *pp;

pp = &pessoa;           // pp aponta para pessoa

pessoa.idade = 50;      // Acesso direto
pp->idade = 30;         // Acesso via ponteiro
```

Obs.: `pp->idade` é equivalente a `(*pp).idade`, mas mais legível.

struct – Ponteiros para Structs

- A criação de **ponteiros para structs** funciona de forma análoga à de variáveis simples.
- Para acessar os campos por meio de um ponteiro, utilizamos o operador `->`.

Exemplo:

```
struct dados_pessoa
{
    int idade;
};

struct dados_pessoa pessoa, *pp;

pp = &pessoa;        // pp aponta para pessoa

pessoa.idade = 50;    // Acesso direto
(*pp).idade = 30;    // Acesso via ponteiro
```

Obs.: `pp->idade` é equivalente a `(*pp).idade`, mas mais legível.

struct – Ponteiros para Structs

- A criação de **ponteiros para structs** funciona de forma análoga à de variáveis simples.
- Para acessar os campos por meio de um ponteiro, utilizamos o operador `->`.

Exemplo:

```
struct dados_pessoa
{
    int idade;
};

struct dados_pessoa pessoa, *pp;

pp = &pessoa;           // pp aponta para pessoa

pessoa.idade = 50;      // Acesso direto
(*pp).idade = 30;       // Acesso via ponteiro
```

Obs.: `pp->idade` é equivalente a `(*pp).idade`, mas mais legível.

Exercício

13.1) Criar uma estrutura para carros com as seguintes informações: fabricante, modelo, ano, cor, preço. O programa deve lê as informações de um carro e, depois da leitura imprimir as informações na tela utilizando a estrutura criada.

```
int matrix[3][3]:
```

13.2) Agora o seu programa deve ler as informações de um carro, e permitir a alteração de um dos campos lido. A informação que será alterada é informada através de um ID de 1 a 5 sendo:

1 -Fabricante, 2- Modelo, 3 - Ano, 4 - Cor, 5 - Preço

A seguir o programa deve ler a nova informação e imprimir todos dados atualizados.

RunCodes! Código Matricula: R4SM

<https://runcodes.icmc.usp.br/offerings/view/83>

Como ficou a struct?

```
int matrix[3][3]:
```

5	9	3
2	6	8
4	0	7

9	2	3
7	8	5
2	2	8

2	2	0	8	6	1
4	9	6	5	3	0
1	2	8	5	2	4

Como ficou a struct?

```
struct Carro {  
    char fabricante[21]; // até 20 caracteres + '\0'  
    char modelo[21];  
    int ano;  
    char cor[21];  
    float preco;  
};
```


Vetor de structs

- Podemos declarar vetores de structs para armazenar vários registros do mesmo tipo - por exemplo, uma turma de alunos.

```
int matrix[3][3]:
```

5	9	3
2	6	8
4	0	7

9	2	3
7	8	5
2	2	3

2	2	0	8	6	1
4	9	6	5	3	0
1	2	9	5	2	

Vetor de structs - Declaração e Acesso a Elementos

Declaração:

```
struct Aluno
{
    char nome[50];
    float nota;
};

struct Aluno turma[100]; // Vetor com 100 alunos
```

Acesso aos elementos:

```
strcpy(turma[0].nome, "Ana");
turma[0].nota = 9.0;

printf("Nota de %s: %.1f\n", turma[0].nome, turma[0].nota);
```

Vetor de structs - Declaração e Acesso a Elementos

Declaração:

```
struct Aluno
{
    char nome[50];
    float nota;
};
```

```
struct Aluno turma[
```

Acesso semelhante a vetores simples, mas com . para acessar os campos.

Acesso aos elementos:

```
strcpy(turma[0].nome, "Ana");
turma[0].nota = 9.0;
```

```
printf("Nota de %s: %.1f\n", turma[0].nome, turma[0].nota);
```

Exercício

13.3) Utilizando a estrutura de carro dos exercícios anteriores, agora o seu programa deve ler uma lista de carros e no final deve imprimir o carro mais barato da lista.

```
int matrix[3][3]:
```

RunCodes! Código Matricula: R4SM

<https://runcodes.icmc.usp.br/exercises/viewProfessor/1493>

`int` **typedef!** `[3]:`

Criando Nomes Alternativos para Tipos de
Dados!

typedef

O typedef permite **criar um nome alternativo (alias)** para um tipo já existente.

Sintaxe:

```
typedef tipo_de_dado novo_nome_alternativo;
```

```
int matrix[3][3];
```

typedef

O typedef permite **criar um nome alternativo (alias)** para um tipo já existente.

Exemplo:

```
typedef unsigned int uint;  
  
uint idade = 25; // Equivale a: unsigned int idade = 25;
```

Exemplo 2:

```
typedef int inteiro;  
  
inteiro idade = 30; // Equivale a: int idade = 30;
```

typedef

O typedef permite **criar um nome alternativo (alias)** para um tipo já existente.

Exemplo:

```
typedef unsigned int uint;  
  
uint idade = 25; // Equivale a: unsigned int idade = 25;
```

Exemplo 2:

```
typedef int inteiro;  
  
inteiro idade = 30; // Equivale a: int idade = 30;
```

typedef não cria um novo tipo - apenas um apelido para um tipo já existente.

typedef

O typedef permite **criar um nome alternativo (alias)** para um tipo já existente.

Exemplo:

```
typedef unsigned int uint;  
  
uint idade = 25; // Equivale a: unsigned int idade = 25;
```

Exemplo 2:

```
typedef int inteiro;  
  
inteiro idade = 30; // Equivale a: int idade = 30;
```

Útil para tornar o código mais legível ou padronizar nomes de tipos.

typedef com struct

O typedef também pode ser usado para simplificar o uso de structs, evitando repetir a palavra-chave struct.

Exemplo:

```
typedef struct
{
    char nome[50];
    int n_usp;
    float nota;
} Aluno;
```

Declaração:

```
Aluno a1;
a1.nota = 9.5;
```

typedef com struct

O typedef também pode ser usado para simplificar o uso de structs, evitando repetir a palavra-chave struct.

Exemplo:

```
typedef struct
{
    char nome[50];
    int n_usp;
    float nota;
} Aluno;
```

Sem typedef

Declaração:

```
Aluno a1;
a1.nota = 9.5;
```

typedef com struct

O typedef também pode ser usado para simplificar o uso de structs, evitando repetir a palavra-chave struct.

Exemplo:

```
typedef struct
{
    char nome[50];
    int n_usp;
    float nota;
} Aluno;
```

Sem typedef

```
typedef struct Aluno
{
    char nome[50];
    int n_usp;
    float nota;
} Aluno;
```

Declaração:

```
Aluno a1;
a1.nota = 9.5;
```

```
struct Aluno a1;
a1.nota = 9.5;
```


typedef com struct

O typedef também pode ser usado para simplificar o uso de structs, evitando repetir a palavra-chave struct.

Exemplo:

```
typedef struct
{
    char nome[50];
    int n_usp;
    float nota;
} Aluno;
```

Declaração:

```
Aluno a1;
a1.nota = 9.5;
```

Sem **struct Aluno**, apenas **Aluno**!
Fica mais limpo e próximo da ideia de "tipo definido pelo usuário".

Prática!

Código Matricula: R4SM

<https://runcodes.icmc.usp.br/offerings/view/83>

[run.codes]



Menu Professor ▾

matheus.m.santos@icmc.usp.br ▾

Hora do Servidor: 14/03/2023 18:46:06

Home > SSC0501

SSC0501 - Introdução à Ciência de Computação I

Professores/Monitores

Professores: Matheus Machado dos Santos
Turma: 2025101
Universidade: USP
Ativa até: 21/07/2025



Código de Matrícula

R4SM



Novo Exercício



Enviar E-mail



Ver Notas



Exportar Tabela de Notas

Exercícios

No.	Exercício	Status	Casos Corretos	Nota	Entregas	Participantes	Prazo de Entrega	Ações
1	Hello World	Finalizado	1/1	10.00	78	43/46	14/03/2025 21:00:00	Ver Detalhes Remover Exercício
2	1.01 Maior número	Não Entregue	0/6	0	0	0/46	19/03/2025 23:59:59	Ver Detalhes Remover Exercício
3	1.02 Par ou ímpar	Não Entregue	0/7	0	0	0/46	19/03/2025 23:59:59	Ver Detalhes Remover Exercício
4	1.03 Positivo, negativo ou zero	Não Entregue	0/7	0	0	0/46	19/03/2025 23:59:59	Ver Detalhes Remover Exercício
5	1.04 Maior de três números	Não Entregue	0/7	0	0	0/46	19/03/2025 23:59:59	Ver Detalhes Remover Exercício

SSC0501 - Introdução à Ciência de Computação I

Obrigado pela atenção!!