

INSTITUTO FEDERAL

Norte de Minas Gerais

Campus Januária

Estruturas de Dados I

- Modularização -



■ Funções e Procedimentos

- Embora sem saber como escrever uma função, já temos utilizado várias ao longo das aulas:

- **printf()**
- **scanf()**
- **strlen()**
- **strcmp()**
- **main()**



Objetivos e Vantagens

- Reduzir a complexidade de um problema através da **divisão** em **problemas menores**.
- Facilitar a **compreensão** e **manutenção** do código.
- Facilitar a **reutilização** de código.





Exemplo Prático

- Faça um programa que, utilizando laços de repetição, produza a seguinte saída:

```
*****  
Impressão de 1 a 5  
*****  
1  
2  
3  
4  
5  
*****
```

Utilize laço FOR para
imprimir as linhas com
20 símbolos *



Problema

- Observe que o trecho de código abaixo teve que ser **reescrito 3 vezes...**

```
for (int i=0; i<=20; i++){  
    printf("*");  
}  
printf("\n");
```




Solução

- E SE...
- Criação de **função específica** para essa tarefa:

```
void print_linha(){  
    for (int i=0; i<=20; i++)  
        printf("*");  
    printf("\n");  
}
```



Solução

■ Solução modular do problema:

```
int main(){  
    print_linha();  
    printf("Impressão de 1 a 5\n");  
    print_linha();  
    for (int i=0; i<=5; i++)  
        printf("%d\n",i);  
    print_linha();  
}
```



Percebam...

```
void print_linha(){  
    for (int i=0; i<=30; i++)  
        printf("*");  
    printf("\n");  
}  
  
int main(){  
    print_linha();  
    printf("Impressão de 1 a 5");  
    print_linha();  
    for (int i=0; i<=5; i++)  
        printf("%d\n",i);  
    print_linha();  
}
```

Caso seja necessário aumentar o tamanho da linha, de 20 para 30 asteriscos, basta alterar a quantidade na função.



Percebam...

Caso deseje
trocar o padrão
de desenho da
linha, basta
trocar o
caractere na
função.

```
void print_linha(){
    for (int i=0; i<=20; i++)
        printf(".");
    printf("\n");
}

int main(){
    print_linha();
    printf("Impressão de 1 a 5\n");
    print_linha();
    for (int i=0; i<=5; i++)
        printf("%d\n",i);
    print_linha();
}
```

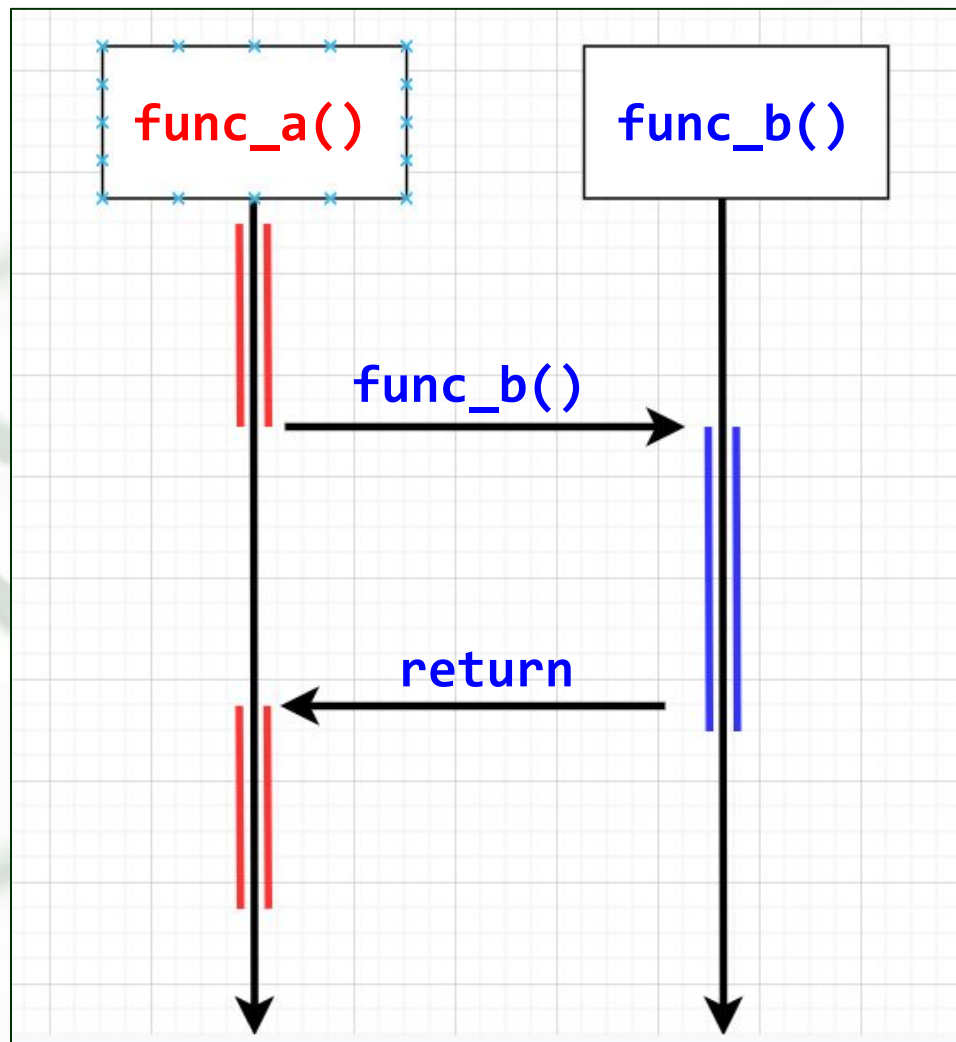


Características de uma Função

- Cada função tem um **nome único** (assinatura), o qual serve para sua invocação/chamada.
- Uma função somente é executada a partir da sua **explícita invocação** por outra função já em execução.
- Quando **func_a()** invoca **func_b()**, a linha de execução em **func_a()** é suspensa e transferida para execução de **func_b()**. Somente após o término de **func_b()**, a linha de execução retorna à **func_a()** a partir do ponto que havia sido suspensa.



Características de uma Função





Características de uma Função

- Uma função pode ser invocada a **partir de qualquer outra função** (e até por ela mesma: ***Técnica de Recursividade***).
- As variáveis declaradas dentro de uma função **são locais** à esta (variáveis de escopo local), e não são acessíveis e nem interferem no restante da aplicação.



Responsabilidades

**Toda função deve ter uma responsabilidade
muito bem definida!**

(Princípio da Responsabilidade Única - SRP)

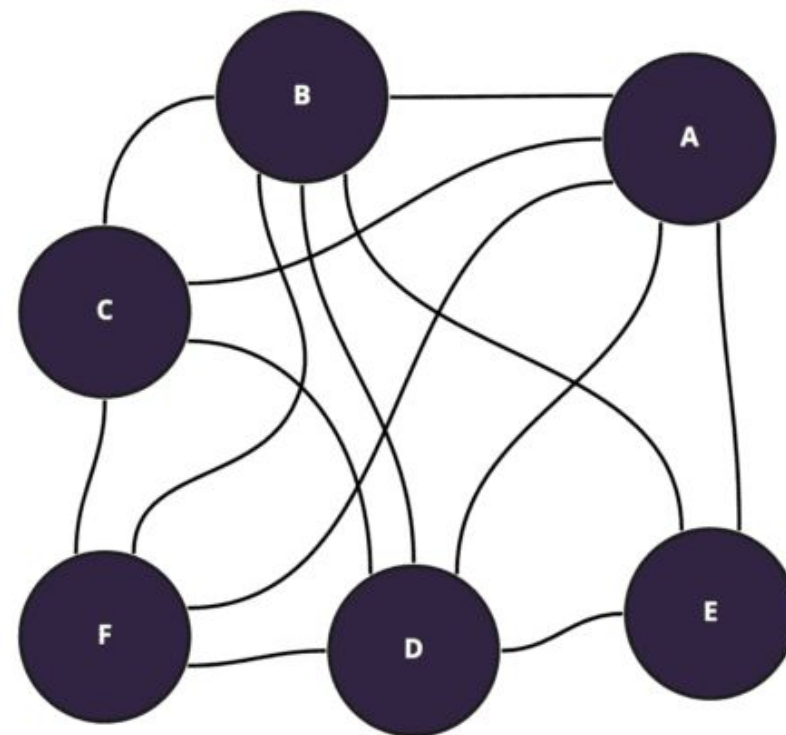
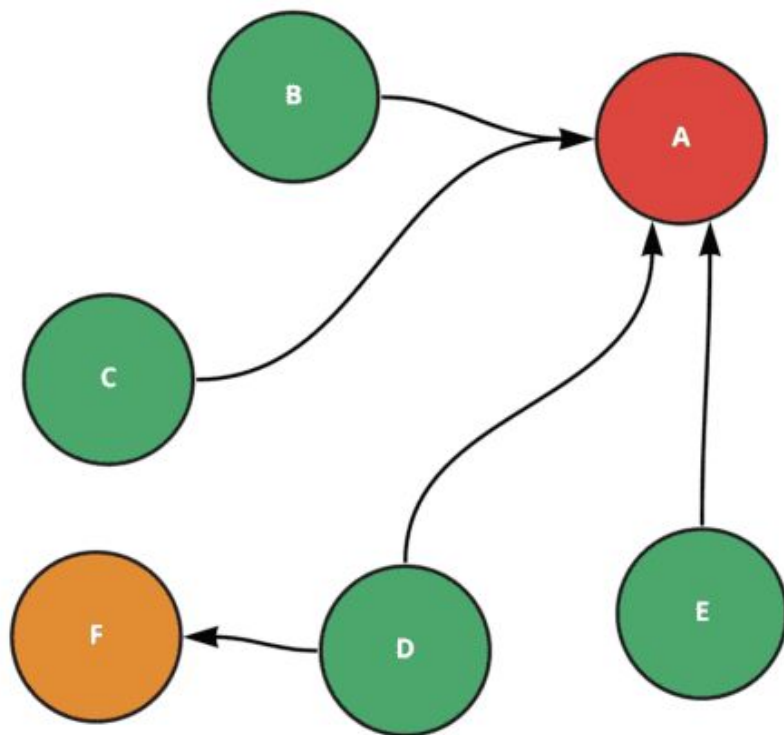
**As funções não devem fazer, nem mais e nem
menos, do que a responsabilidade prevista!**

**Toda função deve ser o mais independente
possível das demais funções!**

(Conceito de Baixo Acoplamento)



Responsabilidades





Parametrização

- Uma função pode estabelecer **parâmetros** que alteram o seu comportamento de forma a **adaptar-se** a situações distintas.





Parametrização

■ Como resolver a impressão abaixo?

.....

Impressão de 1 a 5

1

2

3

4

5



Parametrização

■ Como resolver a impressão abaixo?

.....

Impressão de 1 a 5

1

2

3

4

5

Declara-se 03 funções...

`linhaPonto();`

`linhaTraco();`

`linhaAsterisco();`

?



Parametrização

***ou fazemos
um código
mais
inteligente...***

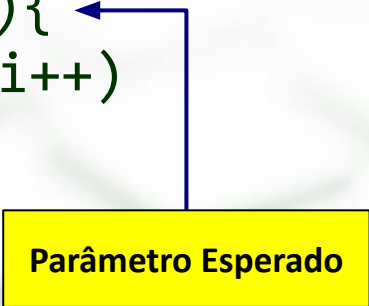
```
void print_linha(char c){  
    for (int i=0; i<=60; i++)  
        printf("%c", c);  
    printf("\n");  
}  
  
int main(){  
    print_linha('.');  
    printf("Impressão de 1 a 5\n");  
    print_linha('-');  
    for (int i=0; i<=5; i++)  
        printf("%d\n",i);  
    print_linha('*');  
}
```




Parametrização

*ou fazemos
um código
mais
inteligente...*

```
void print_linha(char c){  
    for (int i=0; i<=60; i++)  
        printf("%c", c);  
    printf("\n");  
}  
  
int main(){  
    print_linha('.');  
    printf("Impressão de 1 a 5\n");  
    print_linha('-');  
    for (int i=0; i<=5; i++)  
        printf("%d\n",i);  
    print_linha('*');  
}
```



Parâmetro Esperado



Parametrização

*ou fazemos
um código
mais
inteligente...*

```
void print_linha(char c){  
    for (int i=0; i<=60; i++)  
        printf("%c", c);  
    printf("\n");  
}
```

Argumento Enviado

```
int main(){  
    print_linha('.');  
    printf("Impressão de 1 a 5\n");  
    print_linha('-');  
    for (int i=0; i<=5; i++)  
        printf("%d\n",i);  
    print_linha('*');  
}
```



Parametrização

- A **comunicação** entre as funções deve ser feita por meio dos **parâmetros e dos respectivos argumentos** que enviamos.
- Cada função pode definir **N parâmetros** (separados por vírgula), conforme a sua necessidade.
- Em C, o **tipo de cada parâmetro** deve ser explicitamente definido.



Declaração de Função

Parâmetros da Função

```
void nome_funcao(int k, char c, float m){  
    comando1;  
    comando2;  
    ...  
    comandoN;  
}
```

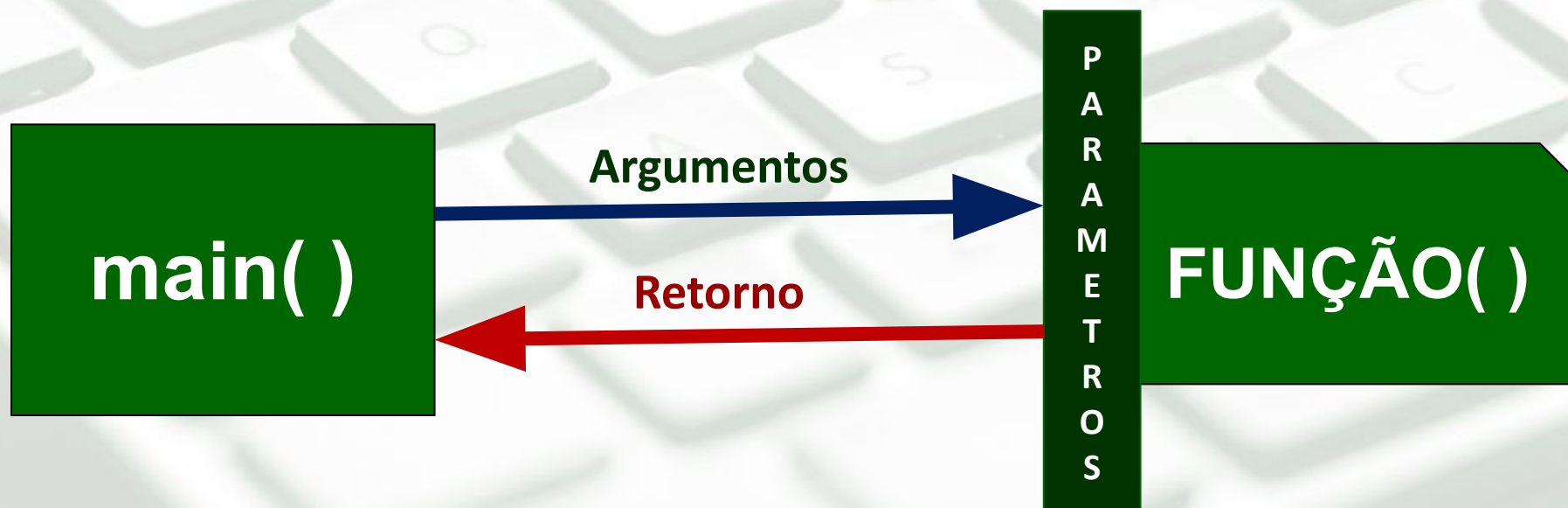
ATENÇÃO

A **quantidade** e a **ordem** dos parâmetros devem ser **sempre respeitadas** na invocação da função.



Retorno de Valores

- Tecnicamente, até aqui trabalhamos somente com **PROCEDIMENTOS** e não propriamente **FUNÇÕES**.
- A diferença é que **uma função retorna um resultado** para quem a invocou.





Retorno de Valores

- A função também deve explicitar o **TIPO** do valor que retorna.

Tipo do Retorno
da Função

Parâmetros da Função

```
int nome_funcao(int k, char c, float m){  
    comando1;  
    comando2;  
    ...  
    comandoN;  
    return valor;  
}
```

Valor que será retornado



Exemplo de uma Função

- *O que será impresso na tela?*

```
int soma(int a, int b){  
    int c;  
    c = a + b;  
    return c;  
}  
  
int main(){  
    int n1 = 8;  
    int n2 = 6;  
    printf("%d",soma(n1,n2));  
}
```



Exemplo de uma Função

■ *Seja...*

```
int soma(int a, int b){  
    int c;  
    c = a + b;  
    return c;  
}  
  
int dobro(int a){  
    return 2*a;  
}
```

■ *Qual o retorno da chamada:*

dobro(soma(dobro(4),3))



Bora CODAR!!!



- Implemente a função **potencia()** que retorna o valor de X elevado a N. Os valores de X e N devem ser enviados através de parâmetros.
- Faça uma função **interface()** que imprime um menu de operações matemáticas na tela (1-soma, 2-subtração, 3-multiplicação, 4-divisão e 5-potência). Obtenha a opção escolhida pelo usuário e retorne-a como resultado da interface. Este valor deverá ser tratado pela função **main()**, que irá processar a escolha, encaminhando o usuário para função específica.
- Desenvolva uma função que recebe por parâmetro um valor inteiro X. Essa função deve calcular o primeiro número primo maior ou igual a X. Na função **main()** preencha um vetor de N números aleatórios, e para cada número gerado, invoque a função criada.
- Faça um programa que sorteie N n^{os} entre 0 e X. Faça a ordenação do Array na função **main()**, mas levando em consideração a qtde. de bits **1** que os números formam em binário (crie função específica para calcular isso). Imprima o resultado em decimal e binário (crie outra função para isso).



Observe o Seguinte Código...

```
void incremento(int a){  
    a += 1;  
}  
  
int main(){  
    int a = 3;  
    incremento(a);  
    printf("%d",a);  
}
```

O que será impresso na tela?



Problema

- O problema anterior se deve a **passagem de parâmetros** ter sido realizada por **"valor"**

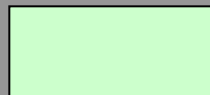
Área de Memória Reservada
para main()

a

3

Área de Memória Reservada
para função incremento()

a





Problema

■ Passagem de Parâmetro por **valor**

Área de Memória Reservada
para main()

a

3

Área de Memória Reservada
para função incremento()

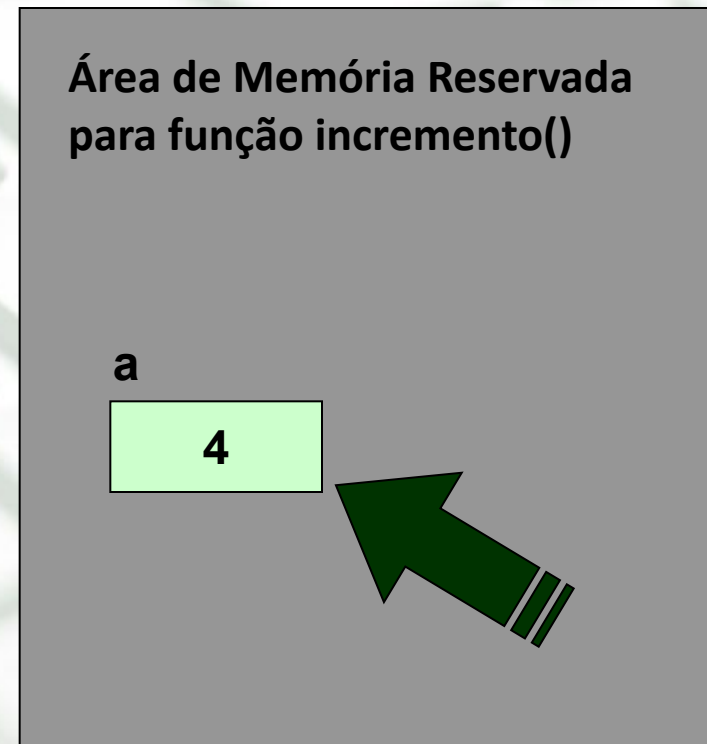
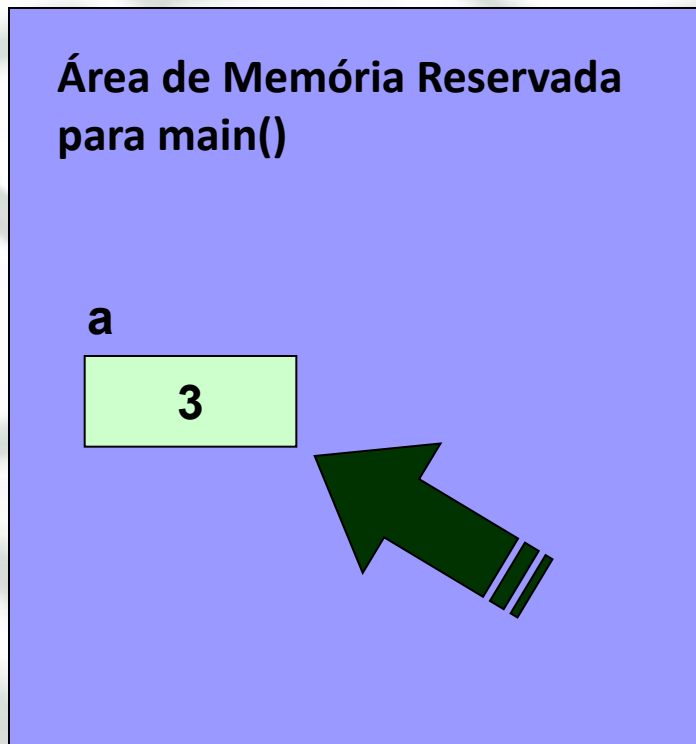
a

3



Problema

■ Processamento da função **incremento()**





Problema

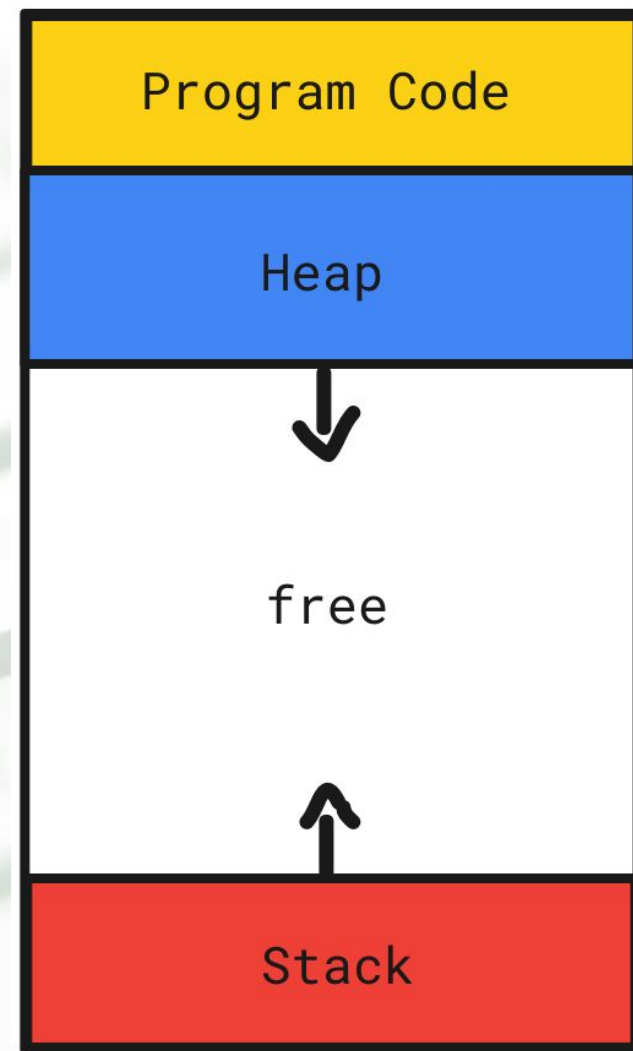
- O problema ocorreu porque o parâmetro de uma função **se comporta como uma variável local**.
- O que foi alterado (incrementado) foi a "variável **local da função**", e não a variável existente na função `main()`.

Na passagem de parâmetro por valor, não é a variável que é enviada para a função, mas sim, uma cópia do seu valor.



Saiba mais...

*Saiba mais, pesquisando e estudando sobre os espaços de endereçamentos **STACK** e **HEAP**.*





Passagem por Referência

- Outra forma de passagem de parâmetros é denominada **passagem por referência**.
- Neste caso, o que é enviado para a função não é uma cópia do valor, mas sim, a própria variável através de uma referência.
- Essa referência é o **ponteiro** da variável, ou seja, o **endereço físico da porção da memória** onde a variável está alocada.



Passagem por Referência

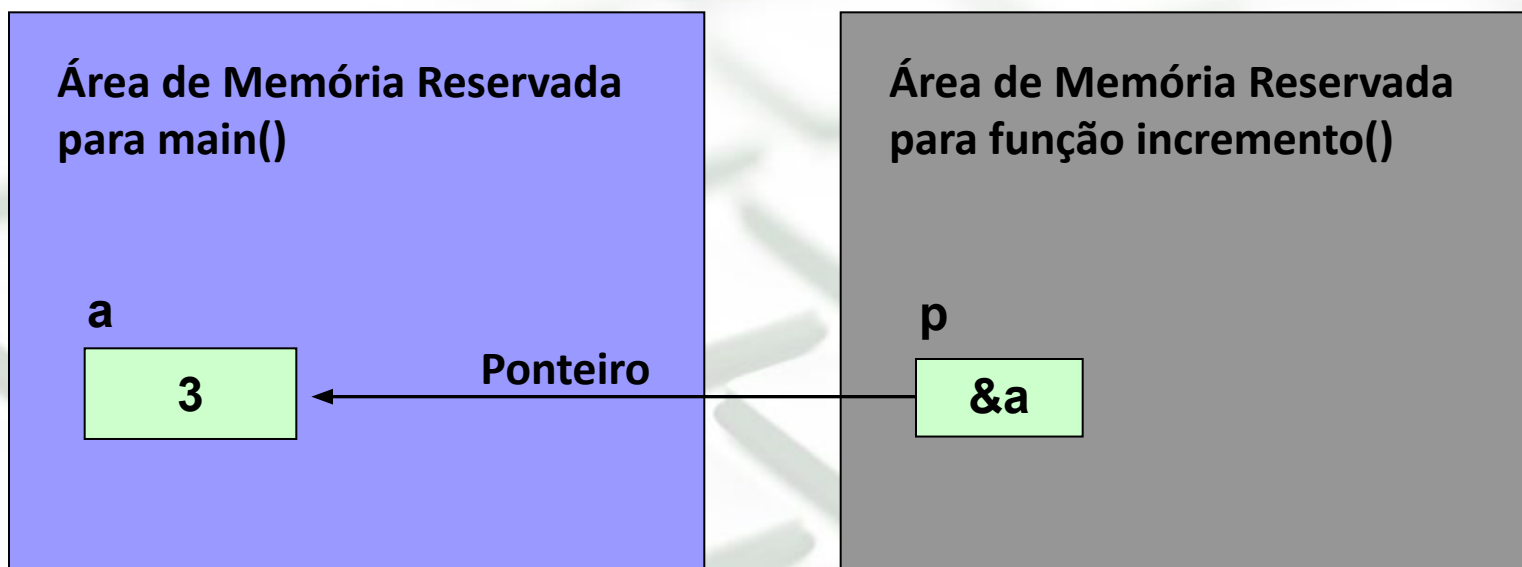
```
void incremento(int *a){  
    *a += 1;  
}  
  
int main(){  
    int a = 3;  
    incremento(&a);  
    printf("%d", a);  
}
```

O que será impresso na tela?



Passagem por Parâmetros

■ Ilustrando...



"p" é uma variável ponteiro;

o conteúdo de "p" é o endereço de "a" (logo, aponta para "a")

`(*p) == 3` (o conteúdo que "p" está apontando é igual a 3)



Parâmetros por Referência

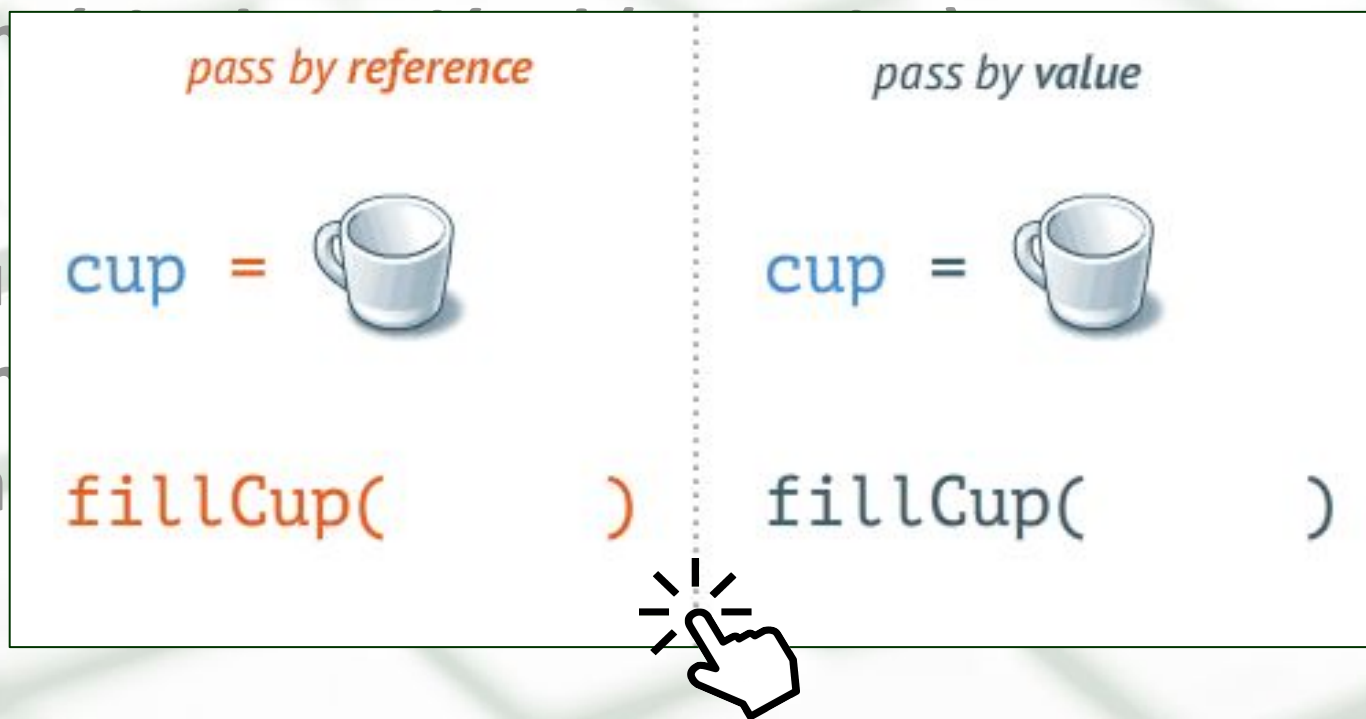
- A passagem de **parâmetros por referência** acontece por meio da **cópia do endereço de memória da variável (ponteiro)**.
- A função invocada consegue desta forma, **manipular diretamente** os dados da variável localizada no escopo da outra função.



Parâmetros por Referência

- A passagem de parâmetros por referência
Em outras palavras... é uma cópia do endereço de memória

- A função manipula a memória local





Introdução a Ponteiros

- A sintaxe de declaração de um ponteiro é a seguinte.

```
tipo    *ptr; //ou  
tipo*   ptr;
```

Onde...

tipo	É o tipo da variável para qual o ponteiro irá referenciar
*	Indica que a variável é um ponteiro (pois receberá um endereço)
ptr	É o nome dado à referência (ou seja, o nome do ponteiro)



Introdução a Ponteiros

```
int main(){  
    int *ptr;  
    int n = 9;  
    ptr = &n;  
    *ptr += 1;  
    printf("A variável N está alocada  
           no endereço %p e, naquele  
           espaço de memória o valor  
           atual é %d", ptr, *ptr);  
}
```

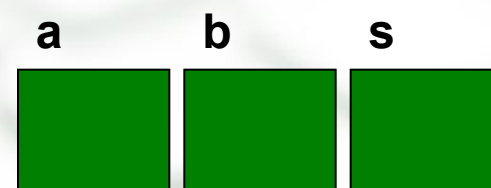
&n pode ser lido como:
"o endereço da variável n"

***ptr** pode ser lido como:
"o valor apontado por ptr"



Acompanhe...

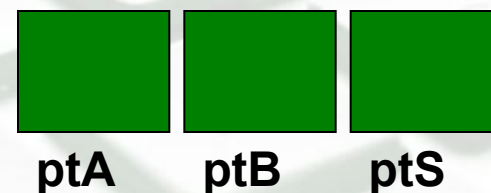
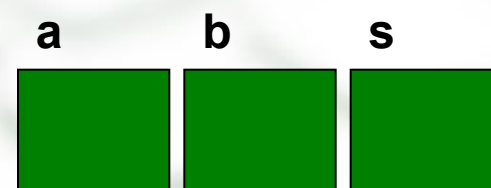
```
int main(){  
→ int a,b,s;  
  int *ptA,*ptB,*ptS;  
  a = 2;  
  b = 3;  
  ptA = &a;  
  ptB = &b;  
  ptS = &s;  
  *ptS = *ptA + *ptB;  
}
```





Acompanhe...

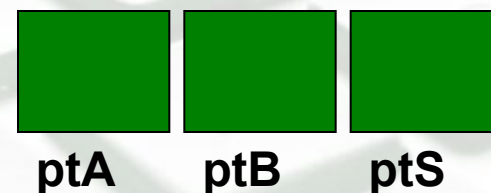
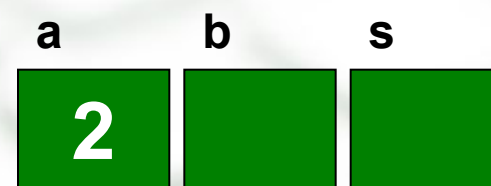
```
int main(){  
    int a,b,s;  
    → int *ptA,*ptB,*ptS;  
    a = 2;  
    b = 3;  
    ptA = &a;  
    ptB = &b;  
    ptS = &s;  
    *ptS = *ptA + *ptB;  
}
```





Acompanhe...

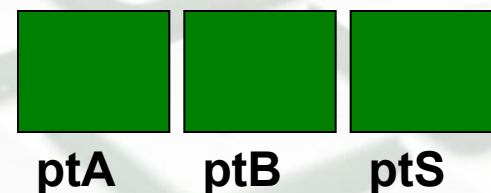
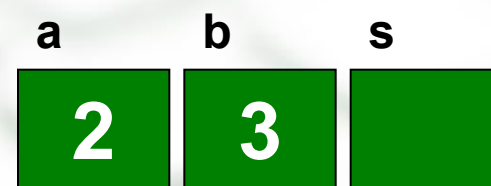
```
int main(){  
    int a,b,s;  
    int *ptA,*ptB,*ptS;  
    → a = 2;  
    b = 3;  
    ptA = &a;  
    ptB = &b;  
    ptS = &s;  
    *ptS = *ptA + *ptB;  
}
```





Acompanhe...

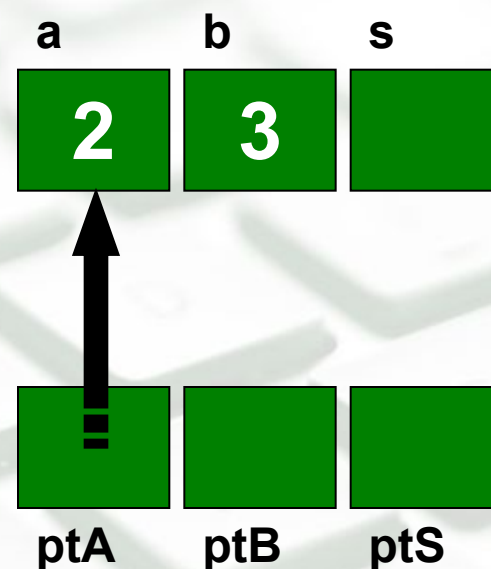
```
int main(){  
    int a,b,s;  
    int *ptA,*ptB,*ptS;  
    a = 2;  
    → b = 3;  
    ptA = &a;  
    ptB = &b;  
    ptS = &s;  
    *ptS = *ptA + *ptB;  
}
```





Acompanhe...

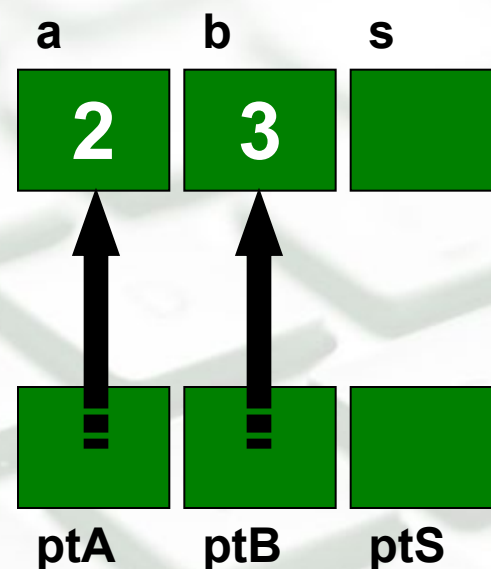
```
int main(){  
    int a,b,s;  
    int *ptA,*ptB,*ptS;  
    a = 2;  
    b = 3;  
    → ptA = &a;  
    ptB = &b;  
    ptS = &s;  
    *ptS = *ptA + *ptB;  
}
```





Acompanhe...

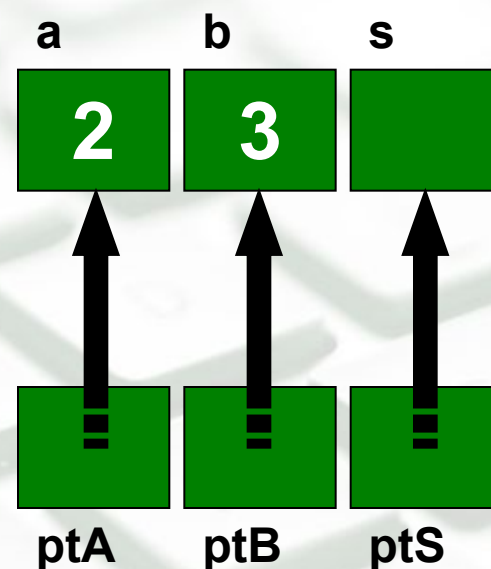
```
int main(){  
    int a,b,s;  
    int *ptA,*ptB,*ptS;  
    a = 2;  
    b = 3;  
    ptA = &a;  
    → ptB = &b;  
    ptS = &s;  
    *ptS = *ptA + *ptB;  
}
```





Acompanhe...

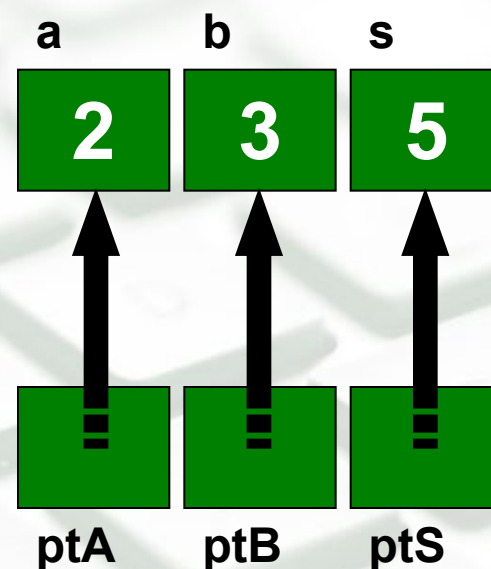
```
int main(){  
    int a,b,s;  
    int *ptA,*ptB,*ptS;  
    a = 2;  
    b = 3;  
    ptA = &a;  
    ptB = &b;  
    → ptS = &s;  
    *ptS = *ptA + *ptB;  
}
```





Acompanhe...

```
int main(){  
    int a,b,s;  
    int *ptA,*ptB,*ptS;  
    a = 2;  
    b = 3;  
    ptA = &a;  
    ptB = &b;  
    ptS = &s;  
    → *ptS = *ptA + *ptB;  
}
```





Questão de Concurso

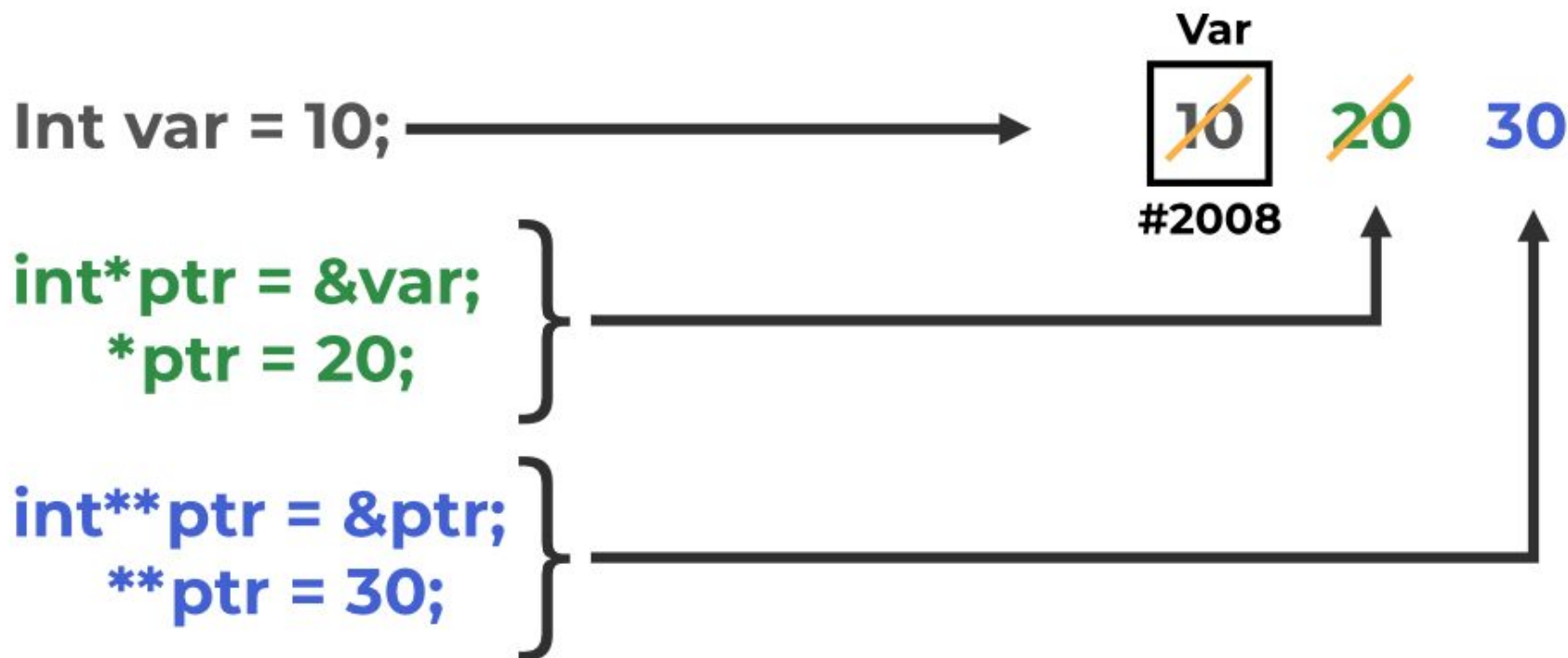
```
int main(){  
    int k = 5;  
    int y = 10;  
    int *s = &y;  
    int j = *s *10;  
    int* q = &k;  
    int *w = &j;  
    *s = *q + *w;  
    printf("%d",y);  
    return 0;  
}
```

**O que será
impresso na tela?**



Calma... Pode piorar...

How Pointer Works in C





Parâmetros por Referência

```
void triplica(int *a){  
    *a *= 3;  
}  
  
int main(){  
    int a = 3;  
    triplica(&a);  
    printf("%d",a);  
}
```

O que será impresso na tela?



Procedimento => Função

Mas não conseguiríamos obter o mesmo resultado sem usar ponteiros e transformando o procedimento em uma função???



```
int triplica(int a){  
    return a*3;  
}  
  
int main(){  
    int a = 3;  
    a = triplica(a);  
    printf("%d",a);  
}
```



Veja esse outro caso...

- Faça um procedimento ou função, que receba duas variáveis inteiras e as troque de lugar (assim como é necessário nos algoritmos de ordenação).

Também seria possível resolver nos dois modos?



Bora Codar!!!



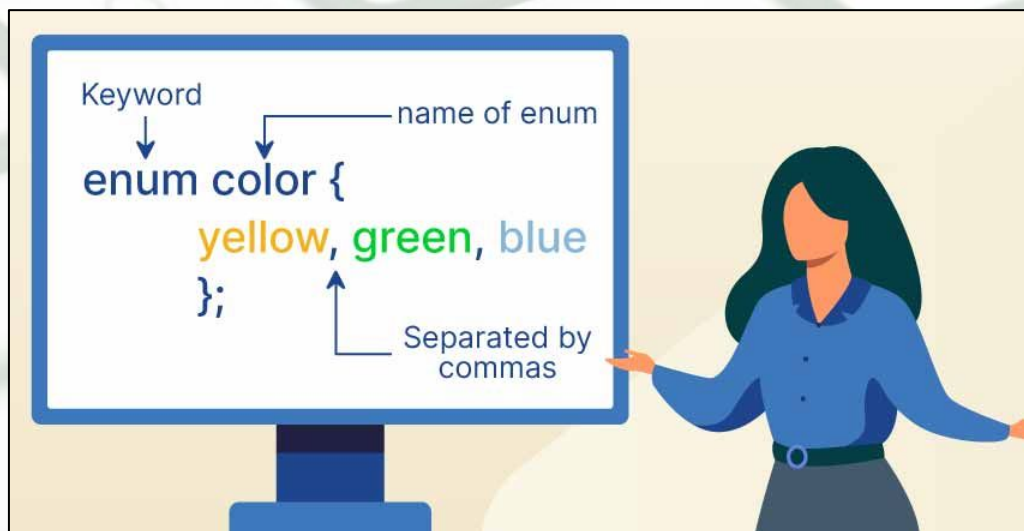
- Faça um programa que declare uma struct chamada **Cotação**. As informações desta struct são:

```
float dolar;  
float euro;  
float libra;  
float peso;
```
- Crie **funções específicas** para que o usuário informe o valor da cotação atual em cada moeda.
- Crie **funções específicas** para que, dado um determinado valor em Reais, a função converta o valor para a respectiva cotação desejada.
- **PENSE E ANALISE...** Seria possível criar uma **ÚNICA** função para atualizar a cotação de cada moeda? Como seria?
- Seria possível criar uma **ÚNICA** função para calcular a conversão de cada moeda?



ENUM

- Pesquise sobre a estrutura **ENUM** e suas aplicações de uso.
- Retorne ao problema anterior e observe se essa estrutura caberia como uma solução tecnicamente viável.
- Elabore e implemente um problema-solução em que a adoção da estrutura **ENUM** e suas vantagens sejam compreendidas.





Situação Problema

- Observe a seguinte função `main()`

```
int main(){
    Pessoa p;
    setPessoa(?); //função para cadastrar os dados de uma pessoa
    getPessoa(?); //função para imprimir os dados de uma pessoa
    return 0;
}
```

- Usando **exatamente** essa função `main()`, desenvolva as outras funções para gerar o resultado esperado.



Cadastro (SET)

Versão Procedimental

```
typedef struct{  
    char nome[100];  
    int idade;  
}Pessoa;
```

```
void setPessoa(Pessoa *nova){  
    printf("Digite o Nome: ");  
    scanf(" %[^\\n]s",nova->nome);  
    printf("Digite a Idade: ");  
    scanf(" %d",&nova->nome);  
}
```

nova é um endereço de memória (ponteiro)!
Para acessar o conteúdo de **nova**, devemos
utilizar o operador **->**

```
int main(){  
    Pessoa p;  
    setPessoa(&p);  
}
```



Cadastro (SET)

Retorno de Função

```
typedef struct{  
    char nome[100];  
    int idade;  
}Pessoa;
```

```
Pessoa setPessoa(){  
    Pessoa nova;  
    printf("Digite o Nome: ");  
    scanf(" %[^\\n]s", nova.nome);  
    printf("Digite a Idade: ");  
    scanf(" %d",&nova.idade);  
    return nova;  
}
```

```
int main(){
```

```
    Pessoa p = setPessoa();
```

```
}
```

A função setPessoa() retorna os dados lidos pelo usuário na variável **nova**.

nova é uma variável!

Para acessar o conteúdo de **nova**, devemos utilizar o operador **.**



Consulta (GET)

```
typedef struct{  
    char nome[100];  
    int idade;  
}Pessoa;
```

```
void getPessoa(Pessoa p){  
    printf("Nome: %s\n",p.nome);  
    printf("Idade: %d\n",p.idade);  
}
```

```
int main(){  
    Pessoa p = setPessoa();  
    getPessoa(p);  
}
```

Normalmente, funções para impressão de informações não tem objetivo de alterar dados.

Podemos, portanto, invocar um procedimento enviando a *struct* como parâmetro de valor!



Edição de *Struct*

Versão Procedimento

```
void editCoordenador(Curso *c){  
    printf("Nome do Novo Coordenador: ");  
    scanf(" %[^\\n]s",c->coordenador);  
}  
  
int main(){  
    Curso c = setCurso();  
    editCoordenador(&c);  
}
```

Para editar uma struct, podemos enviar o parâmetro por referência (ponteiro da variável alvo). Dessa forma, os dados alterados na função ficarão visíveis para o programa principal.



Edição de *Struct*

Versão Função

```
Curso editCoordenador(Curso c){  
    printf("Nome do Novo Coordenador: ");  
    scanf(" %[^\\n]s",c.coordenador);  
    return c;  
}  
  
int main(){  
    Curso c = setCurso();  
    c = editCoordenador(c);  
}
```

Para editar uma struct, podemos fazer uma função que retorna para o *main* uma versão editada da estrutura recebida através de parâmetro de valor. Essa versão é similar ao cadastro de uma nova struct.



Situação Problema

- Deseja-se criar um procedimento para alimentar um array com 10 valores inteiros...
- **Como podemos enviar um array através de parâmetro?**
- O array deve ser enviado por **valor** ou **referência**?



Atenção

■ Informação Importante!

O nome de um ARRAY corresponde ao
ENDEREÇO DE MEMÓRIA do seu primeiro elemento.

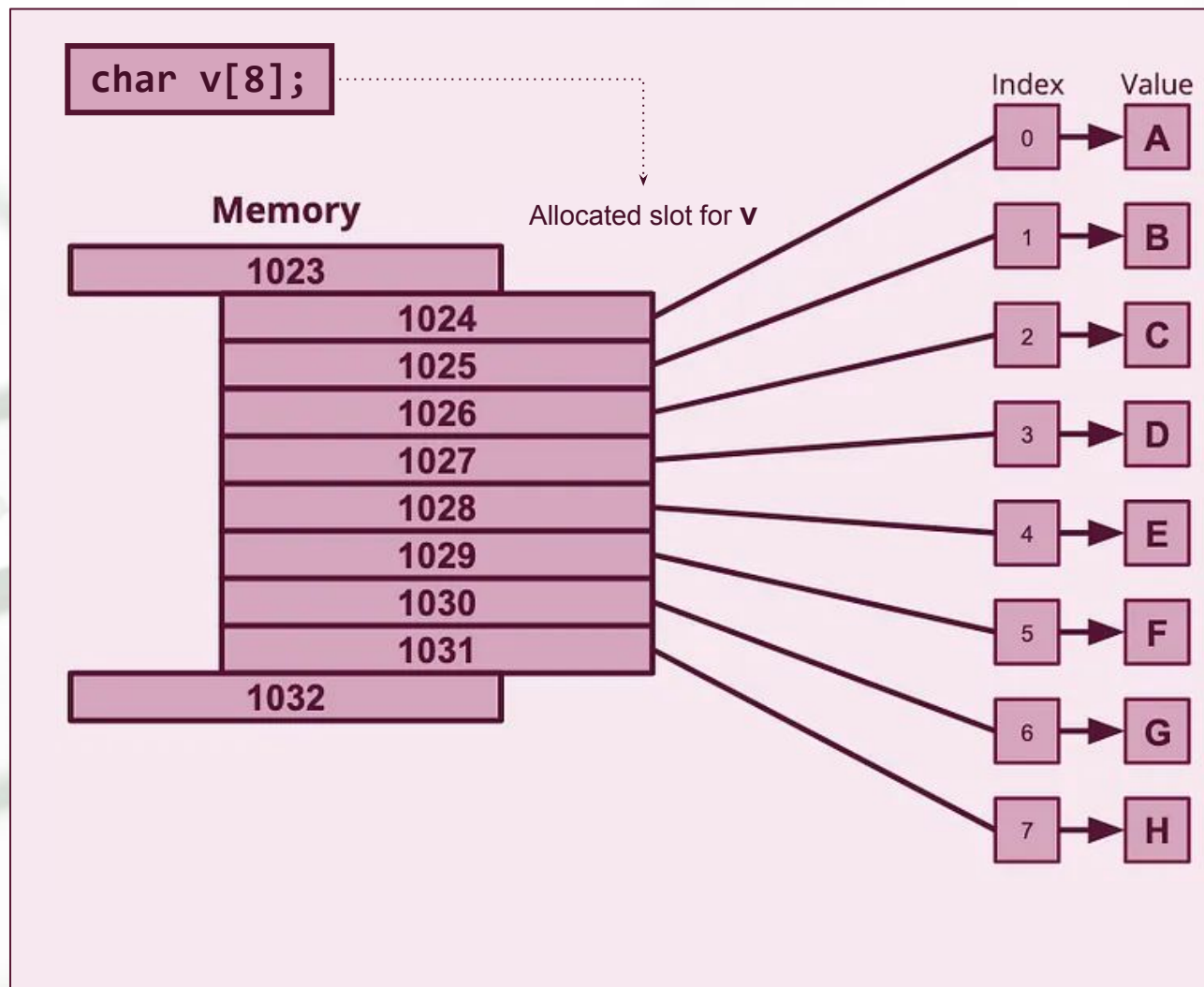
■ Portanto, no vetor: `int vet[10]`
`vet == &vet[0]`

Em outras palavras... O nome que é dado a um Array representa na verdade o endereço de memória (ponteiro) do espaço de onde se inicia.



Parâmetros - Vetor

Quando precisamos enviar um array como argumento para alguma função, basta enviar o seu nome, ou seja, o **endereço de memória do primeiro índice** deste array.

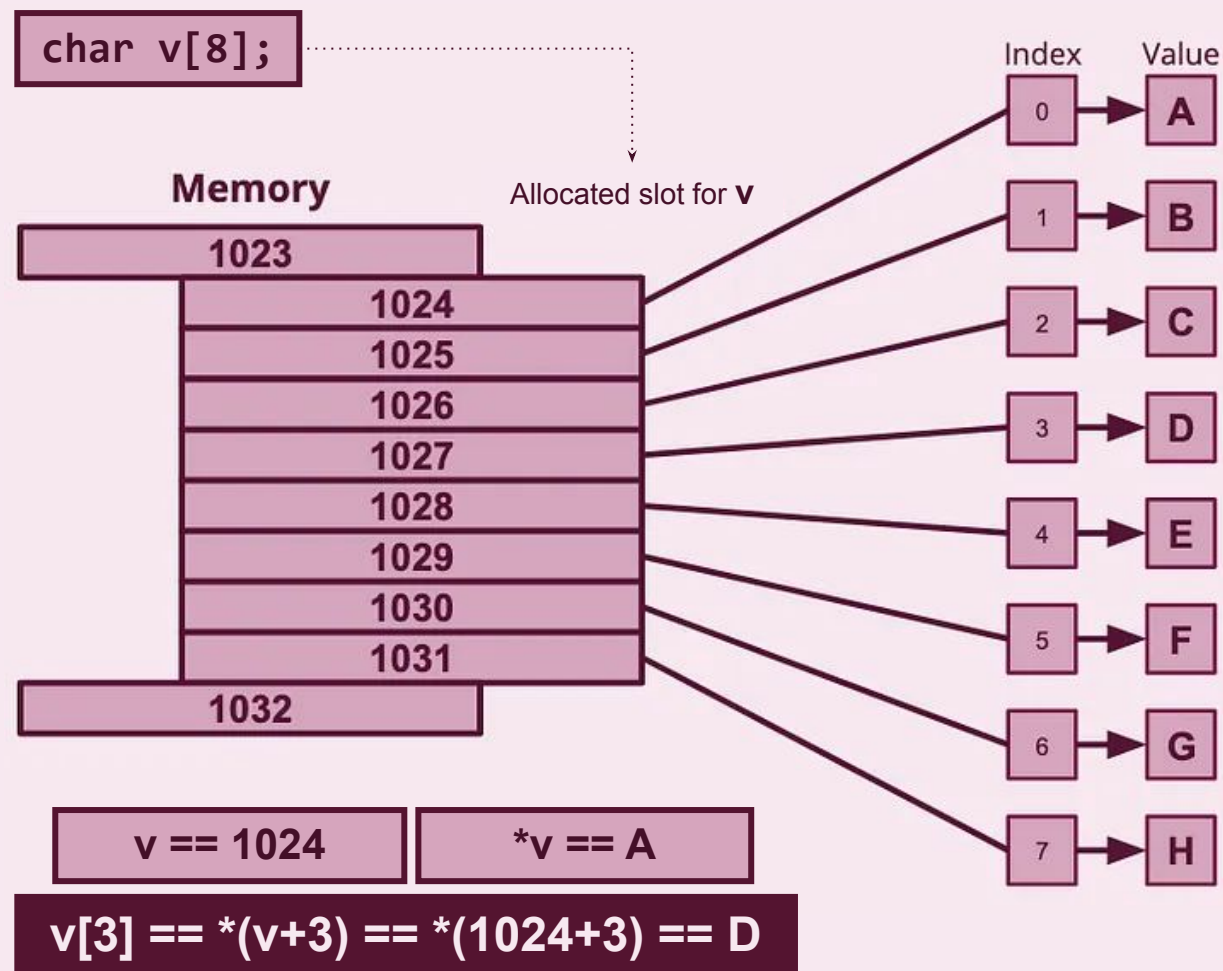




Parâmetros - Vetor

Quando precisamos enviar um array como argumento para alguma função, basta enviar o seu nome, ou seja, o **endereço de memória do primeiro índice** deste array.

Como o **espaço requerido por cada elemento do array é o mesmo** (1 Byte no caso de char), fica fácil para o **compilador calcular a posição na memória de cada elemento**.

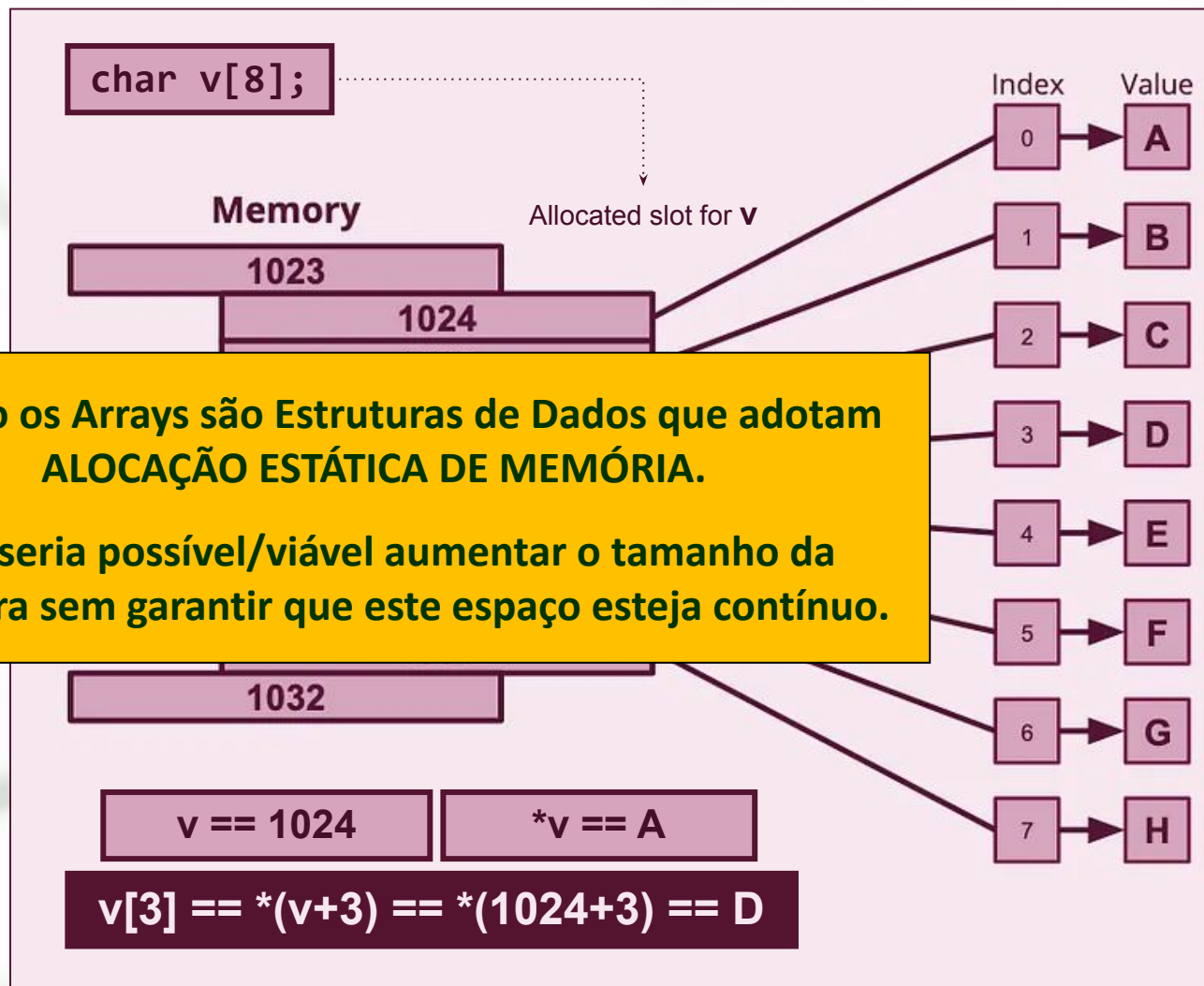




Parâmetros - Vetor

Quando precisamos enviar um array como argumento para alguma função, basta enviar o seu nome, ou seja, o endereço de memória do primeiro índice deste array.

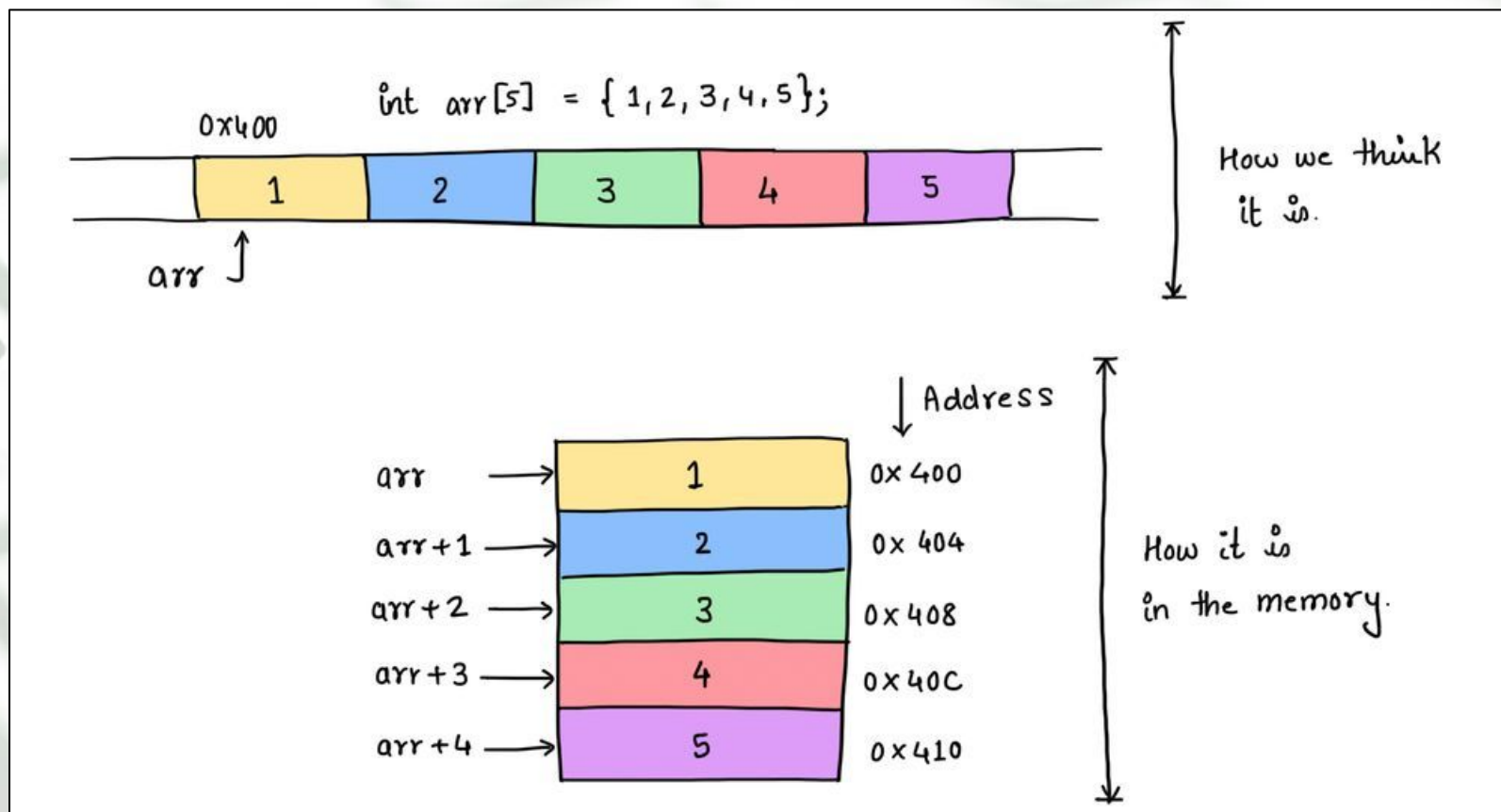
Como o espaço requerido por cada elemento do array é o mesmo (1 Byte no caso de char), fica fácil para o compilador calcular a posição na memória de cada elemento.





Parâmetros - Vetor

■ Ilustrando novamente...





Leitura e Impressão

Endereço da memória
do início do Array...

Tamanho máximo
do Array...

Arrays sempre são
enviados através de
parâmetro de referência
(ponteiro de memória)

Na função, pode-se
utilizar as sintaxes:
`int *vet`
ou
`int vet[]`

Além do ponteiro de onde
se inicia o vetor, é
NECESSÁRIO que a função
saiba o limite máximo
de elementos.

```
void preencheVetor(int *v, int n){
    for (int i=0; i < n; i++){
        printf("Digite o Numero %d: ",i+1);
        scanf(" %d",&v[i]);
    }
}

void imprimeVetor(int v[], int n){
    for (int i=0; i < n; i++)
        printf("\nO numero %d foi %d",i+1,v[i]);
}

int main(){
    int vet[10];
    preencheVetor(vet,10);
    imprimeVetor(vet,10);
}
```




Exercícios A



- Faça um programa que atenda aos seguintes requisitos:
 - Declare um vetor **V** de **N** números inteiros.
 - Faça um procedimento (*setVetor*) que recebe este vetor, e alimente-o com números primos aleatórios.
 - Crie uma função exclusiva para gerar um número primo aleatório (*getPrimo*).
 - Faça um procedimento para impressão do vetor (*getVetor*).
 - Faça uma função que recebe o vetor, e retorne o maior número (*getMaior*).
 - Faça uma função que recebe o vetor, e retorne o menor número presente no vetor (*getMenor*).



Exercícios B



- Complemente o exercício anterior de forma que:
 - Crie uma função (*sortVetor*) que recebe o vetor por parâmetro e realiza a ordenação do mesmo.
 - Crie uma função (*getAleatorio*) que retorne um valor aleatório que esteja armazenado no vetor.
 - Crie uma função (*getIndice*) que recebe o vetor e um número X. A função deve retornar o índice onde o valor X está localizado no vetor. Caso X não exista, a função retornará -1.



Exercícios C



- Faça um programa que atenda aos seguintes requisitos:
 - Declare uma matriz **M** de **8 x 5** números inteiros.
 - Faça um procedimento (*setMatriz*) que recebe esta matriz, e alimente-o com números primos aleatórios.
 - Use uma função exclusiva para gerar um número primo aleatório (*getPrimo*).
 - Faça um procedimento para impressão da Matriz, em formato Linha/Coluna (*getMatriz*).
 - Faça uma função que recebe um Vetor, e retorne a soma de elementos deste vetor.
 - Faça uma função que recebe a Matriz e, usando a função acima, retorne a linha que totaliza a maior soma.



Entendendo Matrizes

- Matriz (*2D Array*) como Argumento de Função

Observe...

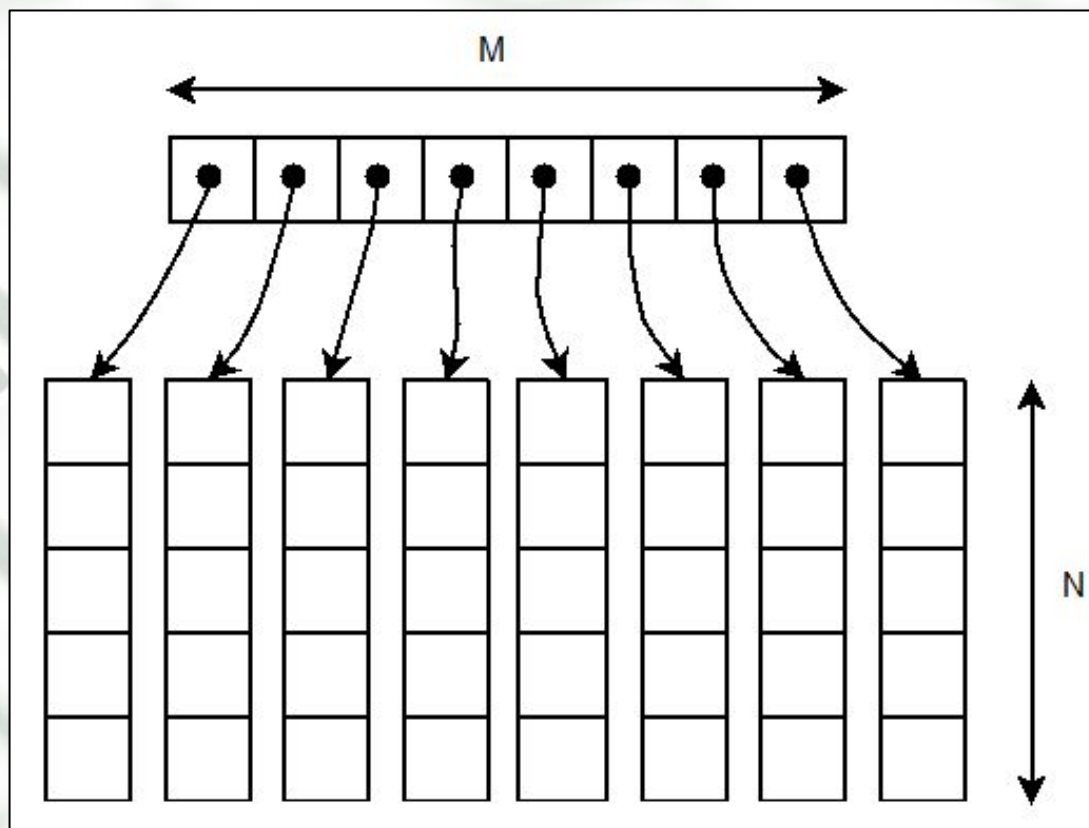
- Se um Vetor é um ponteiro para uma coleção de números em sequência...
- Uma Matriz nada mais é do que um: **Array de Ponteiros**
- Ou... um ponteiro para vários ponteiros...



Entendendo Matrizes

■ Matriz (2D Array) como Argumento de Função

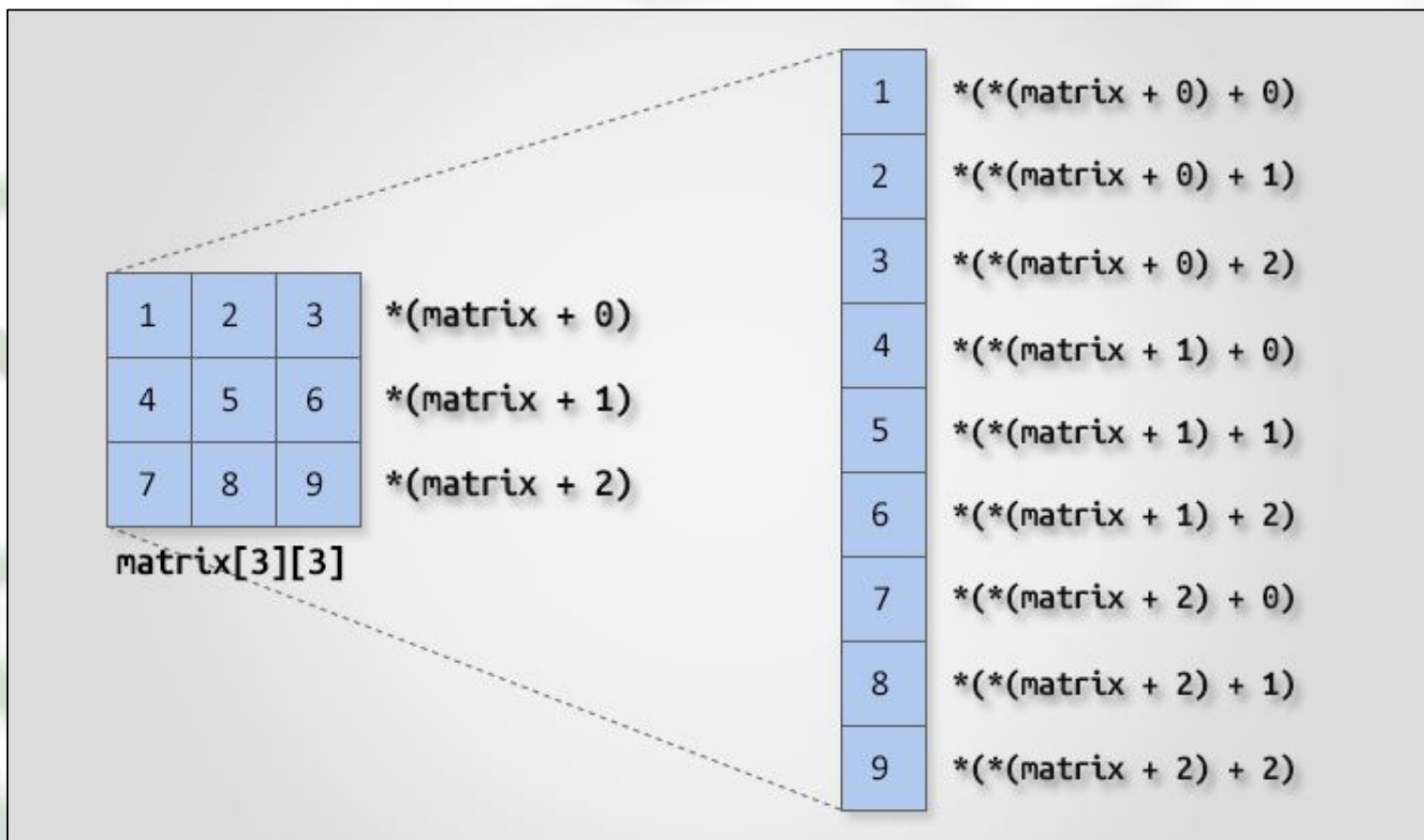
```
int main(){  
    m=8;  
    n=5;  
    int mat[m][n];  
}
```





Entendendo Matrizes

■ Matriz (2D Array) como Argumento de Função





Matriz como Argumento

Se para vetor usa-se:

```
int vet[]
```

Para matriz ALOCADA ESTATICAMENTE, usa-se:

```
int mat[][N]
```

onde N é o valor máximo de elementos de cada sub-array da matriz.

Isso é necessário para saber onde se inicia a próxima linha da matriz...

A função também deve conhecer os limites (linha/coluna) da matriz que irá acessar.

```
void preencheMatriz(int mat[][5], int l, int c){
    for (int i=0; i < l; i++)
        for (int j=0; j < c; j++)
            mat[i][j] = rand()%20;
}

void imprimeMatriz(int mat[][5], int l, int c){
    for (int i=0; i < l; i++){
        for (int j=0; j < c; j++)
            printf("%03d ", mat[i][j]);
        printf("\n");
    }
}

int main(){
    int matriz[8][5];
    preencheMatriz(matriz, 8, 5);
    imprimeMatriz(matriz, 8, 5);
}
```



Matriz como Argumento

Se para vetor usa-se:

```
int vet[]
```

Para matriz ALOCADA ESTATICAMENTE, usa-se:

```
int mat[][N]
```

onde N é o valor máximo de elementos de cada sub-array da matriz.

Isso é necessário para saber onde se inicia a próxima linha da matriz...

A função também deve conhecer os limites (linha/coluna) da matriz que irá acessar.

```
void preencheMatriz(int mat[][5], int l, int c){
    for (int i=0; i < l; i++)
        for (int j=0; j < c; j++)
            mat[i][j] = rand()%20;
}

void imprimeMatriz(int mat[][5], int l, int c){
    for (int i=0; i < l; i++){
        for (int j=0; j < c; j++)
            printf("%03d ", mat[i][j]);
        printf("\n");
    }
}

int main(){
    int matriz[8][5];
    preencheMatriz(matriz, 8, 5);
    imprimeMatriz(matriz, 8, 5);
}
```

Se os limites só forem definidos em tempo de execução, temos que utilizar ALOCAÇÃO DINÂMICA.



Situação Problema

*"Um programa não deve armazenar dados somente de uma única pessoa, mas de **N pessoas**"*





Cadastro (SET) / Forma 1

```
Pessoa setPessoa(){  
    Pessoa novo;  
    printf("Digite o Nome: ");  
    scanf(" %[^\n]s", novo.nome);  
    printf("Digite a Idade: ");  
    scanf(" %d", &novo.idade);  
    return novo;  
}
```

A função para cadastro permanece inalterada...
Uma variável "cont" realiza o controle da quantidade de pessoas cadastradas.

```
int main(){  
    Pessoa vetorPessoas[100];  
    int cont=0;  
    do{  
        vetorPessoas[cont++] = setPessoa();  
        printf("Cadastro Realizado! Continuar?\n");  
        setbuf(stdin, NULL);  
    }while(getch()=='s' && cont < 100);  
}
```




Cadastro (SET) / Forma 2

```
void setPessoa(Pessoa v[], int pos){  
    printf("Digite o Nome: ");  
    scanf(" %[^\n]s",v[pos].nome);  
    printf("Digite a Idade: ");  
    scanf(" %d",&v[pos].idade);  
}
```

A função recebe o vetor e a posição onde irá cadastrar a nova Pessoa.

```
int main(){  
    Pessoa vetorPessoas[100];  
    int cont=0;  
    do{  
        setPessoa(vetorPessoas,cont++);  
        printf("Cadastro Realizado! Continuar?\n");  
        setbuf(stdin,NULL);  
    }while(getch()=='s' && cont < 100);  
}
```



Consulta (GET)

```
void getPessoa(Pessoa p){  
    printf("Nome: %s\n",p.nome);  
    printf("Idade: %d\n",p.idade);  
}
```

Função para imprimir os dados de um registro de Pessoa.

```
int main(){  
    Pessoa vetorPessoas[100];  
    int cont=0;  
    do{  
        vetorPessoas[cont++] = setPessoa();  
        printf("Cadastro Realizado! Continuar?\n");  
        setbuf(stdin,NULL);  
    }while(getch()=='s' && cont < 100);  
    for(int i=0; i < cont; i++)  
        getPessoa(vetorPessoa[i]);  
}
```

Relatório de Pessoas Cadastradas



Edição

```
Pessoa* findPessoa(char nome[], Pessoa vet[], int cont){
    for(int i=0; i < cont; i++)
        if(!strcmp(nome,vet[i].nome))
            return &vet[i];
    return NULL;
}
```

BOA PRÁTICA!!!

Função para recuperar a posição de memória (ponteiro) de uma struct 'Pessoa' no Vetor.

```
int main(){
    Pessoa vetorPessoas[100];
    int cont=0;
    do{
        vetorPessoas[cont++] = setPessoa();
        printf("Cadastro Realizado! Continuar?\n");
        setbuf(stdin,NULL);
    }while(getch()=='s' && cont<100);
    Pessoa* alvo = findPessoa("Adriano",vetorPessoas,10)
    if (alvo) {
        getPessoa(*alvo);
        editPessoa(alvo);
    }
}
```

A função **findPessoa()** será útil para os procedimentos de Consulta, Edição, Exclusão, etc...



Boa Prática!

A função `main()` deve servir apenas para apresentar as opções do sistema ao usuário, servindo assim de "interface" do usuário para as funções executadas pelo programa!

```
int main(){
    Pessoa cadastro[100];
    int cont=0;
    int opt;
    do{
        system("clear");
        printf("Digite a Opção Desejada:\n");
        printf("[1] Cadastrar Pessoa\n");
        printf("[2] Imprimir Relatório\n");
        printf("[0] Sair\n");
        scanf(" %d",&opt);
        switch(opt){
            case 1: vetorPessoas[cont++] = setPessoa();
                    break;
            case 2: getPessoas(cadastro,cont);
                    break;
            default: return 0;
        }
        getchar();
    }while(1);
}
```



Exercícios D



- Faça um programa que atenda aos seguintes requisitos:
 - Declare um vetor V de 100 registros da struct "Produto"
 - Código de Barras
 - Descrição
 - Valor Unitário
 - Qtde. em Estoque
 - Faça um **Menu** que apresente ao usuário as seguintes opções...
 - Cadastro de Produto (apenas 1 cadastro por vez).
 - Impressão do Relatório de Estoque atualizado.
 - Consulta de Produto através do Código de Barras.
 - Edição de Produto através do seu Código de Barras.
 - Venda de Produtos (com atualização de estoque).



Exercícios E



- Um baralho é composto por 52 cartas.
- Cada carta possui as seguintes informações:
 - Valor (1 a 13), Naipe (1 a 4) e Jogador (Número do jogador possui essa carta).
- Desenvolva as seguintes funções...
 - **Função: setBaralho();**
 - Essa função deve gerar as 52 cartas (únicas) de um baralho.
 - **Função: distribuirCartas();**
 - **Deve receber por parâmetros:**
 - O **baralho** gerado;
 - A **quantidade** de cartas distribuídas para cada jogador;
 - O **número de identificação** do jogador que receberá as cartas;
 - **Função: getCartasJogador();**
 - Essa função deve imprimir as cartas que estão de posse de um **jogador identificado através de um parâmetro**.



Exercícios E



- Teste sua solução, exatamente com a seguinte função `main()`

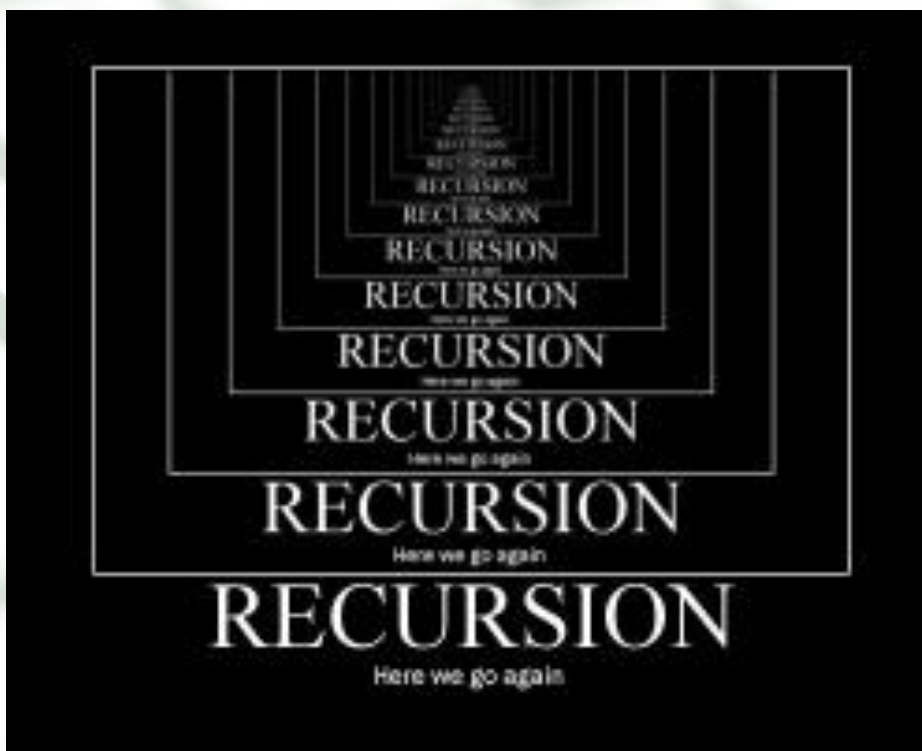
```
#define MAX_CARTAS 52

int main(){
    Carta baralho[MAX_CARTAS];
    int num_jogadores = 6;
    int num_cartas_mao = 4;
    setBaralho(baralho);
    for(int i=0; i < num_jogadores; i++)
        distribuirCartas(baralho, num_cartas_mao, i);
    printf("Cartas com cada Jogador...\n");
    for(int i=0; i < num_jogadores; i++)
        getCartasJogador(baralho, i);
}
```



Recursividade

"Para entender *Recursividade*,
antes você precisa entender *Recursividade...*"





Recursividade

- Muitos problemas possuem a seguinte característica:
 - Cada instância da solução contém uma instância menor do **mesmo problema**.
- Para esses tipos de problemas, temos as seguintes opções:
 - Se a instância é pequena, resolva!
 - Senão, reduza-a a uma instância menor, aplique o mesmo método e volte à instância original.



Recursividade

- Exemplo Prático...
- Calcule o Fatorial de 5
- $5! = 5 \times 4 \times 3 \times 2 \times 1$

Ou...



Recursividade

- Exemplo Prático...
- Calcule o Fatorial de 5
- $5! = 5 \times 4 \times 3 \times 2 \times 1$

```
fatorial(5)
```

```
5 * fatorial(5 - 1)
```

```
4 * fatorial(4 - 1)
```

```
3 * fatorial(3 - 1)
```

```
2 * fatorial(2 - 1)
```

```
1 * fatorial(1 - 1)
```

```
1
```



Recursividade

```
int fat(int n){
    if (n == 1) return 1;
    return n * fat(n-1);
}

int main(){
    printf("%d", fat(5));
}
```

Exemplo de Recursividade
A função fat é invocada dentro do seu próprio escopo

```
factorial( n ):
```

```
if n == 1:
    return 1
else:
    return n * factorial(n-1):
    if n == 1:
        return 1
    else:
```

factorial(n) =

www.mathwarehouse





Recursividade

```
int fat(int n){
    if (n == 1) return 1;
    return n * fat(n-1);
}

int main(){
    printf("%d", fat(5));
}
```

Exemplo de Recursividade
A função fat é invocada dentro do seu próprio escopo

```
factorial( n ):
if n == 1:
    return 1
else:
    return n * factorial(n-1):
    if n == 1:
        return 1
    else:
```



Relax... Estudaremos mais sobre isso em ESTRUTURAS DE DADOS 2