



**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:

```
*****  
Impressao 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**, o qual serve para sua invocação/chamada.
- Uma função pode ser invocada a **partir de qualquer outra função** (e até por ela mesma: *recursividade*).
- Uma função deve realizar **uma tarefa bem restrita e definida** (Boa prática).
- Uma função deve ser o **mais independente possível** das demais funções (Boa prática).



# Funcionamento de uma Função

- O código de uma função somente é executado a partir da sua **explícita invocação**.
- Quando uma função é invocada, a **linha de execução é interrompida e transferida para a sua execução**, e após o término, a linha de execução retoma a partir do ponto ao qual havia sido interrompida.
- As variáveis declaradas dentro de uma função **são locais** à esta (escopo local), e não interfere no restante da aplicação.



# 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?

.....

Impressao de 1 a 5

-----

1

2

3

4

5

\*\*\*\*\*





# Parametrização

## ■ Como resolver a impressão abaixo?

.....

Impressao de 1 a 5

-----

1

2

3

4

5

\*\*\*\*\*

**Declara-se 03 funções...**

`linhaPonto();`

`linhaTraco();`

`linhaAsterisco();`

?



# Parametrização

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

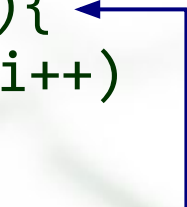
```
void print_linha(char c){  
    for (int i=0; i<=60; i++)  
        printf("%c", c);  
    printf("\n");  
}  
  
int main(){  
    print_linha('.');  
    printf("Impressão de 1 a 5\n");  
    print_linha('-');  
    for (int i=0; i<=5; i++)  
        printf("%d\n",i);  
    print_linha('*');  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



# 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;  
}
```

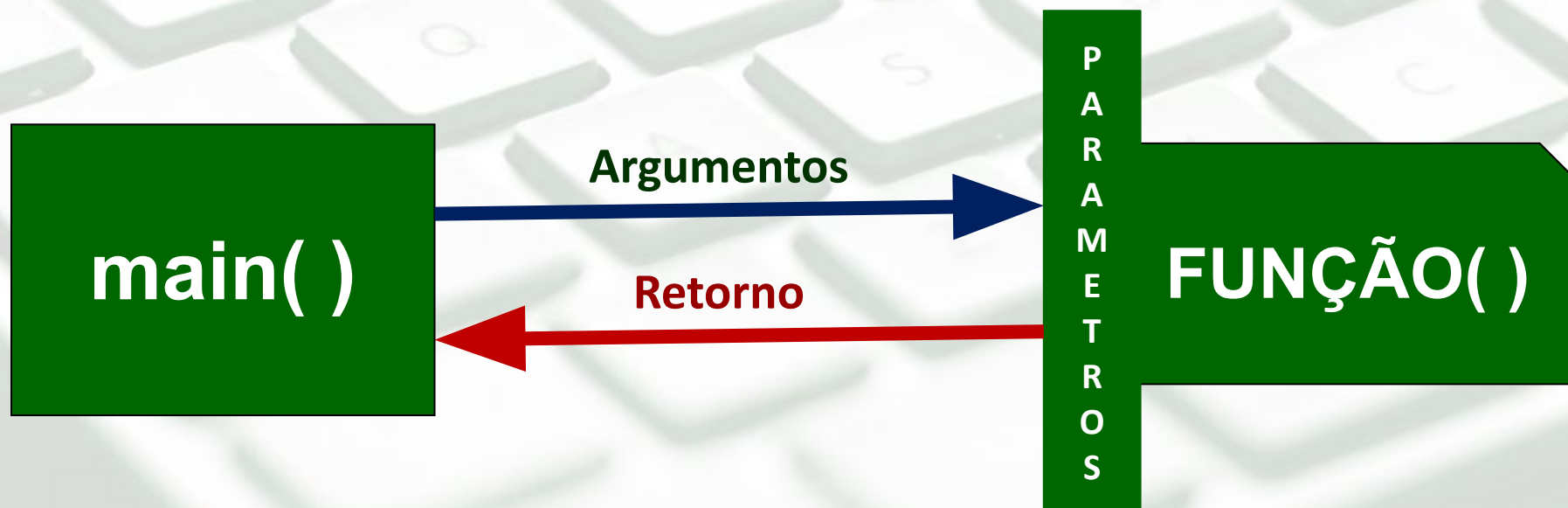
## ATENÇÃO

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))**





# Exercícios

- 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.
- Desenvolva uma função **getPrimo()** que recebe por parâmetro dois valores: X e Y. A função deve retornar um número primo sorteado no intervalo entre X e Y. A função **main()** deve preencher um vetor de N números, todos gerados pela função criada.
- Faça um programa que implemente um novo tipo de dados **Horario** (horas, minutos, segundos). Implemente uma função que **diffHorario**, que recebe por parâmetro duas variáveis do tipo Horário. A função retorna uma variável também do tipo Horário, que representa a diferença de horas, minutos e segundos dos dois parâmetros enviados. Faça uma função **printHorario** para imprimir a struct Horário em um formato padrão.
- Acrescente ao problema anterior uma **função exclusiva** para preenchimento da struct Horário.



# 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"**

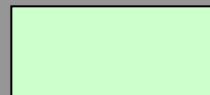
Área de Memória Reservada  
para main()

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()

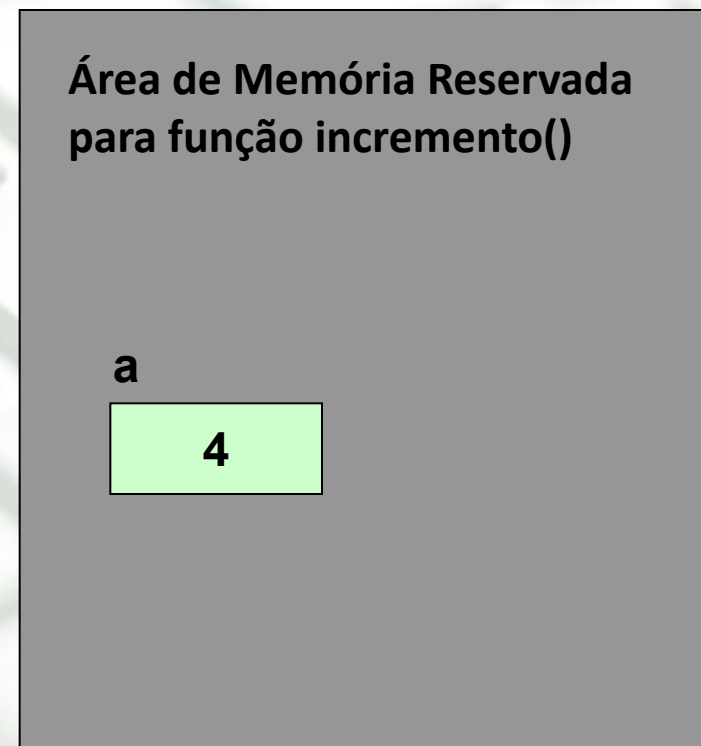
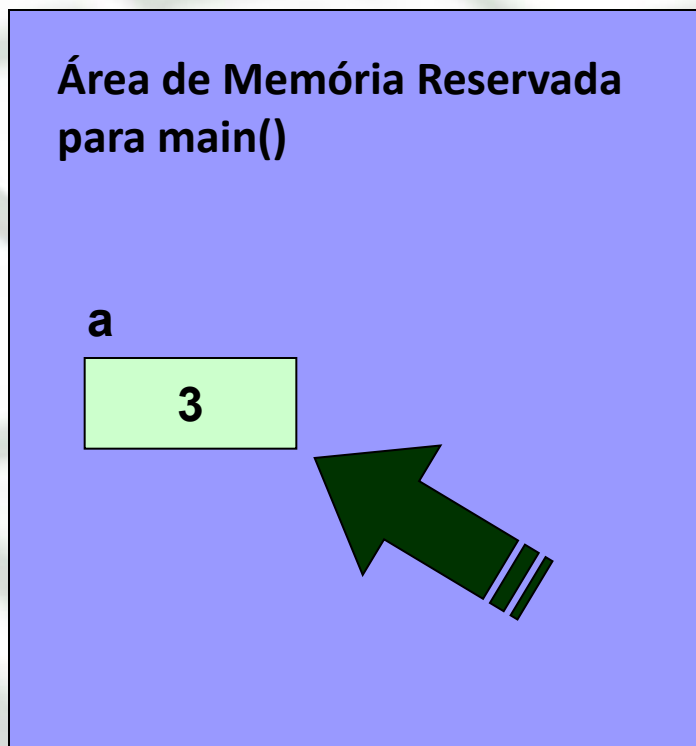
a

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.



# 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.



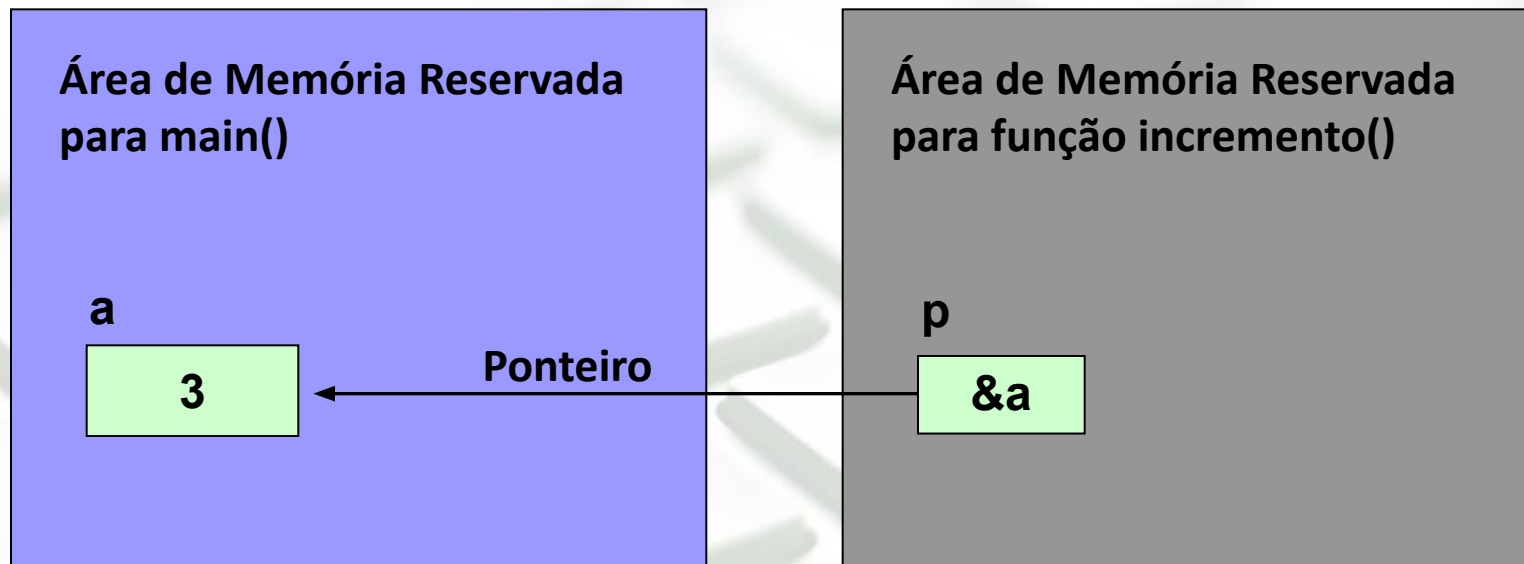
# Passagem por Referência

```
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.





# Introdução a Ponteiros

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

```
tipo *ptr;
```

Onde...

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



# Introdução a Ponteiros

```
int main(){  
    int *ptr;  
    int a = 9;  
    ptr = &a;  
    *ptr += 1;  
    printf("A variável A está alocada  
           no endereço %x e possui  
           valor %d", ptr, a);  
}
```

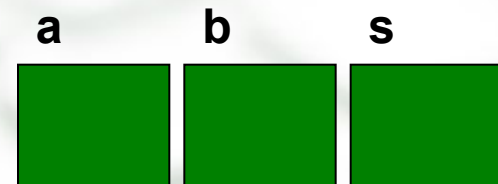
O operador & significa  
"o endereço de..."

\*ptr significa  
"o valor apontado por ptr"



# Acompanhe...

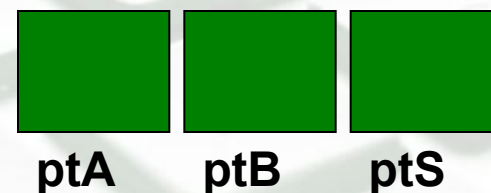
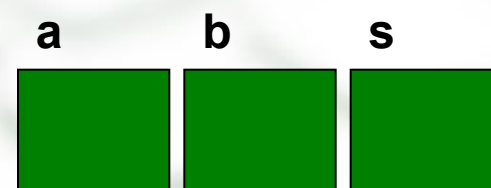
```
int main(){  
→ int a,b,s;  
  int *ptA,*ptB,*ptS;  
  a = 2;  
  b = 3;  
  ptA = &a;  
  ptB = &b;  
  ptS = &s;  
  *ptS = *ptA + *ptB;  
}
```





# Acompanhe...

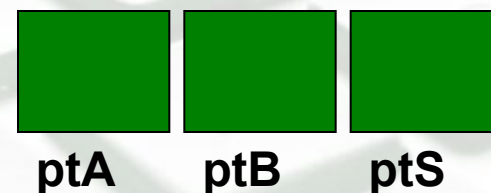
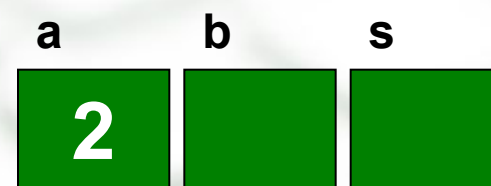
```
int main(){  
    int a,b,s;  
    → int *ptA,*ptB,*ptS;  
    a = 2;  
    b = 3;  
    ptA = &a;  
    ptB = &b;  
    ptS = &s;  
    *ptS = *ptA + *ptB;  
}
```





# Acompanhe...

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

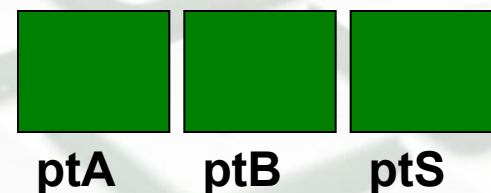
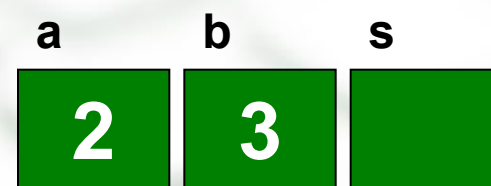






# Acompanhe...

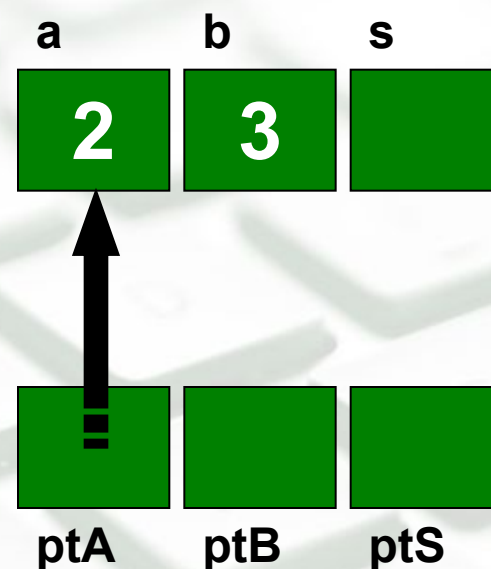
```
int main(){  
    int a,b,s;  
    int *ptA,*ptB,*ptS;  
    a = 2;  
    → b = 3;  
    ptA = &a;  
    ptB = &b;  
    ptS = &s;  
    *ptS = *ptA + *ptB;  
}
```





# Acompanhe...

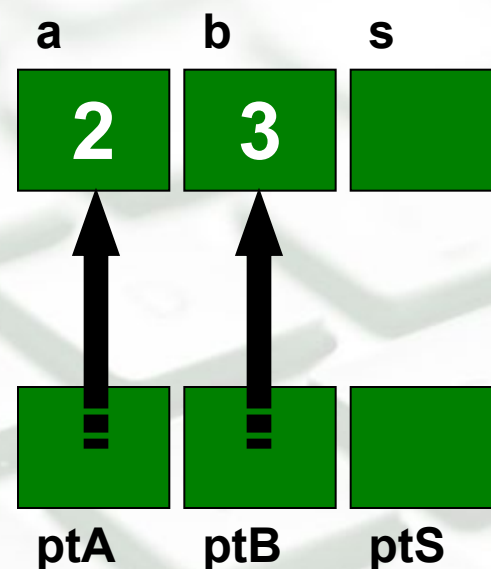
```
int main(){  
    int a,b,s;  
    int *ptA,*ptB,*ptS;  
    a = 2;  
    b = 3;  
    → ptA = &a;  
    ptB = &b;  
    ptS = &s;  
    *ptS = *ptA + *ptB;  
}
```





# Acompanhe...

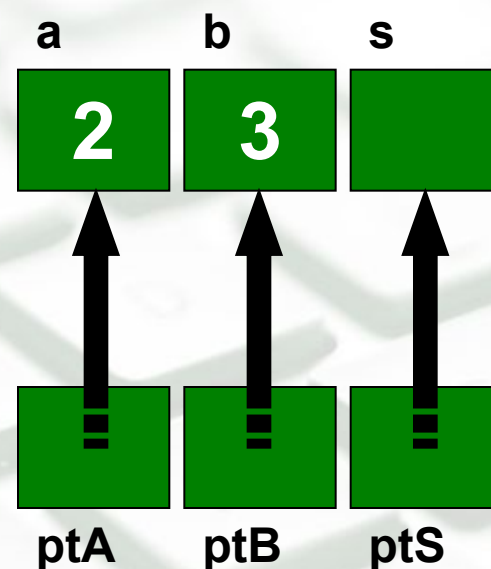
```
int main(){  
    int a,b,s;  
    int *ptA,*ptB,*ptS;  
    a = 2;  
    b = 3;  
    ptA = &a;  
    → ptB = &b;  
    ptS = &s;  
    *ptS = *ptA + *ptB;  
}
```





# Acompanhe...

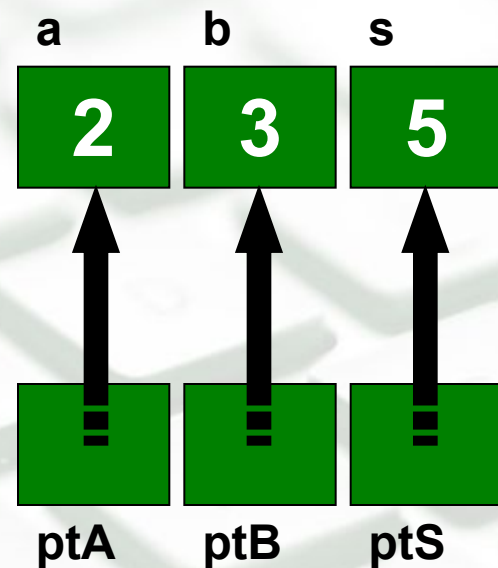
```
int main(){  
    int a,b,s;  
    int *ptA,*ptB,*ptS;  
    a = 2;  
    b = 3;  
    ptA = &a;  
    ptB = &b;  
    → ptS = &s;  
    *ptS = *ptA + *ptB;  
}
```





# 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?**



# 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 dá pra fazer dos dois modos?**



# Exercícios

- Faça um programa que declare uma struct chamada **Cotação**. As informações desta struct são:
  - float dolar;
  - float euro;
  - float libra;
- Crie funções para que o usuário informe o valor da cotação atual em cada moeda.
- Crie funções 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?
- Seria possível criar uma **ÚNICA** função para calcular a conversão de cada moeda?





# 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);  
}
```

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

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



# 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 vetor de 10 variáveis do tipo inteiro...
- Como podemos enviar um vetor como parâmetro?
- O vetor deve ser enviado por **valor** ou **referência**?



# Atenção

## ■ Informação Importante!

- O nome de um **VETOR** corresponde ao **ENDEREÇO** de memória do seu primeiro elemento.

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

O próprio nome do vetor, é em si mesmo, um endereço (ponteiro) de memória, e indica onde é o seu início!

# Parâmetros - Vetor

- Em C, quando enviamos um vetor por parâmetro, na verdade enviamos o **endereço de memória do primeiro índice do vetor**, e não o vetor em sua totalidade.

```
int vet[10];
```

Endereço Memória	6003	6004	6005	6006	6007	6008	6009	6010	6011	6012
Índice	0	1	2	3	4	5	6	7	8	9
Valor	9	5	1	4	6	8	4	7	0	6

```
vet == 6003
```

```
*vet == 9
```

```
*(vet+3) == vet[3] == 4
```

# Leitura e Impressão

Vetores 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 início do vetor, é OBRIGATÓRIO, enviar o limite máximo do vetor.

Endereço da memória de onde começa o vetor...

Qtde. máxima de elementos que esse vetor possui...

```
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

- 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 C

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

```
int main(){
    Carta baralho[52];
    int num_jogadores = 6;
    int num_cartas_mao = 4;
    setBaralho(baralho,52);
    for(int i=0; i < num_jogadores; i++)
        distribuirCartas(baralho, 52, num_cartas_mao, i);
    printf("Cartas com cada Jogador...\n");
    for(int i=0; i < num_jogadores; i++)
        getCartasJogador(baralho, 52, i);
}
```





# Situação Problema

- "O programa não deve armazenar dados somente de 1 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);
}
```





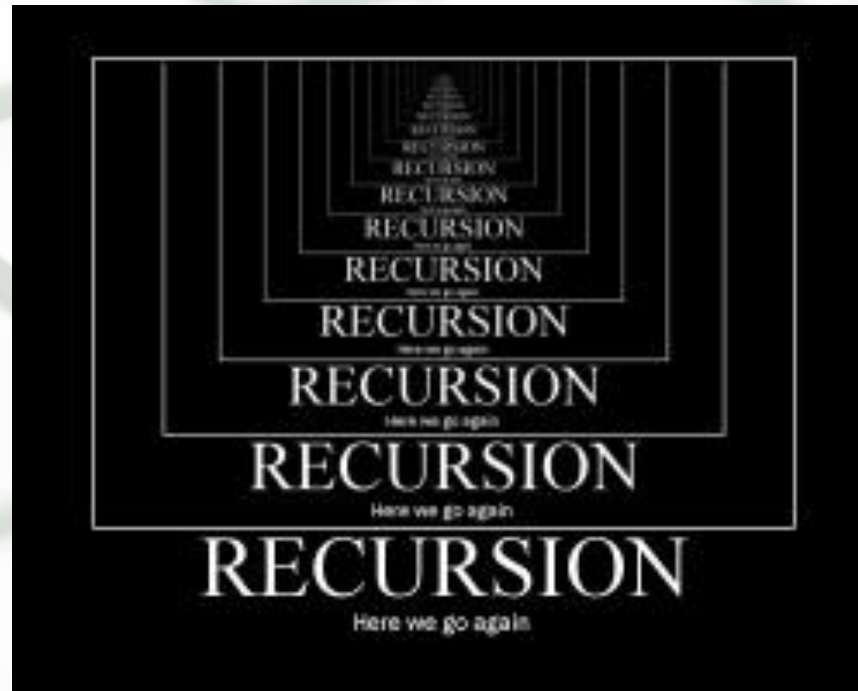
# Exercícios D

- 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).



# 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...

```
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(4));  
    getch();  
}
```

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







# Recursividade

## ■ Exemplo Prático...

NÍVEL →						
ORDEM DE EXECUÇÃO ↓	Fatorial(5)	Fatorial(4)	Fatorial(3)	Fatorial(2)	Fatorial(1)	
	5 * Fatorial(4)					
	→	4 * Fatorial(3)				
		→	3 * Fatorial(2)			
			→	2 * Fatorial(1)		
				→	Fatorial(1)	
					1	End Function de Fatorial(1)
				←	2 * 1	
				←	2	End Function de Fatorial(2)
			←	3 * 2		
			←	6		End Function de Fatorial(3)
		←	4 * 6			
		←	24			End Function de Fatorial(4)
	←	5 * 24				
	←	120				End Function de Fatorial(5)