# AULA 09

{introcomp}

## PONTEIROS

# Objetivo

- Compreender a definição e dominar a implementação de ponteiros em C.
- Dominar a manipulação de arquivos

# Motivação para a utilização de ponteiros

- O que precisamos para fazer para que uma função altere mais de um valor?
- Como fazer uma função que calcule a soma e o produto de dois números inteiros?

# Por que o seguinte trecho de código não funciona?

```
1#include <stdio.h>
2
3    void somaprod (int a, int b, int c, int d){
4        c = a + b;
5        d = a * b;
6    }
7
8    int main(){
9        int a = 2;
10       int b = 4;
11       int s, p
12       somaprod(a, b, s, p);
13
14       printf("soma: %d\n", c);
15       printf("produto: %d\n", d);
16
17   return 0;
18
19   }
```

# Endereços de memória

- Ao declarar uma variável, além de reservar certo espaço para a variável, também reservamos espaço para o endereço de memória da variável

# Como declarar uma variável do tipo ponteiro

- Para declarar uma variável do tipo int, escrevemos:
  - int a;
- Para declarar uma variável do tipo ponteiro para inteiro, escrevemos:
  - int *p;
- Para armazenar o endereço da variável **a** no ponteiro **p**, utilizamos o operador ‘&’:
  - p = &a;

# Como declarar uma variável do tipo ponteiro

- De forma análoga, podemos declarar ponteiros de outros tipos:
  - **float * m;**
  - **char * nome;**
  - **double * xD;**
  - **Ponto * p1;** //este último é uma variável de um TAD

# Trabalhando com ponteiros

- Podemos acessar o valor de uma variável através do operador ‘*’ antes do ponteiro:

```
1     int a;
2     int * p;
3
4     a = 5; //a recebe 5
5
6     p = &a; //p recebe o endereço de a
7
8     *p = 6; //o conteúdo da variável que p aponta recebe 6
```

acessar ***p** é equivalente a acessar **a**

# Exercício 1

Faça um código que leia dois números inteiros e imprima o produto deles, mas realizando a multiplicação através de variáveis do tipo ponteiro.

# Ponteiros e estruturas

A forma de referenciar o conteúdo da struct apontada pelo ponteiro é sempre da forma:

**(*ponteiro).Atributo**

Sendo os parênteses obrigatório.

# Ponteiros e estruturas

- Acessar uma struct apontada por um ponteiro para muitos pode parecer confusa.
- É muito comum errar os parênteses necessários para garantir a precedência dos operadores “.” e “*”
- Porém existe o operador seta “->” que facilita o uso de ponteiros para estruturas.

**(*p).atr;**  **p->atr;**

# Ponteiros e funções

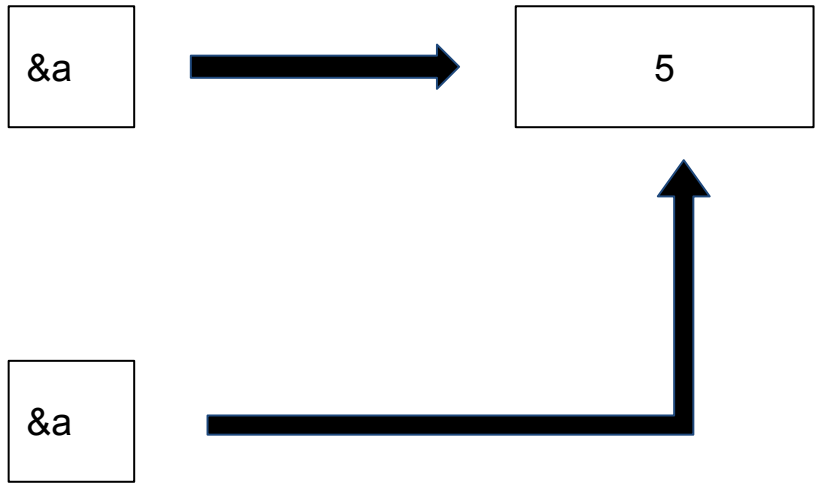
- Podemos passar ponteiros de variáveis como parâmetros de uma função.
  - Isso é útil para resolver, por exemplo, o problema apresentado no começo da aula
- Quando alteramos o conteúdo de uma variável através de um ponteiro dentro de uma função, alteramos também o valor da variável onde a função foi chamada. Por que?
- A passagem de ponteiros para função é **por referência**.

# Ponteiros como parâmetro de funções

```c
#include <stdio.h>

void s(int a, int b, int * soma){
    *soma = a + b;
}

int main(){
    int a, b, soma;

    scanf("%d %d", &a, &b);

    s(a, b, &soma);

    printf("%d\n", soma);

    return 0;
}
```

# Ponteiros como parâmetro de funções

```
7 int a = 5;
8
9 int * p = &a;
```




# Ponteiros como parâmetro de funções

- Portanto, agora podemos alterar diversos valores com apenas uma chamada de função.
- Podemos resolver o problema apresentado no início da aula utilizando ponteiros, da seguinte forma:

```
1 void somaprod (int a, int b, int * soma, int * prod){
2     *soma = a + b;
3     *prod = a * b;
4 }
...

6     int soma, prod;
7
8     somaprod(3, 5, &soma, &prod);
```

# Exercício 2

Faça uma função que receba dois ponteiros para números inteiros e inverta os dois. Seu programa passará somente dois ponteiros para inteiros como os parâmetros da função e **não poderá ter retorno** (deverá ser do tipo void):

```
void inverte(int * a, int * b);
```

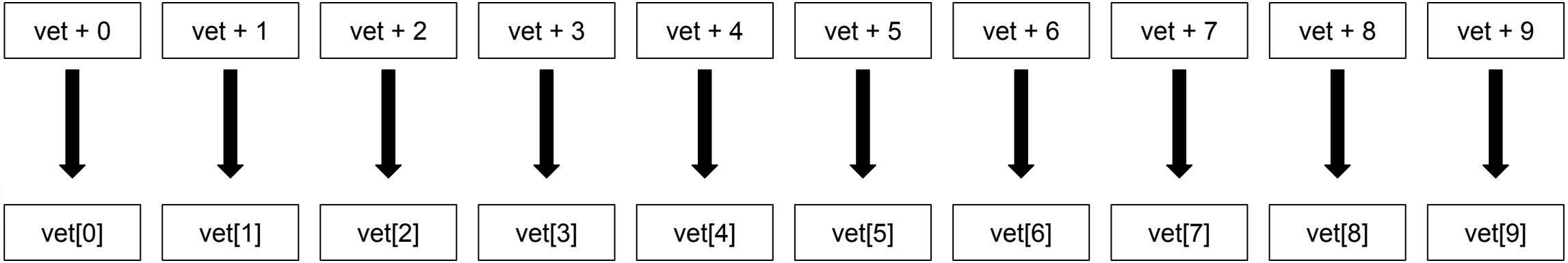
# Ponteiros e vetores

- Ao declarar um vetor, por exemplo:
  - **int vet[10];**

  estamos reservando 10 espaços de memória do tamanho de int
- vet (sem a indexação) representa um ponteiro para a primeira célula do vetor
  - (vet + 0) aponta para o primeiro elemento do vetor
  - (vet + 1) aponta para o segundo elemento do vetor
  - (vet + 2) aponta para o terceiro elemento do vetor
  - (vet + 3) aponta para o quarto elemento do vetor

# Ponteiros e vetores

- vet[4] é o valor do quinto elemento do vetor, e é uma forma simplificada de escrever *(vet + 4)




# Vetores como parâmetros de funções

- Assim vimos que vetores são tratados como ponteiros, logo a passagem de vetores como parâmetros para funções pode ser feita da seguinte forma:

  **void funcao (int *vet);**

- ao inves de:

  **void funcao (int vet[]);**

# Leitura de arquivos

- A linguagem C permite acesso tanto para leitura quanto para escrita de arquivos.

- O primeiro passo é criar um ponteiro para o arquivo, escrevendo:
  - **FILE * fp;**

# Leitura de arquivos

- Após criar o ponteiro do arquivo, deve-se chamar a função **fopen** e armazenar o retorno da função no ponteiro criado
- A função fopen possui 2 parâmetros: O caminho do arquivo e o modo de abertura.
- Exemplo de chamada da função:
  - **fp = fopen("introcomp.txt", "a+");**
- Como padrão, o caminho começa onde o arquivo executável foi executado, mas é possível escrever o caminho desde a pasta raiz .

# Modos de acesso ao arquivo

- **r** -> leitura do arquivo
  - o arquivo acessado precisa existir
- **w** -> escrita no arquivo
  - caso o arquivo já exista, o conteúdo anterior é perdido
  - caso não exista, um novo arquivo é criado
- **a** -> escrita no final do arquivo
  - caso o arquivo já exista, o conteúdo anterior permanece e o programa escreve no final do arquivo
  - caso não exista, um novo arquivo é criado

# Modos de acesso ao arquivo

- É possível ler e escrever ao mesmo tempo, basta utilizar o operador **‘+’**, como **r+**, **a+**, **w+** .

- Vale lembrar que as regras citadas anteriormente continuam valendo.

# Checando se um arquivo foi aberto corretamente

- Quando a função fopen funciona corretamente, ela retorna um ponteiro do tipo **FILE***
- Entretanto, ao se deparar com algum erro, a função fopen retorna uma constante nula (NULL).

```
4 FILE* arquivo = fopen("arquivo.txt", "r");
5 if (arquivo == NULL){
6         printf("Erro ao abrir o arquivo");
7         exit(1);//encerra o programa, mas nao eh obrigatorio
8 }
```

# Funções de leitura de arquivo

- Podemos ler o conteúdo de um arquivo de forma similar a leitura do terminal
- a função fscanf possui 3 parâmetros, dois deles iguais a scanf
  - **fscanf(fp, “%d %d”, &a, &b)**;
- a função fgetc possui um parâmetro, que é do tipo FILE*
  - fgetc(fp);
  - para armazenar um caractere lido com essa função, deve-se utilizar seu retorno:
    - c = fgetc(fp);

# Funções de escrita no arquivo

- Podemos também escrever algo num arquivo de forma similar a qual escrevemos no terminal

- A função mais utilizada é a fprintf, que também possui 3 parâmetros, e dois deles são iguais aos do printf
  - **fprintf(fp, “%d, %d, %d”, a, b, c);**

# Fechamento do arquivo

- Ao terminar de trabalhar com o arquivo, deve-se sempre lembrar de o fechar

- a função fclose é bem simples, possuindo apenas um parâmetro:
  - **fclose(fp);**

# Exemplo de código

O código abaixo faz a leitura de uma string do arquivo **“arq_entrada.txt”**, cria um arquivo de saída **“arq_saida.txt”** e escreve nele a string lida do primeiro arquivo.

```
3  int main(){
4      FILE* entrada = fopen("arq_entrada.txt", "r");
5      char str[10];
6      fscanf(entrada, "%s", str);
7
8      FILE* saida = fopen("arq_saida.txt", "w");
9      fprintf(saida, "%s", str);
10     return 0;
11 }
```

# Exercício 3

Faça um programa que leia **do terminal** o nome de uma pessoa e sua data de aniversário e salve essas informações num arquivo de texto (pessoa.txt) com o seguinte formato:

Nome da pessoa: <nome>
Data de aniversário: <data>

# Exercício 4

Faça um programa que leia uma palavra do terminal e uma palavra num arquivo de texto e diga se as duas palavras são iguais.