c_c++

tags:: #c #memoria
Banco de dados (mysql) Banco de dados (mongoDB)

---

aula 2 - programação modular

ponteiros

toda vez que passamos alguma variável como argumento para uma função, ela faz uma cópia dessa variável, o que pode causar problemas no quesito alocação de memória, além de que quando alteramos o valor desse argumento dentro da função não alteramos de fato o valor original dele, mas sim de uma copia dentro dela (tornando obrigatório retornar esse valor). Ao invés disso podemos passar o endereço de memória onde aquele variável está alocada por meio de um ponteiro (que como o nome sugere, é um tipo de variável que aponta para o endereço de outra variável) .

sintaxe

o primeiro passo para criarmos um ponteiro é definir o tipo do valor dele, ou seja, se ele vai ser `float, int, char etc...` , após isso criamos os ponteiros usando o caractere `*` antes o nome de variável.
Agora temos que fazer esse ponteiro apontar para alguma variável, ou seja, temos que passar um endereço de memória a ser apontado, para isso declaramos o nome dele (sem o `*`) e depois atribuímos a variável com o caractere `&` antes do nome dela.

```
int main(){

int number = 8;
int *ponteiro;

ponteiro = &number; //agora a variavel ponteiro vai apontar para number
}
```

De forma visual ficaria mais ou menos assim: a variável **ponteiro** aponta para o endereço de memoria de variável **x (A20203B)** que por sua vez esta **armazenando** o valor de dessa variável que é **8**.

Alterando o valor da variável

Para alterarmos o valor de variável que esta sendo apontada por um ponteiro, temos que chamar o ponteiro da mesma forma que ele foi declarado (com o caractere `*` antes do nome) e atribuir um novo valor contido no endereço da memória .

```
int main(){

int number = 8;
int *ponteiro;

ponteiro = &number; //agora a variavel ponteiro vai apontar para number

*ponteiro = 10;

printf("%d", number);// vai exibir 10
}
```

Ponteiro e vetores

Em C, você não precisa passar explicitamente o endereço de memória de um vetor para um ponteiro porque o nome de um vetor já é, por definição, um ponteiro para o primeiro elemento do vetor. Ou seja, o próprio nome do vetor atua como um ponteiro para o endereço da sua primeira posição.

```
int arr[5] = {1, 2, 3, 4, 5};
int *p;

p = arr; // 'arr' já é o endereço do primeiro elemento do vetor
```

Neste exemplo, `arr` atua como o endereço de `arr[0]`, então não precisamos usar o operador `&` para passar seu endereço.

Alterando os valores dentro de um vetor

Para alterar o valor de um vetor por meio de um ponteiro, você pode acessar e modificar os elementos diretamente usando o ponteiro, como se estivesse acessando o vetor por índices. Isso é feito tanto por notação de índices quanto por aritmética de ponteiros.

```
int arr[5] = {1, 2, 3, 4, 5};
int *p;

p = arr; // 'p' agora aponta para o primeiro elemento de 'arr'

// Alterando os valores usando o ponteiro
*p = 10;      // Altera o primeiro elemento (arr[0])
*(p + 1) = 20; // Altera o segundo elemento (arr[1])
p[2] = 30;     // Outra forma de alterar o terceiro elemento (arr[2])
```

```
// Resultado: arr = {10, 20, 30, 4, 5}
```

Aqui, `*p` acessa o primeiro elemento, `*(p + 1)` o segundo, e `p[2]` o terceiro. Assim, você pode modificar os valores diretamente.

Passagem por referência

podemos passar o endereço de memoria para alguma função e criar um ponteiro nele para alterar o valor da variável ao invés de criar um ponteiro especifico para cada variável. Vamos destrinchar esse exemplo:

```
#include <stdio.h>

void calculoVantagens(float numeroHoras, float salarioHora, int numeroFilhos,
float valorPorFilho, float *salarioBruto, float *salarioFamilia, float
*vantagens);

void calculoDeducoes(float salarioBruto, float taxaIR, float *INSS, float
*IRPF, float *Deducoes);


int main(){

    float numeroHoras, salarioHora, valorPorFilho, taxaIR;

    int numeroFilhos;

    float salarioBruto, salarioFamilia, vantagens, INSS, IRPF, Deducoes;



    printf("Digite o numero de horas trabalhadas: ");

    scanf(" %f", &numeroHoras);

    printf("Digite o salario por hora: ");

    scanf(" %f", &salarioHora);

    printf("Digite o numero de filhos: ");

    scanf(" %d", &numeroFilhos);

    printf("Digite o valor por filho: ");

    scanf(" %f", &valorPorFilho);
```

```
    printf("Digite a taxa de IR: ");

    scanf(" %f", &taxaIR);



    calculoVantagens(numeroHoras, salarioHora, numeroFilhos, valorPorFilho,
&salarioBruto, &salarioFamilia, &vantagens);

    calculoDeducoes(salarioBruto, taxaIR, &INSS, &IRPF, &Deducoes);



    printf("\n\nSalario Bruto: %.2f\n", salarioBruto);

    printf("Salario Familia: %.2f\n", salarioFamilia);

    printf("Vantagens: %.2f\n", vantagens);

    printf("INSS: %.2f\n", INSS);

    printf("IRPF: %.2f\n", IRPF);

    printf("Deducoes: %.2f\n", Deducoes);



    return 0;

}



void calculoVantagens(float numeroHoras, float salarioHora, int numeroFilhos,
float valorPorFilho, float *salarioBruto, float *salarioFamilia, float
*vantagens){

    *salarioBruto = numeroHoras * salarioHora;

    *salarioFamilia = numeroFilhos * valorPorFilho;

    *vantagens = *salarioBruto + *salarioFamilia;

}



void calculoDeducoes(float salarioBruto, float taxaIR, float *INSS, float
*IRPF, float *Deducoes){
```

```
    *INSS = salarioBruto * 0.08;

    *IRPF = salarioBruto * taxaIR;

    *Deducoes = *INSS + *IRPF;

}
```

Aqui temos um conjunto de dados de múltiplos funcionários que devem ser alterados , ao invés de altera os dados manualmente de cada funcionário podemos criar funções e passar somente o endereço de memoria das variáveis que queremos alterar.

Passando as variáveis

Após coletarmos os dados podemos passar os endereços dessas variáveis utilizando o caractere `&` antes do nome delas.

```
    calculoVantagens(numeroHoras, salarioHora, numeroFilhos, valorPorFilho,
&salarioBruto, &salarioFamilia, &vantagens);

    calculoDeducoes(salarioBruto, taxaIR, &INSS, &IRPF, &Deducoes);
```

Recebendo e utilizando os endereços

Na hora da função receber os argumentos temos de declarar os tipos deles como ponteiros , pois estamos recendo apenas o endereço de memória e precisamos manipular o valor contido nas variáveis, então criamos um ponteiro que vai apontar para elas.

Observação: no momento que declaramos o argumento da função como tipo ponteiro e passamos um endereço de memória para ele, na hora que criarmos ele dentro da função, o mesmo estará apontado para o endereço fornecido.

```
void calculoVantagens(float numeroHoras, float salarioHora, int numeroFilhos,
float valorPorFilho, float *salarioBruto, float *salarioFamilia, float
*vantagens){

    *salarioBruto = numeroHoras * salarioHora;

    *salarioFamilia = numeroFilhos * valorPorFilho;

    *vantagens = *salarioBruto + *salarioFamilia;

}



void calculoDeducoes(float salarioBruto, float taxaIR, float *INSS, float
```

```
*IRPF, float *Deducoes){

    *INSS = salarioBruto * 0.08;

    *IRPF = salarioBruto * taxaIR;

    *Deducoes = *INSS + *IRPF;

}
```

Aqui declaramos o ponteiro dentro da função e alteramos a valor de dentro da variável que ele aponta.

---

aula 3 - string

setLocale()

uma função da lib <locale.h> que formata os caracteres para o padrão de escrita, sintaxe `setlocale(LC_ALL, “Portuguese”);` (colocar dentro do main, no topo)

```
#include <stdio.h>
#include <locale.h>

int main() {
    // Definir a localidade para português do Brasil
    setlocale(LC_ALL, "pt_BR.UTF-8");

    // Definir um buffer para armazenar a string
    char buffer[50];

    // Solicitar ao usuário que digite algo
    printf("Digite uma frase (até 49 caracteres): ");

    // Usar fgets para ler no máximo 49 caracteres (o último espaço é para o
caractere nulo)
    fgets(buffer, sizeof(buffer), stdin);

    // Exibir a string digitada
    printf("Você digitou: %s", buffer);

    return 0;
}
```

Trabalhando com strings

O mais importante ao trabalhar com strings em C é lembrar que elas não têm o mesmo comportamento que em outras linguagens, pois o C não oferece suporte nativo a strings como tipos de dados complexos. No C, strings são, na verdade, **arrays de caracteres** (`char`), e por isso o tratamento é mais manual.

Como funcionam as strings em C:

Ao manipular strings em C, usamos arrays de `char`, onde a sequência de caracteres é armazenada. A particularidade é que, em C, toda string deve ser **terminada pelo caractere nulo** (`'\0'`), que indica o fim da sequência. Isso quer dizer que, mesmo que o array tenha espaço para mais caracteres, o compilador considera que a string termina quando encontra o `'\0'`.

O caractere de **nova linha** (`'\n'`) não é usado para delimitar o fim de uma string, mas para indicar uma quebra de linha no texto. Já o **caractere nulo** (`'\0'`) é o marcador real que sinaliza o fim de uma string em C. Se não houver esse caractere nulo ao final de uma string, o programa pode continuar lendo "lixo de memória" (dados não usados) até encontrar um `'\0'` acidentalmente.

Funções para manipulação de strings em C:

Em C, temos funções para **capturar dados** do teclado, como:

- `scanf`: Lê dados formatados, mas pode parar de ler strings no primeiro espaço (encontrar o /n do enter).
- `fgets`: Lê strings de maneira mais segura, incluindo espaços e limitando o número de caracteres (ele pega o /n do enter).
- `getchar`: Lê um único caractere do teclado.

Para **exibir dados na tela**, temos:

- `printf`: Imprime texto formatado.
- `putchar`: Imprime um único caractere.
- `puts`: Imprime uma string e adiciona uma nova linha automaticamente.

Caracteres especiais em C:

Ao ler e exibir dados, alguns **caracteres especiais** podem ser manipulados, como:

- `'\n'`: Indica uma nova linha (equivalente ao Enter).
- `'\0'`: Indica o fim de uma string.

Esses caracteres são lidos pelo programa e podem ser exibidos, ou usados internamente para controle, como o `'\0'`. No entanto, o `'\0'` não é impresso, pois é apenas um marcador para o compilador saber onde a string termina. Se não cuidarmos desses detalhes, podemos

encontrar comportamentos inesperados, como o **uso incorreto de** `scanf`, que pode deixar o caractere de nova linha (`'\n'`) no buffer de entrada, ou strings que exibem caracteres aleatórios (lixo de memória) se o `'\0'` não estiver presente no final.

Quebra de linhas indesejadas

Ao usar a função `fgets()`, ela inclui o caractere de nova linha (`'\n'`) na string sempre que o Enter é pressionado, caso haja espaço suficiente no buffer. Isso pode causar uma **quebra de linha extra** na hora de exibir os resultados.

Esse comportamento é normal, pois o `fgets()` captura o caractere de nova linha que foi inserido quando o usuário pressionou Enter para finalizar a entrada. Se não for tratado, esse `'\n'` pode aparecer na saída e causar quebras de linha indesejadas.

Se você quiser **remover o caractere de nova linha (`'\n'`)** após a leitura, pode fazer uma verificação simples para substituí-lo pelo caractere nulo (`'\0'`), que marca o fim da string.

getchar() e putchar()

- **getchar()**: Lê um único caractere do teclado e retorna seu valor.
- **putchar()**: Exibe um caractere na tela.

```c
#include <stdio.h>

int main() {
    char c;
    printf("Digite um caractere: ");
    c = getchar();  // Lê um caractere
    printf("Você digitou: ");
    putchar(c);     // Exibe o caractere
    return 0;
}
```

saída:

```
Digite um caractere: A
Você digitou: A
```

gets()

A função `gets()` era usada para **ler uma string** do teclado (entrada padrão) até que fosse encontrado o caractere de nova linha (`\n`) **(diferente do fgets ele não incorpora o /n)**. Ao contrário de `fgets()`, que permite limitar o número de caracteres lidos para evitar estouro de buffer, `gets()` não faz essa verificação. Isso significa que `gets()` não impede a leitura de

uma quantidade de caracteres maior do que o espaço disponível na memória, o que pode causar problemas graves de segurança.

```c
#include <stdio.h>

int main() {
    char buffer[20];  // Buffer de 20 caracteres
    printf("Digite uma string: ");
    gets(buffer);  // Função insegura
    printf("Você digitou: %s\n", buffer);
    return 0;
}
```

Por que `gets()` foi descontinuada?

`gets()` foi removida do padrão da linguagem C a partir do C11 porque **não é segura**. Ela pode causar **overflow de buffer** (quando mais dados são lidos do que o buffer pode armazenar), permitindo a escrita acidental de dados em áreas de memória que não deveriam ser modificadas. Isso abre brechas para ataques como **buffer overflow**, comuns em software vulnerável.

fgets()

fgets() é uma função usada para ler strings a partir de um arquivo ou do teclado, pertencente à biblioteca `<stdio.h>`. Diferente de **gets()**, ela é considerada mais segura porque permite limitar o número de caracteres que podem ser lidos, evitando o estouro de buffer.

Ela recebe três argumentos:

1. **str**: Um vetor (array) de caracteres que armazenará a string lida.
2. **tamanho**: O número máximo de caracteres a serem lidos (tamanho do buffer - 1, pois o último espaço é reservado para o caractere nulo `\0`).
3. **fp**: O local de onde a string será lida, que pode ser um arquivo ou o teclado (usando `stdin`).

Ao contrário de **gets()**, **fgets()** inclui o caractere de nova linha (`\n`) na string lida, a menos que o buffer seja pequeno demais para comportá-lo. Isso é importante, pois permite que a função leia entradas maiores com mais controle sobre a quantidade de dados que pode ser processada.

Uma boa prática é sempre colocar o numero de caracteres menor do que o máximo do vetor, podemos também usar o `sizeof` que pega o tamanho do vetor - 1 de forma automática

```c
#include <stdio.h>
```

```c
int main() {
    char str[50];

    printf("Digite uma string: ");
    fgets(str, 49, stdin);  /* Lê até 49 caracteres (50-1) + \n, mas é
recomendado
    colocar a menos do que o maximo*/

        printf("Digite uma string: ");
    fgets(str, sizeof(str), stdin);  // faz de forma automatica

    printf("Você digitou: %s", str);

    return 0;
}
```

saída:

```
Digite uma string: Olá!
Você digitou: Olá!
```

puts()

A função puts() escreve seu argumento string na tela, seguido por uma nova linha, ou seja, um ‘ \n ‘. Uma chamada a puts() requer bem menos tempo do que a mesma chamada a printf() porque
puts() pode escrever apenas strings de caractere, não podendo escrever números ou efetuar conversões de formato.

```c
#include <stdio.h>

int main() {
    char str[] = "Olá, mundo!";
    puts(str);  // Equivalente a printf("%s\n", str);
    return 0;
}
```

saída:

```
Olá, mundo!
```

Funções da biblioteca string.h

strlen()

Calcula o número de caracteres em uma string, sem contar o caractere nulo (`\0`), que indica o fim da string.

```c
#include <stdio.h>
#include <string.h>

int main() {
    char str[] = "Olá, mundo!";
    printf("O tamanho da string é: %lu\n", strlen(str));
    return 0;
}
```

Saída:

```
O tamanho da string é: 11
```

strcpy()

Copia o conteúdo de uma string (fonte) para outra string (destino), ou seja, o primeiro argumento é quem vai receber a string e o segundo é da onde vai ser retirado.

```c
#include <stdio.h>
#include <string.h>

int main() {
    char origem[] = "Olá, mundo!";
    char destino[20];

    strcpy(destino, origem);
    printf("String copiada: %s\n", destino);

    return 0;
}
```

Saída:

```
String copiada: Olá, mundo!
```

strcat()

strcat() copia a sequência de caracteres contida em origem para o final da string destino, e retorna a string destino, o primeiro caractere da string origem, sobrescreve o caractere ‘ \0 ‘ de destino. o primeiro argumento é quem recebe e o segundo é a fonte.

```c
#include <stdio.h>
#include <string.h>

int main() {
    char destino[30] = "christopher";
    char origem[] = " ";
    char sobrenome[] = "willians"

    strcat(destino, origem);
    strcat(destino, sobrenome);
    printf("String concatenada: %s\n", destino);

    return 0;
}
```

Saída:

```
String concatenada: christopher willians
```

strcmp()

Compara duas strings lexicograficamente (ordem alfabética). Retorna:

```
- `0` se as strings forem iguais;
- Um valor negativo se a primeira for "menor" que a segunda;
- Um valor positivo se a primeira for "maior" que a segunda.
```

```c
#include <stdio.h>
#include <string.h>

int main() {
    char str1[] = "abc";
    char str2[] = "abcd";

    int resultado = strcmp(str1, str2);

    if (resultado == 0) {
        printf("As strings são iguais.\n");
    } else if (resultado < 0) {
        printf("str1 é menor que str2.\n");
    } else {
        printf("str1 é maior que str2.\n");
    }
```

```
    return 0;
}
```

Saída:

```
str1 é menor que str2.
```

strchr()

Busca dentro de string – A função strchr() devolve a localização (endereço de memória), ou ponteiro, para a primeira ocorrência do caractere buscado (ch) dentro da string alvo. Se não encontrada, strchr() devolve um valor nulo, que é igual a zero.

```
#include <stdio.h>
#include <string.h>

int main() {
    char str[] = "Olá, mundo!";
    char *pos = strchr(str, 'm');

    if (pos != NULL) {
        printf("Caractere encontrado na posição: %ld\n", pos - str);
    } else {
        printf("Caractere não encontrado.\n");
    }

    return 0;
}
```

Saída:

```
Caractere encontrado na posição: 6
```

strstr()

Busca dentro de string – A função strstr() devolve a localização (endereço de memória), ou ponteiro, para a primeira ocorrência da string buscada dentro da string alvo. Se não encontrada, strstr() devolve um valor nulo, que é igual a zero.

```
#include <stdio.h>
#include <string.h>

int main() {
    char str[] = "Olá, mundo!";
    char *pos = strstr(str, "mundo");
```

```
    if (pos != NULL) {
        printf("Substring encontrada na posição: %ld\n", pos - str);
    } else {
        printf("Substring não encontrada.\n");
    }

    return 0;
}
```

Saída:

```
Substring encontrada na posição: 5
```

funções da biblioteca ctype.h

islower(), isupper(), toupper() e tolower()

- `islower()`:
  - Verifica se um caractere é **minúsculo** (a-z).
  - Retorna um valor diferente de zero (verdadeiro) se o caractere for minúsculo, e zero (falso) se não for.
- `toupper()`:
  - Converte um caractere **minúsculo** em **maiúsculo**.
  - Se o caractere for minúsculo, retorna sua versão maiúscula. Caso contrário, retorna o caractere sem alterações.
- `tolower()`:
  - Converte um caractere **maiúsculo** em **minúsculo**.
  - Se o caractere for maiúsculo, retorna sua versão minúscula. Caso contrário, retorna o caractere sem alterações.
- `isupper()`:
  - Verifica se um caractere é **maiúsculo** (A-Z).
  - Retorna um valor diferente de zero (verdadeiro) se o caractere for maiúsculo, e zero (falso) se não for.

```
#include <stdio.h>
#include <ctype.h>

int main() {
    char c;

    // Entrada do usuário
    printf("Digite um caractere: ");
```

```
    scanf("%c", &c);

    // Verificar se o caractere é minúsculo
    if (islower(c)) {
        printf("%c é um caractere minúsculo\n", c);
    }

    // Verificar se o caractere é maiúsculo
    if (isupper(c)) {
        printf("%c é um caractere maiúsculo\n", c);
    }

    // Converter minúsculo para maiúsculo
    printf("Maiúsculo: %c\n", toupper(c));

    // Converter maiúsculo para minúsculo
    printf("Minúsculo: %c\n", tolower(c));

    return 0;
}
```

Saída esperada:

Se o usuário digitar o caractere `a`:

```
Digite um caractere: a
a é um caractere minúsculo
Maiúsculo: A
Minúsculo: a
```

Se o usuário digitar o caractere `A`:

```
Digite um caractere: A
A é um caractere maiúsculo
Maiúsculo: A
Minúsculo: a
```

---

aula 4 - Estruturas Heterogêneas: struct

O que são Estruturas (Structs) em C?

Em C, uma `struct` é uma coleção de variáveis que podem ser de diferentes tipos, agrupadas sob um mesmo nome. Elas permitem organizar e manipular dados heterogêneos de maneira conveniente.

Sintaxe Básica

```
struct NomeDaEstrutura {
    tipo_dado1 campo1;
    tipo_dado2 campo2;
    // Outros campos
};
```

Exemplo de Declaração

```
struct Funcionario {
    char nome[50];
    int idade;
    float salario;
};
```

Aqui, a estrutura `Funcionario` contém três campos: nome, idade e salário. Cada um desses campos pode ser acessado separadamente.

Atribuição de Valores

```
struct Funcionario func1;
func1.idade = 30;
```

Os campos de uma estrutura são acessados com o operador `.`. Neste exemplo, o campo `idade` de `func1` é atribuído o valor 30.

Matrizes de Registros

É possível declarar matrizes de estruturas para armazenar múltiplos registros:

```
struct Funcionario funcionarios[5];
```

Passagem de Estruturas para Funções

Podemos passar uma estrutura inteira ou um de seus campos como parâmetro para uma função. Para modificar os valores diretamente, é possível usar ponteiros:

```
void atualizarIdade(struct Funcionario *f, int novaIdade) {
    f->idade = novaIdade;
```

```
}
```

No exemplo acima, a função recebe um ponteiro para a estrutura e atualiza o campo `idade`.

---

aula 5 - Ponteiros

O que são Ponteiros?

Ponteiros são variáveis que armazenam o endereço de outras variáveis na memória. Eles são úteis para acessar e manipular diretamente os valores armazenados nos endereços de memória.

Declaração de Ponteiros

A sintaxe básica para declarar um ponteiro é:

```
tipo_dado *nome_ponteiro;
```

Exemplo:

```
int *p;
```

Aqui, `p` é um ponteiro para um `int`.

Operador `&` e Operador `*`

- O operador `&` retorna o endereço de uma variável.
- O operador `*` acessa o valor armazenado no endereço apontado pelo ponteiro.

Exemplo:

```
int valor = 10;
int *p = &valor;
printf("%d", *p);  // Imprime 10
```

Neste exemplo, `p` armazena o endereço de `valor` e `*p` acessa o valor 10.

Ponteiro para Ponteiro

Um ponteiro pode apontar para outro ponteiro, criando múltiplos níveis de indireção:

```
int **p2;
```

Aqui, `p2` é um ponteiro para outro ponteiro que, por sua vez, aponta para um `int`.

Uso com Arrays

Ponteiros podem ser usados para manipular arrays de forma eficiente, pois um array é essencialmente um ponteiro para seu primeiro elemento.

```
int vet[5] = {1, 2, 3, 4, 5};
int *p = vet;
```

Aqui, `p` aponta para o primeiro elemento de `vet`. Podemos acessar os elementos do array usando aritmética de ponteiros.

Funções e Ponteiros

Ponteiros são frequentemente usados para passar variáveis para funções de modo que as modificações feitas dentro da função reflitam fora dela.

```
void incrementar(int *p) {
    (*p)++;
}
```

No exemplo acima, a função `incrementar` altera diretamente o valor da variável apontada pelo ponteiro.

---

aula 6 - Manipulação de Arquivos em C

Ponteiro `FILE`

O ponteiro `FILE` é uma estrutura especial usada pela linguagem C para manipular arquivos. Ele armazena informações como a posição do cursor e o estado do arquivo (leitura, escrita, etc.).

Exemplo:

```
FILE *arquivo;
```

Esse ponteiro é necessário para operações como leitura e escrita de arquivos.

Tipos de arquivos e modos de leitura

Temos duas extensões de arquivo o `.txt e o .bin` (arquivo de texto e binário respectivamente), e para ambos temos diferentes modos de abertura para a execução de

diferentes ações

Extensão txt

- modo `r` - serve somente para a leitura do texto, se faz necessário que já tenha um arquivo de texto.
- modo `w` - serve para escrita, ele cria o arquivo se não existir e substitui se houver um.
- modo `a` - serve para adicionar coisas no final do arquivo, se faz necessário que já tenha um arquivo de texto.

Extensão bin

- modo `rb` - serve somente para a leitura do texto, se faz necessário que já tenha um arquivo de binário.
- modo `wb` - serve para escrita, ele cria o arquivo se não existir e substitui se houver um.
- modo `ab` - serve para adicionar coisas no final do arquivo, se faz necessário que já tenha um arquivo binário.

Funções de Manipulação de Arquivos

arquivo de texto

```c
#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

#include <locale.h>

#include <string.h>


void salvarTexto(char texto[]);

void salvarTextoMaiusculo(char texto[]);

void imprimirArquivo(char nomeArquivo[]);



int main() {

    setlocale(LC_ALL, "Portuguese");
```

```c
    char texto[1000];

    printf("Digite um pequeno texto: ");

    fgets(texto, sizeof(texto), stdin);


    salvarTexto(texto);


    salvarTextoMaiusculo(texto);


    printf("\nConteúdo de arq1.txt:\n");

    imprimirArquivo("arq1.txt");

    printf("\nConteúdo de arq2.txt:\n");

    imprimirArquivo("arq2.txt");

    return 0;

}

void salvarTexto(char texto[]) {

    FILE *arquivo = fopen("arq1.txt", "w");

    if (arquivo == NULL) {

        printf("Erro ao abrir o arquivo %s\n", "arq1.txt");

        exit(1);

    }

    for(int i = 0; i < strlen(texto); i++) {

        fputc(texto[i], arquivo);

    }

    fclose(arquivo);

}

void salvarTextoMaiusculo(char texto[]) {

    FILE *arquivo = fopen("arq2.txt", "w");
```

```
    if (arquivo == NULL) {
        printf("Erro ao abrir o arquivo %s\n", "arq2.txt");
        exit(1);
    }
    for(int i = 0; i < strlen(texto); i++) {
        fputc(toupper(texto[i]), arquivo);
    }
    fclose(arquivo);
}

void imprimirArquivo(char nomeArquivo[]) {
    FILE *arquivo = fopen(nomeArquivo, "r");
    if (arquivo == NULL) {
        printf("Erro ao abrir o arquivo %s\n", nomeArquivo);
        exit(1);
    }
    char c= fgetc(arquivo);
    while(c != EOF) {
        printf("%c", c);
        c = fgetc(arquivo);
    }
    fclose(arquivo);
}
```

fopen()

Abre um arquivo em um modo específico (leitura, escrita, etc.). Retorna um ponteiro do tipo `FILE`, ou `NULL` em caso de erro.

```
FILE *fp = fopen("arquivo.txt", "r");
if (fp == NULL) {
    printf("Erro ao abrir o arquivo\n");
}
```

`fclose()`

Fecha um arquivo aberto. Garante que os dados no buffer sejam gravados e libera o arquivo.

```
fclose(fp);
```

`fputc()`

Escreve um caractere no arquivo, devemos colocar como primeiro argumento o que vai ser escrito e como segundo argumento o arquivo que vai ser alocado. Retorna o caractere escrito ou `EOF` em caso de erro.

```
fputc('A', fp);
```

`fgetc()`

Lê um caractere do arquivo, toda vez que ele é chamado ele passa para o caractere do lado. Retorna o caractere lido ou `EOF` quando o fim do arquivo é atingido.

```
char ch = fgetc(fp);
printf("Caractere lido: %c\n", ch);
```

`fputs()`

Escreve uma string no arquivo, devemos colocar como primeiro argumento o que vai ser escrito e como segundo argumento o arquivo que vai ser alocado. Não adiciona automaticamente uma nova linha no final da string.

```
fputs("Olá, mundo\n", fp);
```

`fgets()`

Lê uma string do arquivo até encontrar uma nova linha ou o final do arquivo. O primeiro argumento é onde será armazenado o conteúdo lido, o segundo é o tamanho de onde é armazenado e o terceiro é da onde estamos pegando.

```c
char buffer[100];
fgets(buffer, 100, fp);
printf("String lida: %s\n", buffer);
```

`fprintf()`

A função `fprintf()` é usada para **escrever dados em um arquivo**, de forma formatada, como fazemos com `printf()` para escrever na tela. Ela permite que você escreva diferentes tipos de dados (números, strings, etc.) no arquivo, mas **em formato de texto**.

```c
FILE *fp = fopen("arquivo.txt", "w");  // Abre o arquivo para escrita
if (fp != NULL) {
    int idade = 25;
    fprintf(fp, "Idade: %d\n", idade);  // Escreve "Idade: 25" no arquivo
    fclose(fp);  // Fecha o arquivo
}
```

O que acontece?

- O número `25` (inteiro) é **convertido para texto** e escrito no arquivo como `"Idade: 25\n"`.
- O arquivo de texto conterá a string **"Idade: 25"**, legível por qualquer editor de texto.

`fscanf()`

A função `fscanf()` é usada para **ler dados de um arquivo**, também de forma formatada, semelhante a `scanf()` que lê da entrada do teclado. Ela lê dados como strings e números a partir de um arquivo de **texto** e converte esses dados de volta para variáveis.

```c
FILE *fp = fopen("arquivo.txt", "r");  // Abre o arquivo para leitura
if (fp != NULL) {
    int idade;
    fscanf(fp, "Idade: %d", &idade);  // Lê o texto e converte o número para a variável 'idade'
    printf("Idade lida: %d\n", idade);  // Exibe a idade lida do arquivo
    fclose(fp);  // Fecha o arquivo
}
```

O que acontece?

- O `fscanf()` lê o texto `"Idade: 25"` do arquivo.
- Ele converte o valor `25` para um **número inteiro** e o armazena na variável `idade`.
- A leitura só acontece porque o dado no arquivo está formatado corretamente, de acordo com o que foi esperado pelo `fscanf()`.

Como o `fscanf()` consegue ler as varivaeis?

Quando você lê um arquivo de texto usando fscanf(), a função depende do formato dos dados no arquivo para saber como interpretar cada valor e armazená-lo nas variáveis corretas. A forma como você especifica o formato na string de formato de fscanf() é essencial para que ele saiba qual variável deve guardar qual valor.

A string de formato que você passa para fscanf() deve corresponder à estrutura e ao conteúdo dos dados no arquivo. Vamos explicar isso com mais detalhes.

Exemplo Simples: Lendo dados formatados

Imagine que você tem um arquivo de texto com o seguinte conteúdo:

```
Nome: João
Idade: 30
Altura: 1.75
```

Agora, você quer ler essas informações e armazená-las nas variáveis corretas: uma string para o nome, um inteiro para a idade e um float para a altura.

Passo 1: Formato dos dados

Você sabe que o arquivo de texto tem o seguinte formato:

- A palavra "Nome:" seguida por um nome.
- A palavra "Idade:" seguida por um número inteiro.
- A palavra "Altura:" seguida por um número decimal (float).

Passo 2: `fscanf()` com string de formato

Para ler os dados corretamente, você usaria `fscanf()` com uma string de formato que reflete a estrutura do arquivo. Aqui está um exemplo de como fazer isso:

```c
#include <stdio.h>

int main() {
    FILE *fp = fopen("dados.txt", "r");  // Abrindo o arquivo para leitura
    if (fp != NULL) {
        char nome[100];
        int idade;
        float altura;

        // Lendo os dados do arquivo e armazenando nas variáveis corretas
        fscanf(fp, "Nome: %s\n", nome);         // Lê o nome e armazena na
variável 'nome'
        fscanf(fp, "Idade: %d\n", &idade);      // Lê a idade e armazena na
```

```
variável 'idade'
        fscanf(fp, "Altura: %f\n", &altura);    // Lê a altura e armazena na
variável 'altura'

        // Exibindo os valores lidos
        printf("Nome: %s\n", nome);
        printf("Idade: %d\n", idade);
        printf("Altura: %.2f\n", altura);

        fclose(fp);  // Fechando o arquivo
    } else {
        printf("Erro ao abrir o arquivo\n");
    }

    return 0;
}
```

Como `fscanf()` sabe qual variável usar:

- A string de formato `"Nome: %s\n"` indica que o primeiro valor esperado é uma **string** (`%s`). O `fscanf()` vai procurar um valor de texto após a palavra "Nome:" e armazená-lo na variável `nome`.
- A string `"Idade: %d\n"` indica que o próximo valor é um **inteiro** (`%d`). `fscanf()` vai procurar um número após "Idade:" e armazená-lo na variável `idade`.
- A string `"Altura: %f\n"` indica que o último valor esperado é um **float** (`%f`). `fscanf()` vai procurar um número decimal após "Altura:" e armazená-lo na variável `altura`.

Importante:

- **Ordem dos dados**: O `fscanf()` lê os dados na ordem em que eles estão no arquivo. Se os dados no arquivo não estiverem no formato ou ordem esperada, o `fscanf()` pode falhar ou armazenar valores incorretos.
- **Correspondência exata**: A string de formato passada para `fscanf()` deve **corresponder exatamente** ao formato dos dados no arquivo. Por exemplo, se o arquivo contiver `"Idade:"` e a string de formato espera `"Age:"`, a leitura falhará.

Exemplo de Arquivo:

Se o arquivo `dados.txt` contiver:

```
Nome: João
Idade: 30
Altura: 1.75
```

Esse código armazenará:

- `"João"` na variável `nome`
- `30` na variável `idade`
- `1.75` na variável `altura`

```
João 30 1.75
```

Você pode usar o seguinte código para ler esses valores de uma vez:

```
fscanf(fp, "%s %d %f", nome, &idade, &altura);
```

Isso vai ler:

- O nome "João" para a variável `nome` (string).
- A idade "30" para a variável `idade` (inteiro).
- A altura "1.75" para a variável `altura` (float).

Conclusão:

O `fscanf()` sabe qual variável deve armazenar os valores com base no **formato que você especifica** na string de formato. Ele lê o arquivo de acordo com a estrutura que você define na string de formato (`%s`, `%d`, `%f`, etc.), e é responsabilidade do programador garantir que o arquivo esteja formatado corretamente para que a leitura funcione como esperado.

Se o arquivo não estiver formatado conforme o que o `fscanf()` espera, ele não conseguirá interpretar corretamente os dados.

Outros exemplos:

Lendo múltiplos valores em uma linha:

Se o arquivo tiver múltiplos valores em uma única linha, como:

Arquivo binário

```
#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

#include <locale.h>

#include <string.h>
```

```c
typedef struct Criafuncionario{

    char nome[100];

    char telefone[15];

    char email[100];

} Funcionario;



void gravarVetor(Funcionario *funcionarios, int tamanho);


int main (){


    setlocale(LC_ALL, "Portuguese");


    Funcionario funcionarios[5];


    for (int i = 0; i < 5; i++) {

        printf("Digite o nome do funcionário %d: ", i + 1);

        fgets(funcionarios[i].nome, sizeof(funcionarios[i].nome), stdin);

        funcionarios[i].nome[strcspn(funcionarios[i].nome, "\n")] = '\0'; //
Remove o \n

        printf("Digite o telefone do funcionário %d: ", i + 1);

        fgets(funcionarios[i].telefone, sizeof(funcionarios[i].telefone),
stdin);

        funcionarios[i].telefone[strcspn(funcionarios[i].telefone, "\n")] =
'\0'; // Remove o \n

        printf("Digite o email do funcionário %d: ", i + 1);

        fgets(funcionarios[i].email, sizeof(funcionarios[i].email), stdin);

        funcionarios[i].email[strcspn(funcionarios[i].email, "\n")] = '\0'; //
Remove o \n

    }
```

```
    gravarVetor(funcionarios, 5);
}


void gravarVetor(Funcionario *funcionarios, int tamanho) {
    FILE *arquivo = fopen("funcionarios.bin", "wb");
    if (arquivo == NULL) {
        printf("Erro ao abrir o arquivo\n");
        exit(1);
    }
    fwrite(funcionarios, sizeof(Funcionario), tamanho, arquivo);

    fclose(arquivo);
}
```

arquivo 2

```
#include <stdio.h>
#include <stdlib.h>

typedef struct {
    char nome[100];
    char telefone[15];
    char email[100];
} Funcionario;


int main() {
    FILE *arquivo = fopen("funcionarios.bin", "rb");
```

```
    if (arquivo == NULL) {
        printf("Erro ao abrir o arquivo\n");
        return 1;
    }

    Funcionario terceiroFuncionario;

    fseek(arquivo, 2 * sizeof(Funcionario), SEEK_SET);

    fread(&terceiroFuncionario, sizeof(Funcionario), 1, arquivo);
    fclose(arquivo);

    printf("Terceiro Funcionário:\n");
    printf("Nome: %s\n", terceiroFuncionario.nome);
    printf("Telefone: %s\n", terceiroFuncionario.telefone);
    printf("Email: %s\n", terceiroFuncionario.email);

    return 0;
}
```

`fwrite()`

Escreve blocos de bytes em um arquivo, como primeiro argumento passamos o que vai ser escrito (necessita ser um ponteiro a variável, caso ao contrario devemos passar o endereço de memória dela), como segundo argumento passamos o tamanho padrão da estrutura que estamos lendo, como terceiro argumento passamos a quantidade dessa estrutura que vamos alocar (exemplo, se tivermos um array de 3 struct, podemos alocar apenas duas) e como quarto argumento passamos onde ele vai ser escrito. Utilizada com arquivos binários para gravar inteiros, floats, structs, etc.

```
int num = 12345;
fwrite(&num, sizeof(int), 1, fp);
```

`fread()`

Lê blocos de bytes de um arquivo, ideal para arquivos binários, , como primeiro argumento passamos o que vai ser lida (necessita ser um ponteiro a variável, caso ao contrario devemos passar o endereço de memória dela), como segundo argumento passamos o tamanho padrão da estrutura que estamos lendo, como terceiro argumento passamos a quantidade dessa estrutura que vamos ler (exemplo, se tivermos um array de 3 struct, podemos ler apenas duas) e como quarto argumento passamos onde ele vai ser lida.

```c
int num;
fread(&num, sizeof(int), 1, fp);
printf("Número lido: %d\n", num);
```

`fseek()`

Move o ponteiro do arquivo para uma posição específica, permitindo leitura e escrita não sequenciais. De maneira geral, passamos como seu primeiro argumento o arquivo que vai ser lido, como o segundo argumento passamos o tamanho da linha que vamos avançar (exemplo, se tivermos varias strings, uma em cada linha, podemos usar o tamanho delas para pular) e o terceiro argumento é de onde vamos partir. **Temos 3 modos :**

- `SEEK_SET`, ele começa do inicio do arquivo, temos que passar como o segundo argumento o tamanho de bytes que vamos avançar.
- `SEEK_CUR`, ele começa do ponto atual do arquivo, temos que passar como o segundo argumento o tamanho de bytes que vamos avançar.
- `SEEK_END`, ele começa do fim do arquivo, temos que passar como o segundo argumento o tamanho de bytes que vamos avançar.

Constante	Valor	Significado
SEEK_SET	0	Início do arquivo
SEEK_CUR	1	Ponto atual no arquivo
SEEK_END	2	Fim do arquivo

```c
fseek(fp, 0, SEEK_END);  // Move o ponteiro para o fim do arquivo

typedef struct {

    char nome[100];

    char telefone[15];

    char email[100];

} Funcionario;
```

```
fseek(arquivo, 2 * sizeof(Funcionario), SEEK_SET);/* pula duas vezes o tamanho da struct funcionario e parti do começo do arquivo, assim Move o ponteiro para o 3º funcionário*/
```

`rewind()`

Move o ponteiro do arquivo para o início.

```
rewind(fp);
```

`feof()`

Verifica se o final do arquivo foi atingido, ele veio como solução para verificar o E0F que indica o fim do arquivo.

```
while (!feof(fp)) {
    char ch = fgetc(fp);
    printf("%c", ch);
}
```

---

aula 7 - Alocação de memoria

A linguagem C permite alocar (reservar) dinamicamente (em tempo de execução) blocos de memória utilizando ponteiros. A esse processo dá-se o nome alocação dinâmica. A alocação dinâmica permite ao programador “criar” vetores ou arrays em tempo de execução, ou seja, alocar memória para novos arrays quando o programa está sendo executado, e não apenas quando se está escrevendo o programa.

Essa estratégia é utilizada quando não se sabe ao certo quanto de memória será necessário para armazenar os dados com que se quer trabalhar. Desse modo, pode-se definir o tamanho do vetor ou array em tempo de execução, evitando assim o desperdício de memória.

Então a alocação dinâmica de memória:

- Reserva um bloco consecutivo de bytes na memória e retorna o endereço inicial deste bloco;
- Permite escrever programas mais flexíveis;
- Poupa memória ao evitar a alocação de grandes espaços de memória que só serão liberados quando o programa terminar.

Tamanho dos tipos de dados:

- Tipos diferentes, podem ter tamanhos diferentes na memória, portanto, alocar memória do tipo int é diferente de alocar memória do tipo char, ou mesmo o tipo de dado struct que você criou:

Tipo	Tamanho
char	1 byte
int	4 bytes
float	4 bytes
double	8 bytes
struct	?? Bytes

função malloc()

A função `malloc()` aloca dinamicamente um bloco de memória durante a execução do programa. Ela faz o pedido de memória ao sistema operacional e retorna um ponteiro genérico com o endereço do início do espaço de memória alocado.

Tipos de Ponteiros

Os ponteiros são variáveis que armazenam o endereço de outras variáveis na memória. Dependendo do nível de referência, podemos ter diferentes tipos de ponteiros:

1. Ponteiro para variável

- **Sintaxe:** `int *p;`

Um **ponteiro de nível 1** armazena o endereço de uma variável do tipo `int`. Ele aponta diretamente para um local na memória onde um valor inteiro é armazenado.

Exemplo:

```
int valor = 10;
int *p = &valor; // 'p' aponta para o endereço de 'valor'
```

Aqui, `p` é um ponteiro que armazena o endereço da variável `valor` e, através de `*p`, podemos acessar o valor 10.

2. Ponteiro para ponteiro

- **Sintaxe:** `int **p;`

Um **ponteiro de nível 2** armazena o endereço de outro ponteiro, que, por sua vez, aponta para uma variável. Esse tipo de ponteiro é usado quando queremos manipular indiretamente o valor de um ponteiro.

Exemplo:

```
int valor = 10;
int *p1 = &valor;   // 'p1' aponta para 'valor'
int **p2 = &p1;     // 'p2' aponta para 'p1'
```

Aqui, `p2` armazena o endereço de `p1`, que aponta para a variável `valor`. Podemos acessar o valor de `valor` usando `**p2`.

3. Ponteiro para ponteiro para ponteiro

- **Sintaxe:** `int ***p;`

Um **ponteiro de nível 3** armazena o endereço de um ponteiro de nível 2, que, por sua vez, armazena o endereço de um ponteiro de nível 1, o qual finalmente aponta para uma variável. Esse nível de indireção raramente é necessário, mas pode ser útil em casos complexos, como estruturas de dados e algoritmos recursivos.

Exemplo:

```
int valor = 10;
int *p1 = &valor;   // 'p1' aponta para 'valor'
int **p2 = &p1;     // 'p2' aponta para 'p1'
int ***p3 = &p2;    // 'p3' aponta para 'p2'
```

Nesse exemplo, `p3` armazena o endereço de `p2`, que armazena o endereço de `p1`, que por sua vez armazena o endereço de `valor`. Podemos acessar o valor de `valor` usando `***p3`.

Resumo:

- `int *p;` → Ponteiro que aponta diretamente para um endereço de memória onde um valor `int` está armazenado.
- `int **p;` → Ponteiro que aponta para outro ponteiro, que por sua vez aponta para um valor `int`.
- `int ***p;` → Ponteiro que aponta para um ponteiro de ponteiro, que por sua vez aponta para um valor `int`.

Sintaxe:

Para utilizar a função `malloc()`, ela deve ser atribuída a um ponteiro. A função retorna um ponteiro genérico do tipo `void *`, que precisa ser convertido para o tipo de dado desejado.

Isso significa que devemos transformar esse ponteiro genérico para o tipo de dado que queremos manipular. Por exemplo, se quisermos criar um vetor de inteiros, o ponteiro que recebe a atribuição deve ser do tipo `int *`.

exemplo:

```
int *vet = (int*) malloc(3 * sizeof(int));
```

Nesse exemplo, estamos alocando dinamicamente memória suficiente para armazenar 3 inteiros, e o ponteiro `vet` apontará para o início desse bloco de memória.

Explicação detalhada:

- **Conversão do ponteiro genérico:** O ponteiro retornado por `malloc()` é do tipo `void *` (um ponteiro genérico). Para usá-lo como um ponteiro para inteiros, fazemos um *casting* para o tipo desejado, neste caso `(int *)`.
- **Tamanho da memória alocada:** A função `malloc()` recebe como argumento o número de bytes a serem alocados. No exemplo, calculamos esse valor multiplicando a quantidade de elementos (3) pelo tamanho de cada elemento (`sizeof(int)`), o que garante que o tamanho da memória seja apropriado para armazenar 3 inteiros.

Criando matrizes com malloc()

Para criar uma matriz usando a função malloc(), precisamos alocar memória dinamicamente para as linhas e colunas. Isso envolve a criação de um ponteiro para ponteiro, ou seja, um ponteiro que aponta para outros ponteiros, que por sua vez apontam para as colunas da matriz. A ideia é criar um vetor de ponteiros (para as linhas) e, em seguida, alocar memória para cada linha (as colunas da matriz).

Passos:

1. **Criar um ponteiro de ponteiros**: Começamos criando um ponteiro do tipo `int **` para armazenar o endereço das linhas da matriz.
2. **Alocar memória para as linhas**: Usamos `malloc()` para alocar memória para o vetor de ponteiros (linhas da matriz).
3. **Alocar memória para as colunas**: Em seguida, para cada linha, alocamos um vetor de inteiros (as colunas) usando outro `malloc()`.

Exemplo de código:

```
int **matriz;

// Alocando memória para 5 ponteiros (linhas)
matriz = (int**) malloc(5 * sizeof(int*));
```

```
for (int i = 0; i < 5; i++) {
    // Para cada linha, alocamos memória para 5 inteiros (colunas)
    matriz[i] = (int*) malloc(5 * sizeof(int));
}
```

Explicação detalhada

Declaração de `matriz`:

```
int **matriz;
```

Aqui, `matriz` é um ponteiro que apontará para outros ponteiros, ou seja, um "vetor de ponteiros".

Alocação de memória para as linhas:

```
matriz = (int**) malloc(5 * sizeof(int*));
```

Estamos alocando memória para 5 ponteiros, cada um representando uma linha da matriz. O `sizeof(int*)` garante que estamos alocando o espaço adequado para armazenar o endereço de um ponteiro para inteiro.

Alocação de memória para as colunas:

```
for (int i = 0; i < 5; i++) {
    matriz[i] = (int*) malloc(5 * sizeof(int));
}
```

Aqui, percorremos cada linha da matriz (`matriz[i]`) e alocamos memória suficiente para 5 inteiros em cada uma delas. Cada linha da matriz pode ser vista como um vetor de inteiros.

Observação:

Se você quiser uma matriz de tamanho diferente (por exemplo, 5 linhas e 3 colunas), basta modificar o valor no `malloc()` que aloca as colunas:

```
matriz[i] = (int*) malloc(3 * sizeof(int));
```

Funções `free`, `calloc` e `realloc` em C

No C, quando alocamos memória dinamicamente usando funções como `malloc()`, precisamos gerenciar essa memória corretamente. As funções `free()`, `calloc()` e `realloc()` desempenham papéis importantes nesse gerenciamento. Vamos entender como e por que usá-las.

`free()`

A função `free()` é usada para liberar a memória alocada dinamicamente. Quando usamos `malloc()`, `calloc()` ou `realloc()`, o sistema operacional reserva um espaço na memória, mas precisamos liberar esse espaço após o uso para evitar vazamento de memória.

Por que usar `free()`?

- Libera a memória que não será mais usada, evitando desperdício de recursos.
- Ajuda a prevenir "memory leaks" (vazamento de memória), que ocorrem quando a memória é alocada, mas nunca liberada.

Sintaxe:

```
free(ptr);
```

Aqui, `ptr` é o ponteiro que aponta para o bloco de memória que foi alocado dinamicamente.

Exemplo simples:

```
int *p = (int*) malloc(5 * sizeof(int));

// ... uso do ponteiro 'p'

free(p); // libera a memória alocada
```

Exemplo com matriz:

Ao liberar a memória de uma matriz alocada dinamicamente, precisamos liberar cada linha da matriz antes de liberar o ponteiro que aponta para as linhas.

```
int **matriz;

// Alocação da matriz 5x5
matriz = (int**) malloc(5 * sizeof(int*));
for (int i = 0; i < 5; i++) {
    matriz[i] = (int*) malloc(5 * sizeof(int));
}

// ... uso da matriz

// Liberação da memória
for (int i = 0; i < 5; i++) {
    free(matriz[i]); // libera cada linha
```

```
}
free(matriz); // libera o ponteiro das linhas
```

`calloc()`

A função `calloc()` também é usada para alocar memória dinamicamente, mas diferente de `malloc()`, ela inicializa todos os bits da memória alocada com zero. Isso é útil quando você precisa garantir que a memória alocada seja inicializada.

Por que usar `calloc()`?

- Garante que toda a memória alocada seja zerada.
- Pode evitar erros causados por lixo de memória.

Sintaxe:

```
ptr = (tipo_dado*) calloc(n, sizeof(tipo_dado));
```

Aqui, `n` é o número de elementos, e `tipo_dado` é o tipo de dado que será armazenado.

Exemplo simples:

```
int *p = (int*) calloc(5, sizeof(int));

// ... uso do ponteiro 'p'

free(p); // libera a memória alocada
```

`realloc()`

A função `realloc()` é usada para redimensionar um bloco de memória alocado dinamicamente. Se você precisar aumentar ou diminuir o tamanho de um bloco de memória, `realloc()` será útil.

Por que usar `realloc()`?

- Permite expandir ou reduzir a memória alocada sem perder os dados atuais.
- Evita a necessidade de alocar um novo bloco e copiar manualmente os dados.

Sintaxe:

```
ptr = realloc(ptr, novo_tamanho);
```

Aqui, `novo_tamanho` é o novo tamanho que você deseja alocar.

Exemplo simples:

```c
int *p = (int*) malloc(5 * sizeof(int));

// Redimensiona a memória para armazenar 10 inteiros
p = (int*) realloc(p, 10 * sizeof(int));

// ... uso do ponteiro 'p'

free(p); // libera a memória alocada
```

Essas três funções — `free()`, `calloc()` e `realloc()` — são essenciais para o gerenciamento de memória em C, permitindo que o programa aloque, redimensione e libere memória conforme necessário.

---

aula 8 - Recursividade

O que é Recursividade?

Recursividade é uma técnica onde uma função chama a si mesma para resolver um problema. O problema é dividido em subproblemas menores até que se atinja um caso base, onde o processo recursivo termina. Esse processo é frequentemente usado para resolver problemas que podem ser naturalmente decompostos, como a função fatorial e o jogo das Torres de Hanói.

Exemplo Simples: Cálculo de Fatorial

A fórmula para calcular o fatorial de um número é:

[
n! = n \times (n-1)!
]

Em termos recursivos, o fatorial de um número pode ser definido como:

```c
int fatorial(int n) {
    if (n == 0)
        return 1;  // Caso base
    else
        return n * fatorial(n - 1);  // Chamada recursiva
}
```

Neste exemplo, a função `fatorial` chama a si mesma para calcular o fatorial de `n-1` até que `n` seja 0, o caso base, onde a função para de se chamar.

Condição de Parada

Uma função recursiva deve sempre ter uma **condição de parada**. Sem essa condição, a função continuaria chamando a si mesma indefinidamente, o que resultaria em um erro de execução por falta de memória.

Torres de Hanói

Um exemplo clássico de recursividade é o problema das **Torres de Hanói**. O objetivo é mover uma pilha de discos de um pino para outro, respeitando as seguintes regras:

1. Apenas um disco pode ser movido por vez.
2. Um disco maior nunca pode ser colocado sobre um disco menor.

A solução recursiva do problema envolve mover os `n-1` discos para um pino intermediário, mover o maior disco diretamente para o destino e, finalmente, mover os `n-1` discos do pino intermediário para o destino.

Exemplo de Código

```
void torresHanoi(int n, char origem, char destino, char trabalho) {
    if (n == 1) {
        printf("Mova o disco 1 da origem %c para o destino %c\n", origem,
destino);
    } else {
        torresHanoi(n-1, origem, trabalho, destino);
        printf("Mova o disco %d da origem %c para o destino %c\n", n, origem,
destino);
        torresHanoi(n-1, trabalho, destino, origem);
    }
}
```

Tipos de Recursão

Existem dois tipos principais de recursão:

- **Recursão direta**: A função chama a si mesma diretamente.
- **Recursão indireta**: A função chama outra função, que eventualmente chama a função original.

Recursão Direta

Um exemplo de recursão direta é o cálculo do fatorial.

Recursão Indireta

No caso de recursão indireta, a função A chama a função B, que chama novamente a função A. Essa técnica é menos comum, mas pode ser útil em certos contextos.

Vantagens e Desvantagens da Recursividade

Vantagens

- **Simplicidade**: A solução recursiva pode ser mais intuitiva e fácil de entender.
- **Elegância**: Soluções recursivas são geralmente mais elegantes e concisas do que suas contrapartes iterativas.

Desvantagens

- **Desempenho**: Recursividade pode ser mais lenta devido ao tempo necessário para salvar o estado de cada chamada de função.
- **Uso de Memória**: Grandes recursões podem consumir muita memória, pois cada chamada de função ocupa espaço na pilha de execução.

PROJETOS

AULA 2

```c
#include <stdio.h>



void calculoVantagens(float numeroHoras, float salarioHora, int numeroFilhos,
float valorPorFilho, float *salarioBruto, float *salarioFamilia, float
*vantagens);

void calculoDeducoes(float salarioBruto, float taxaIR, float *INSS, float
*IRPF, float *Deducoes);
```

```c
int main(){

    float numeroHoras, salarioHora, valorPorFilho, taxaIR;

    int numeroFilhos;

    float salarioBruto, salarioFamilia, vantagens, INSS, IRPF, Deducoes;



    printf("Digite o numero de horas trabalhadas: ");

    scanf(" %f", &numeroHoras);

    printf("Digite o salario por hora: ");

    scanf(" %f", &salarioHora);

    printf("Digite o numero de filhos: ");

    scanf(" %d", &numeroFilhos);

    printf("Digite o valor por filho: ");

    scanf(" %f", &valorPorFilho);

    printf("Digite a taxa de IR: ");

    scanf(" %f", &taxaIR);



    calculoVantagens(numeroHoras, salarioHora, numeroFilhos, valorPorFilho,
&salarioBruto, &salarioFamilia, &vantagens);

    calculoDeducoes(salarioBruto, taxaIR, &INSS, &IRPF, &Deducoes);



    printf("\n\nSalario Bruto: %.2f\n", salarioBruto);

    printf("Salario Familia: %.2f\n", salarioFamilia);

    printf("Vantagens: %.2f\n", vantagens);

    printf("INSS: %.2f\n", INSS);

    printf("IRPF: %.2f\n", IRPF);

    printf("Deducoes: %.2f\n", Deducoes);
```

```
    return 0;

}



void calculoVantagens(float numeroHoras, float salarioHora, int numeroFilhos,
float valorPorFilho, float *salarioBruto, float *salarioFamilia, float
*vantagens){

    *salarioBruto = numeroHoras * salarioHora;

    *salarioFamilia = numeroFilhos * valorPorFilho;

    *vantagens = *salarioBruto + *salarioFamilia;

}



void calculoDeducoes(float salarioBruto, float taxaIR, float *INSS, float
*IRPF, float *Deducoes){

    *INSS = salarioBruto * 0.08;

    *IRPF = salarioBruto * taxaIR;

    *Deducoes = *INSS + *IRPF;

}
```

AULA 3

concatenar strings

```
#include <stdio.h>
#include <string.h>



int main()

{

    char nome[50], sobrenome[50], espaco[10] = {" "};
```

```
    printf("digite o seu primeiro nome: ");
    fgets(nome, sizeof(nome), stdin);
    printf("digite o seu sobrenome: ");
    fgets(sobrenome, sizeof(sobrenome), stdin);
     strtok(nome, "\n");
      strtok(sobrenome, "\n");
    strcat(nome, espaco);
    strcat(nome, sobrenome);
    printf(" seu nome e: %s", nome);
    printf("\n\n");
    system("pause");
    return 0;
}
```

substituir as vogais

```
#include <stdio.h>
#include <string.h>

int main() {
    char string[100], consoante;
    int i, cont_vogais = 0;

    printf("Digite uma string: ");
    fgets(string, 100, stdin);
```

```
    printf("Digite uma consoante: ");
    scanf(" %c", &consoante);

    for (i = 0; string[i] != '\0'; i++) {
        if (string[i] == 'a' || string[i] == 'e' || string[i] == 'i' ||
            string[i] == 'o' || string[i] == 'u' || string[i] == 'A' ||
            string[i] == 'E' || string[i] == 'I' || string[i] == 'O' ||
            string[i] == 'U') {
            cont_vogais++;
            string[i] = consoante;
        }
    }

    printf("Número total de vogais: %d\n", cont_vogais);
    printf("Nova string: %s\n", string);
    return 0;
}
```

verificar se uma string esta dentro da outra

```
#include <stdio.h>
#include <string.h>

int main()
{
    char nome[50], sobrenome[50];
```

```
    char *p;



    printf("digite o seu um nome: ");

    fgets(nome, sizeof(nome), stdin);

    printf("\ndigite o seu outro nome: ");

    fgets(sobrenome, sizeof(sobrenome), stdin);

    strtok(nome, "\n");

    strtok(sobrenome, "\n");

    if(strstr(nome, sobrenome) != NULL){

        p = strstr(nome, sobrenome);

        printf("\numa string esta contida na outra, local de memoria: %p",
&p);

    } else{

        printf("\na primeira string não ta contida na segunda");

    }

    system("pause");

    return 0;

}
```

```
#include <stdio.h>

#include <stdlib.h>

#include <locale.h>



int main() {

    char str[100];
```

```c
    int i = 0;

    setlocale(LC_ALL, "Portuguese");


    printf("Digite uma string: ");

    fgets(str, sizeof(str), stdin);



    while (str[i] != '\0') {


        if (str[i] >= 'a' && str[i] <= 'z') {

            str[i] = str[i] - 32;

        }

        i++;

    }


    printf("String convertida: %s\n", str);


    return 0;

}
```

```c
#include <stdio.h>

#include <ctype.h>

#include <string.h>


void invert_words(char *str) {

    char temp[100];
```

```
    int j = 0, len = strlen(str);

    for (int i = len - 1; i >= 0; i--) {
        if (str[i] == ' ' || str[i] == '\n') {
            temp[j++] = str[i];
        } else {
            int start = i;
            while (i >= 0 && str[i] != ' ' && str[i] != '\n') {
                i--;
            }
            for (int k = start; k > i; k--) {
                temp[j++] = str[k];
            }
        }
    }
    temp[j] = '\0';
    strcpy(str, temp);
}

int main() {
    char string[100];

    printf("Digite uma string: ");
    fgets(string, sizeof(string), stdin);
```

```
    strupr(string);
    printf("String em maiúsculas: %s", string);

    invert_words(string);
    printf("String com palavras invertidas: %s", string);

    strlwr(string);
    printf("String em minúsculas: %s", string);

    return 0;
}

/*
#include <stdio.h>
#include <string.h>
#include <ctype.h>

int main()
{
    char nome[50], outro[50];
    int n = 0;

    printf("Digite o seu nome: ");
    fgets(nome, sizeof(nome), stdin);
```

```
// Remover a nova linha que o fgets adiciona
strtok(nome, "\n");

// Inverter as letras: maiúsculas para minúsculas e vice-versa
while (nome[n] != '\0') {
    if (islower(nome[n])) {
        outro[n] = toupper(nome[n]);
    } else {
        outro[n] = tolower(nome[n]);
    }
    n++;
}
outro[n] = '\0';  // Finalizar a string invertida

// Exibir a string com letras invertidas
printf("\nSua palavra com as letras invertidas: %s", outro);

// Converter a string original para minúsculas
for (n = 0; nome[n] != '\0'; n++) {
    nome[n] = tolower(nome[n]);
}
printf("\nSua palavra em minúsculas: %s", nome);

// Converter a string original para maiúsculas
for (n = 0; nome[n] != '\0'; n++) {
```

```
        nome[n] = toupper(nome[n]);
    }
    printf("\nSua palavra em maiúsculas: %s", nome);

    return 0;
}

 */
```

AULA 4

```
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
#include <string.h>

typedef struct funcionario{
    int id;
    char nome[100];
    int idade;
    float salario;
}FUNCIONARIO;

FUNCIONARIO coletaDados();
void imprimeDados(FUNCIONARIO func[], int quantidade);
void reajusteSalario(float *salario);
```

```c
void realSalario(FUNCIONARIO func[], int quantidade);


int main(){
    int quantidade;
    setlocale(LC_ALL, "Portuguese");

    printf("Digite a quantidade de funcionários: ");
    scanf(" %d", &quantidade);
    printf("\n");

    FUNCIONARIO func[quantidade];

    for (int i = 0; i < quantidade; i++){
        func[i] = coletaDados();
    }

    imprimeDados(func, quantidade);

    for (int i = 0; i < quantidade; i++){
        reajusteSalario(&func[i].salario);
    }

    realSalario(func, quantidade);

    printf("\n\n");
```

```
    system("pause");
    return 0;
}


FUNCIONARIO coletaDados(){
    FUNCIONARIO nario;
    printf("Digite o ID do funcionário: ");
    scanf("%d", &nario.id);
    printf("Digite o nome do funcionário: ");
    getchar();
    fgets(nario.nome, sizeof(nario.nome), stdin);
    nario.nome[strlen(nario.nome)-1] = '\0';
    printf("Digite a idade do funcionário: ");
    scanf("%d", &nario.idade);
    printf("Digite o salário do funcionário: ");
    scanf("%f", &nario.salario);
    printf("\n");

    return nario;
}


void imprimeDados(FUNCIONARIO func[], int quantidade){
    for (int i = 0; i < quantidade; i++){
        printf("Funcionário %d\n", i+1);
        printf("\n");
```

```
        printf("ID: %d\n", func[i].id);
        printf("Nome: %s\n", func[i].nome);
        printf("Idade: %d\n", func[i].idade);
        printf("Salário: %.2f\n", func[i].salario);
        printf("\n");
    }
}

void reajusteSalario(float *salario){
    *salario *= 1.10;
}

void realSalario(FUNCIONARIO func[], int quantidade){
    printf("Salários reajustados:\n\n");
    for (int i = 0; i < quantidade; i++){
        printf("Nome: %s\n", func[i].nome);
        printf("Salário: %.2f\n", func[i].salario);
        printf("\n");
    }
}
```

AULA 5

```
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
```

```c
#include <string.h>


void salario(float *salario);


int main() {
    float *salario1, *salario2, *salario3;
    float valor1, valor2, valor3;
    setlocale(LC_ALL, "Portuguese");


    printf("\nDigite o valor do primeiro salário: ");
    scanf("%f", &valor1);
    salario1 = &valor1;
    printf("\nDigite o valor do segundo salário: ");
    scanf("%f", &valor2);
    salario2 = &valor2;
    printf("\nDigite o valor do terceiro salário: ");
    scanf("%f", &valor3);
    salario3 = &valor3;


    salario(salario1);
    salario(salario2);
    salario(salario3);


    printf("\nSalário 1: %.2f", *salario1);
    printf("\nSalário 2: %.2f", *salario2);
```

```
    printf("\nSalário 3: %.2f", *salario3);


    printf("\n\n");
    system("pause");
    return 0;
}


void salario(float *salario) {
    *salario += 100;
}
```

```
#include <stdio.h>
#include <locale.h>

int main() {
    setlocale(LC_ALL, "Portuguese");

    float vetorFloat[10];
    double vetorDouble[10];


    printf("Endereços do vetor de float:\n");
    for (int i = 0; i < 10; i++) {
        printf("Posição %d: %p\n", i, &vetorFloat[i]);
    }
```

```
    printf("\nEndereços do vetor de double:\n");
    for (int i = 0; i < 10; i++) {
        printf("Posição %d: %p\n", i, &vetorDouble[i]);
    }

    system("pause");
    return 0;
}
```

AULA 6

atv1

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <locale.h>
#include <string.h>

void salvarTexto(char texto[]);
void salvarTextoMaiusculo(char texto[]);
void imprimirArquivo(char nomeArquivo[]);

int main() {
    setlocale(LC_ALL, "Portuguese");

    char texto[1000];
```

```
    printf("Digite um pequeno texto: ");
    fgets(texto, sizeof(texto), stdin);

    salvarTexto(texto);

    salvarTextoMaiusculo(texto);

    printf("\nConteúdo de arq1.txt:\n");
    imprimirArquivo("arq1.txt");

    printf("\nConteúdo de arq2.txt:\n");
    imprimirArquivo("arq2.txt");

    return 0;
}

void salvarTexto(char texto[]) {
    FILE *arquivo = fopen("arq1.txt", "w");
    if (arquivo == NULL) {
        printf("Erro ao abrir o arquivo %s\n", "arq1.txt");
        exit(1);
    }
    for(int i = 0; i < strlen(texto); i++) {
        fputc(texto[i], arquivo);
```

```
    }
    fclose(arquivo);
}

void salvarTextoMaiusculo(char texto[]) {
    FILE *arquivo = fopen("arq2.txt", "w");
    if (arquivo == NULL) {
        printf("Erro ao abrir o arquivo %s\n", "arq2.txt");
        exit(1);
    }
    for(int i = 0; i < strlen(texto); i++) {
        fputc(toupper(texto[i]), arquivo);
    }
    fclose(arquivo);
}

void imprimirArquivo(char nomeArquivo[]) {
    FILE *arquivo = fopen(nomeArquivo, "r");
    if (arquivo == NULL) {
        printf("Erro ao abrir o arquivo %s\n", nomeArquivo);
        exit(1);
    }
    char c= fgetc(arquivo);
    while(c != EOF) {
        printf("%c", c);
```

```
        c = fgetc(arquivo);
    }
    fclose(arquivo);
}
```

atv2 - arquivo 1

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <locale.h>
#include <string.h>

typedef struct Criafuncionario{
    char nome[100];
    char telefone[15];
    char email[100];
} Funcionario;

void gravarVetor(Funcionario *funcionarios, int tamanho);

int main (){

    setlocale(LC_ALL, "Portuguese");

    Funcionario funcionarios[5];
```

```
    for (int i = 0; i < 5; i++) {
        printf("Digite o nome do funcionário %d: ", i + 1);
        fgets(funcionarios[i].nome, sizeof(funcionarios[i].nome), stdin);
        funcionarios[i].nome[strcspn(funcionarios[i].nome, "\n")] = '\0'; // Remove o \n

        printf("Digite o telefone do funcionário %d: ", i + 1);
        fgets(funcionarios[i].telefone, sizeof(funcionarios[i].telefone), stdin);
        funcionarios[i].telefone[strcspn(funcionarios[i].telefone, "\n")] = '\0'; // Remove o \n

        printf("Digite o email do funcionário %d: ", i + 1);
        fgets(funcionarios[i].email, sizeof(funcionarios[i].email), stdin);
        funcionarios[i].email[strcspn(funcionarios[i].email, "\n")] = '\0'; // Remove o \n
    }

    gravarVetor(funcionarios, 5);
}

void gravarVetor(Funcionario *funcionarios, int tamanho) {
    FILE *arquivo = fopen("funcionarios.bin", "wb");
    if (arquivo == NULL) {
        printf("Erro ao abrir o arquivo\n");
        exit(1);
```

```
    }

    fwrite(funcionarios, sizeof(Funcionario), tamanho, arquivo);

    fclose(arquivo);
}
```

arquivo 2

```
#include <stdio.h>
#include <stdlib.h>

typedef struct {
    char nome[100];
    char telefone[15];
    char email[100];
} Funcionario;

int main() {
    FILE *arquivo = fopen("funcionarios.bin", "rb");
    if (arquivo == NULL) {
        printf("Erro ao abrir o arquivo\n");
        return 1;
    }

    Funcionario terceiroFuncionario;
```

```
    fseek(arquivo, 2 * sizeof(Funcionario), SEEK_SET);

    fread(&terceiroFuncionario, sizeof(Funcionario), 1, arquivo);
    fclose(arquivo);

    printf("Terceiro Funcionário:\n");
    printf("Nome: %s\n", terceiroFuncionario.nome);
    printf("Telefone: %s\n", terceiroFuncionario.telefone);
    printf("Email: %s\n", terceiroFuncionario.email);

    return 0;
}
```

AULA 7

```
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>

int main() {
    setlocale(LC_ALL, "Portuguese");

    int number, *vetor, i;

    printf("Digite quantos números deseja alocar: ");
    scanf("%d", &number);
```

```c
    vetor = (int*) malloc(number * sizeof(int));
    if (vetor == NULL) {
        perror("Erro ao alocar memória");
        return 1;
    }

    for (i = 0; i < number; i++) {
        do {
            printf("Digite o %dº número: ", i + 1);
            scanf("%d", &vetor[i]);
            if (vetor[i] % 2 == 0) {
                printf("O número digitado não é ímpar. Por favor, digite um
número ímpar.\n");
            }
        } while (vetor[i] % 2 == 0 && vetor[i] != 0);
    }

    printf("Números ímpares digitados: ");
    for (i = 0; i < number; i++) {
        printf("%d ", vetor[i]);
    }
    printf("\n");

    free(vetor);
```

```
    system("pause");


    return 0;
}
```

atv 2

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int numAlunos, i;
    float *notas, soma = 0, media;


    printf("Digite o numero de alunos: ");
    scanf("%d", &numAlunos);


    notas = (float*) malloc(numAlunos * sizeof(float));

    if (notas == NULL) {
        printf("Erro ao alocar memoria.\n");
        return 1;
    }

    for (i = 0; i < numAlunos; i++) {
```

```
        printf("Digite a nota do aluno %d: ", i + 1);
        scanf("%f", &notas[i]);
        soma += notas[i];
    }

    media = soma / numAlunos;

    printf("\nNotas dos alunos:\n");
    for (i = 0; i < numAlunos; i++) {
        printf("Aluno %d: %.2f\n", i + 1, notas[i]);
    }
    printf("\nMedia da turma: %.2f\n", media);

    free(notas);

    return 0;
}
```

atv3

```
#include <stdio.h>
#include <stdlib.h>

int** alocarMatriz(int linhas, int colunas);
void lerMatriz(int** matriz, int linhas, int colunas);
int** somarMatrizes(int** matriz1, int** matriz2, int linhas, int colunas);
```

```c
void imprimirMatriz(int** matriz, int linhas, int colunas);

void liberarMatriz(int** matriz, int linhas);


int main() {

    int linhas, colunas, i;

    int **matriz1, **matriz2, **matrizSoma;


    printf("\nDigite o numero de linhas: ");

    scanf("%d", &linhas);

    printf("Digite o numero de colunas: ");

    scanf("%d", &colunas);


    matriz1 = alocarMatriz(linhas, colunas);

    printf("\nDigite os valores da primeira matriz:\n");

    lerMatriz(matriz1, linhas, colunas);


    matriz2 = alocarMatriz(linhas, colunas);

    printf("Digite os valores da segunda matriz:\n");

    lerMatriz(matriz2, linhas, colunas);


    matrizSoma = somarMatrizes(matriz1, matriz2, linhas, colunas);


    printf("\nMatriz soma:\n");

    imprimirMatriz(matrizSoma, linhas, colunas);
```

```
    liberarMatriz(matriz1, linhas);
    liberarMatriz(matriz2, linhas);
    liberarMatriz(matrizSoma, linhas);

    system("pause");

    return 0;
}

int** alocarMatriz(int linhas, int colunas) {
    int **matriz = (int**) malloc(linhas * sizeof(int*));
    for (int i = 0; i < linhas; i++) {
        matriz[i] = (int*) malloc(colunas * sizeof(int));
    }
    return matriz;
}

void lerMatriz(int** matriz, int linhas, int colunas) {
    printf("\n");
    for (int i = 0; i < linhas; i++) {
        for (int j = 0; j < colunas; j++) {
            printf("Matriz[%d][%d]: ", i, j);
            scanf("%d", &matriz[i][j]);
        }
    }
```

```
    printf("\n");
}

int** somarMatrizes(int** matriz1, int** matriz2, int linhas, int colunas) {
    int **matrizSoma = alocarMatriz(linhas, colunas);
    for (int i = 0; i < linhas; i++) {
        for (int j = 0; j < colunas; j++) {
            matrizSoma[i][j] = matriz1[i][j] + matriz2[i][j];
        }
    }
    return matrizSoma;
}

void imprimirMatriz(int** matriz, int linhas, int colunas) {
    for (int i = 0; i < linhas; i++) {
        for (int j = 0; j < colunas; j++) {
            printf("%d ", matriz[i][j]);
        }
        printf("\n");
    }
    printf("\n");
}

void liberarMatriz(int** matriz, int linhas) {
    for (int i = 0; i < linhas; i++) {
```

```
        free(matriz[i]);
    }
    free(matriz);
}
```

malloc com ponteiro triplo

```
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>

// Funções
void criaMatriz(int ***matriz, int linhas, int colunas);
void exibeMatriz(int **matriz, int linhas, int colunas);
void somaMatriz(int **matriz1, int **matriz2, int **matrizSoma, int linhas,
int colunas);
void limpaMatriz(int **matriz, int linhas);

int main() {
    setlocale(LC_ALL, "Portuguese");

    int **matriz, **matriz2, **matrizSoma, linhas, colunas;

    printf("\nSeja bem-vindo ao programa de soma de matrizes\n");

    printf("Digite o número de linhas da primeira matriz: ");
    scanf("%d", &linhas);
```

```c
printf("\nDigite o número de colunas da primeira matriz: ");
scanf("%d", &colunas);

// Criando as matrizes
criaMatriz(&matriz, linhas, colunas);
criaMatriz(&matriz2, linhas, colunas);
criaMatriz(&matrizSoma, linhas, colunas);

// Preenchendo a primeira matriz
printf("\nDigite os valores da primeira matriz:\n");
for (int i = 0; i < linhas; i++) {
    for (int j = 0; j < colunas; j++) {
        printf("Matriz[%d][%d]: ", i, j);
        scanf("%d", &matriz[i][j]);
    }
}

// Preenchendo a segunda matriz
printf("\nDigite os valores da segunda matriz:\n");
for (int i = 0; i < linhas; i++) {
    for (int j = 0; j < colunas; j++) {
        printf("Matriz[%d][%d]: ", i, j);
        scanf("%d", &matriz2[i][j]);
    }
```

```
    }

    // Exibindo a primeira matriz
    printf("\nMatriz 1:\n");
    exibeMatriz(matriz, linhas, colunas);

    // Exibindo a segunda matriz
    printf("\nMatriz 2:\n");
    exibeMatriz(matriz2, linhas, colunas);

    // Somando as matrizes
    somaMatriz(matriz, matriz2, matrizSoma, linhas, colunas);

    // Exibindo a matriz soma
    printf("\nMatriz Soma:\n");
    exibeMatriz(matrizSoma, linhas, colunas);

    // Liberando a memória
    limpaMatriz(matriz, linhas);
    limpaMatriz(matriz2, linhas);
    limpaMatriz(matrizSoma, linhas);

    return 0;
}
```

```c
// Função para criar a matriz (usando ponteiro para ponteiro)
void criaMatriz(int ***matriz, int linhas, int colunas) {
    *matriz = (int **)malloc(linhas * sizeof(int *));  // Aloca as linhas
    for (int i = 0; i < linhas; i++) {
        (*matriz)[i] = (int *)malloc(colunas * sizeof(int));  // Aloca as colunas
    }
}

// Função para exibir a matriz
void exibeMatriz(int **matriz, int linhas, int colunas) {
    for (int i = 0; i < linhas; i++) {
        for (int j = 0; j < colunas; j++) {
            printf("%d ", matriz[i][j]);
        }
        printf("\n");
    }
}

// Função para somar as matrizes
void somaMatriz(int **matriz1, int **matriz2, int **matrizSoma, int linhas, int colunas) {
    for (int i = 0; i < linhas; i++) {
        for (int j = 0; j < colunas; j++) {
            matrizSoma[i][j] = matriz1[i][j] + matriz2[i][j];
        }
```

```
    }
}


// Função para liberar a memória da matriz
void limpaMatriz(int **matriz, int linhas) {
    for (int i = 0; i < linhas; i++) {
        free(matriz[i]);  // Libera cada linha
    }
    free(matriz);  // Libera o array de ponteiros
}
```

AULA 8

```
#include <stdio.h>
#include <locale.h>

int fatorialRecursivo(int n);

int fatorialIterativo(int n);

int main() {
    int n, escolha, resultado;
    setlocale(LC_ALL, "Portuguese");

    printf("Digite um valor: ");
    scanf("%d", &n);
```

```
    printf("Escolha o método:\n1 - Fatorial Recursivo\n2 - Fatorial
Iterativo\n");
    scanf("%d", &escolha);

    switch (escolha) {
    case 1:
        resultado = fatorialRecursivo(n);
        printf("Resultado usando o fatorial recursivo: %d\n", resultado);
        break;
    case 2:
        resultado = fatorialIterativo(n);
        printf("Resultado usando o fatorial iterativo: %d\n", resultado);
        break;
    default:
        printf("Opção inválida!\n");
        break;
    }

    return 0;
}

int fatorialIterativo(int n) {
    int fatorial = 1;
    for (int i = 1; i <= n; i++) {
```

```
        fatorial *= i;
    }
    return fatorial;
}

int fatorialRecursivo(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * fatorialRecursivo(n - 1);
    }
}
```

AULA 9

atv1 - main.c

```
#include <stdio.h>
#include "my_lib.h"

int main(){
    float numeroHoras, salarioHora, valorPorFilho, taxaIR;
    int numeroFilhos;
    float salarioBruto, salarioFamilia, vantagens, INSS, IRPF, Deducoes;

    printf("Digite o numero de horas trabalhadas: ");
    scanf(" %f", &numeroHoras);
```

```
    printf("Digite o salario por hora: ");

    scanf(" %f", &salarioHora);

    printf("Digite o numero de filhos: ");

    scanf(" %d", &numeroFilhos);

    printf("Digite o valor por filho: ");

    scanf(" %f", &valorPorFilho);

    printf("Digite a taxa de IR: ");

    scanf(" %f", &taxaIR);



    calculoVantagens(numeroHoras, salarioHora, numeroFilhos, valorPorFilho,
&salarioBruto, &salarioFamilia, &vantagens);

    calculoDeducoes(salarioBruto, taxaIR, &INSS, &IRPF, &Deducoes);



    printf("\n\nSalario Bruto: %.2f\n", salarioBruto);

    printf("Salario Familia: %.2f\n", salarioFamilia);

    printf("Vantagens: %.2f\n", vantagens);

    printf("INSS: %.2f\n", INSS);

    printf("IRPF: %.2f\n", IRPF);

    printf("Deducoes: %.2f\n", Deducoes);



    return 0;

}
```

atv 1- my_lib.h

```
void calculoVantagens(float numeroHoras, float salarioHora, int numeroFilhos,
float valorPorFilho, float *salarioBruto, float *salarioFamilia, float
*vantagens);
```

```
void calculoDeducoes(float salarioBruto, float taxaIR, float *INSS, float
*IRPF, float *Deducoes);



void calculoVantagens(float numeroHoras, float salarioHora, int numeroFilhos,
float valorPorFilho, float *salarioBruto, float *salarioFamilia, float
*vantagens){

    *salarioBruto = numeroHoras * salarioHora;

    *salarioFamilia = numeroFilhos * valorPorFilho;

    *vantagens = *salarioBruto + *salarioFamilia;

}



void calculoDeducoes(float salarioBruto, float taxaIR, float *INSS, float
*IRPF, float *Deducoes){

    *INSS = salarioBruto * 0.08;

    *IRPF = salarioBruto * taxaIR;

    *Deducoes = *INSS + *IRPF;

}
```

atv 3 - main.c

```
#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include <locale.h>

#include "calculadora.h"



void exibeMenu() {

    setlocale(LC_ALL,"Portuguese");

    printf("\n=== MENU ===\n");
```

```
    printf("1. Soma\n");
    printf("2. Subtra��o\n");
    printf("3. Multiplica��o\n");
    printf("4. Divis�o\n");
    printf("5. Ver �ltimo resultado\n");
    printf("6. Sair\n");
    printf("\n Escolha uma op��o: ");
}

int main() {
    setlocale(LC_ALL,"Portuguese");
    resu *p = criaOperacao(); // Cria o TAD
    int opcao;
    float a, b, resultado;

    do {
        exibeMenu();
        scanf("%d", &opcao);

        switch(opcao) {
            case 1: // Soma
                printf("Digite dois n�meros: \n");
                scanf("%f %f", &a, &b);
                resultado = soma(p, a, b);
                printf("\n Resultado: %.2f\n", resultado);
```

```
            break;
        case 2: // Subtra��o
            printf("Digite dois n�meros:\n ");
            scanf("%f %f", &a, &b);
            resultado = subtracao(p, a, b);
            printf("\n Resultado: %.2f\n", resultado);
            break;
        case 3: // Multiplica��o
            printf("Digite dois n�meros: \n");
            scanf("%f %f", &a, &b);
            resultado = multiplicacao(p, a, b);
            printf("\n Resultado: %.2f\n", resultado);
            break;
        case 4: // Divis�o
            printf("Digite dois n�meros: \n");
            scanf("%f %f", &a, &b);
            resultado = divisao(p, a, b);
            printf("\n Resultado: %.2f\n", resultado);
            break;
        case 5: // Ver �ltimo resultado
            printf("\n �ltimo resultado: %.2f\n", ultimoResultado(p));
            break;
        case 6: // Sair
            printf("\nSaindo...\n");
            break;
```

```
            default:
                printf("\nOp��o inv�lida!\n");
        }
    } while(opcao != 6);

    liberaOperacao(p); // Libera a mem�ria
    return 0;
}
```

atv 3 - calculadora.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "calculadora.h"

struct armazena{
float resultadoFinal;
};

resu* criaOperacao(){
 resu *p = (resu*) malloc(sizeof(resu));
 if(p!= NULL){
    p->resultadoFinal = 0;
 }
 return p;
```

```
}


float soma (resu *p, float a, float b){
p->resultadoFinal = a + b;
return p->resultadoFinal;
}


float subtracao (resu *p, float a, float b){
p->resultadoFinal = a - b;
return p->resultadoFinal;
}


float multiplicacao (resu *p, float a, float b){
p->resultadoFinal = a * b;
return p->resultadoFinal;
}


float divisao (resu *p, float a, float b){
p->resultadoFinal = a / b;
return p->resultadoFinal;
}


float ultimoResultado(resu *p){
return p->resultadoFinal;
}
```

```
void liberaOperacao(resu *p){

free(p);

}
```

atv3 - calculadora.h

```
typedef struct armazena resu;


resu* criaOperacao();


float soma(resu *p, float x, float y);


float subtracao(resu *p, float x, float y);


float multiplicacao(resu *p, float x, float y);


float divisao(resu *p, float x, float y);


float ultimoResultado(resu *p);


void liberaOperacao(resu *p);
```