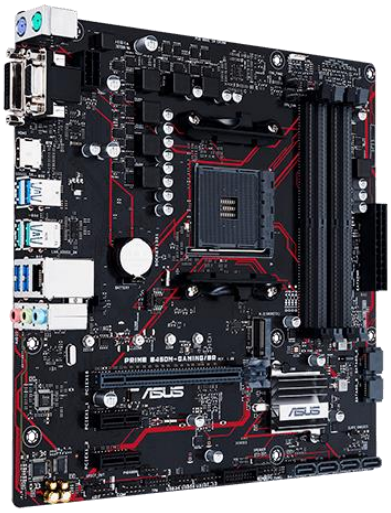
FUNÇÕES



Não deveríamos falar sobre funções? Sim, vamos falar, mas para introduzir a discursão vamos de analisar a composição dessa placa mãe?

Bom, cada componente tem seu espaço reservado e seu local de encaixe. Pergunto: caso um pente de memória venha a queimar, o dono do computador deve descartar, trocar, todo o computador? Não, né? Mas, por quê? Porque o computador é modularizado. Cada unidade tem sua responsabilidade, sua função, e essa composição permite maior facilidade para a troca de itens, ajuste e detecção de problemas.

Excelente! Agora podemos falar sobre as funções no contexto da programação. Pois elas possuem total relação com o conceito de modularização. Na programação a modularização permite a construção de grandes programas dividido em pequenas partes, os subprogramas, as funções. O primeiro módulo que temos contato desde o início da nossa disciplina é o programa principal (o main), ele é um subprograma, uma função, que a partir dele podemos "chamar" outras funções, e outras funções podem "chamar" umas às outras.

Bom, vamos conhecer como construí-las e "chama-las"?

Declarando Funções

Como mencionado anteriormente, cada função possui responsabilidade específica e desta forma para realiza suas operações, por vezes necessitam receber valores de entrada, tais valores chamamos de **parâmetros**, e ao final de todo o processo as funções podem **retornar valores**, como resultados, para quem as chamou.

E todo esse processo definimos na declaração das funções, na sua assinatura, que sintaticamente, definimos como:

```
<tipo_do_valor_retorno> <nome_na_função> ( <lista_de_parâmetros>) { < corpo_da_função> }
```

- <tipo_do_valor_retorno> como falado, uma função pode retornar ou não valores, é opcional. Quando retornado deve ser indicado qual o tipo desse valor, por exemplo: int, float, char.. Caso não retorne nenhum valor, então deve ser informado **void**. E caso o campo fique vazio, o compilador C assume que será **int**.

- <nome_na_função> o nome de uma função respeita as mesmas regras para o nome das variáveis.
- <lista_de_parâmetros> é opcional uma função receber valores, quando não recebem o campo pode ser deixado vazio ou pode ser declarado void. Mas caso passem a receber valores, a sintaxe estabelecida é:

```
<tipo_da_variável_1> <nome_da_variável1>, <tipo_da_variável_2><nome_da_variável2>, [...]
```

- < corpo_da_função> corresponde ao bloco de comandos da função, todas as variáveis que forem declaradas dentro dela são "visíveis" apenas no bloco dessa função. Um detalhe interessante, é que se a função pode retornar um valor.

Vamos ver um exemplo? Bora!

Exemplo

Olha só, pensemos em uma função que recebe dois números inteiros e retorna o produto dessa soma como resultado. Vamos listar os itens que a assinatura da função dele possuir:

- <tipo_do_valor_retorno> bom, se a responsabilidade dessa função é de retornar a soma de dois inteiros, então o valor retornado será um inteiro também. Show! Bora pro próximo item!
- <nome_na_função> se ela deve realizar uma soma, é fácil! Vamos chamar de soma.
- <lista_de_parâmetros> bom, se nossa função vai receber dois inteiros, vamos dar o nome para esses parâmetros e indicar que eles são inteiros: int num1, int num2.

Dessa forma, a assinatura para nossa função de exemplo, será:

```
int soma (int num1, int num2);
```

Observa só, não declaramos aqui o corpo da função, , colocamos apenas a **assinatura**. Quando declaramos só a assinatura "fechamos" essa instrução com "**;**". Você pode perguntar: declarar a assinatura já é construir a função? Não, não é! Esta linha corresponde **apenas a assinatura** da função. Jaja vamos descobrir sua funcionalidade. Bom, mas vamos construir a função completa? Bora lá!

Já descobrimos a assinatura da função, falta só pensarmos no corpo:

```
int soma (int num1, int num2){
        int valor; //variável que guardará a soma de num1 e num2

        valor = num1+num2;

        return valor;
}
```

Observa, como a responsabilidade da função é retornar um valor, fizemos uso da instrução "**return**". H'm, isso te faz lembrar do comando "*return 0*" que as vezes fazemos uso na função *main*? Show!

Vamos pensar que esta mesma função não retornasse valores? O que modificaríamos? <tipo_do_valor_retorno> correto? Isso mesmo. Como não haverá valor de retorno, informamos **void** nessa seção.

```
void soma (int num1, int num2){
        int valor; //variável que guardará a soma de num1 e num2

        valor = num1+num2;
        printf("A soma é igual a %d", valor);
}
```

Olha que legal, podemos fazer uso de qualquer função de outras bibliotecas como scanf e printf dentro das nossas funções. E podemos mais! Podemos chamar nossas funções dentro de outras funções. Top, né? Vamos lá entender como podemos "chamar" as funções.

Chamando as funções

Em C podemos declarar as funções dentro de bibliotecas, em arquivos .c individuais, ou no mesmo arquivo .c em que a função principal esteja.

Vamos testar o uso no mesmo arquivo do main? Então, **ATENÇÃO!** Da mesma forma que as variáveis, uma função tem que ser declarada antes de ser chamada (referenciada) no programa.

Vamos fazer uso da função soma do exemplo anterior? Bora!

Então primeiramente vamos definir a função e depois vamos *chamar* (usar) na função "main":

```
#include<stdio.h>
#include<stdlib.h>
int soma (int num1, int num2){
        int valor;

        valor = num1+num2;

        return valor;
}

int main (void){
        int x, y, resultado;

        printf("Informe dois valores:");
        scanf("%d %d", &x, &y);

        resultado = soma(x,y);
        printf("A resultado da soma eh = %d", resultado);
}
```

Podemos fazer uso da função soma, pois ela foi declarada antes do seu uso. x,y são conhecidos apenas pela função main. Os parâmetros num1, num2, recebem os valores de x e y.

Opa! Opa! Liguem o alerta! X e Y não se tornar as variáveis num1 e num2, quando chamamos a função soma, passamos apenas os valores de X e Y, a este processo chamamos de **PASSAGEM POR VALOR**.

Mas será que podemos fazer com que num1, num2 e valor, assumam os endereços de X, Y e resultado? H'm, a palavra "**endereço**" te fez lembrar de algo? Sim, sim! **PONTEIROS**! Vamos fazer uso deles?

```
#include<stdio.h>
#include<stdlib.h>
void soma (int *num1, int *num2, int *valor){

        *valor = *num1+*num2;

}

int main (void){
        int x, y, resultado;

        printf("Informe dois valores:");
        scanf("%d %d", &x, &y);

        soma(&x,&y, &resultado);
        printf("A resultado da soma eh = %d", resultado);
}
```

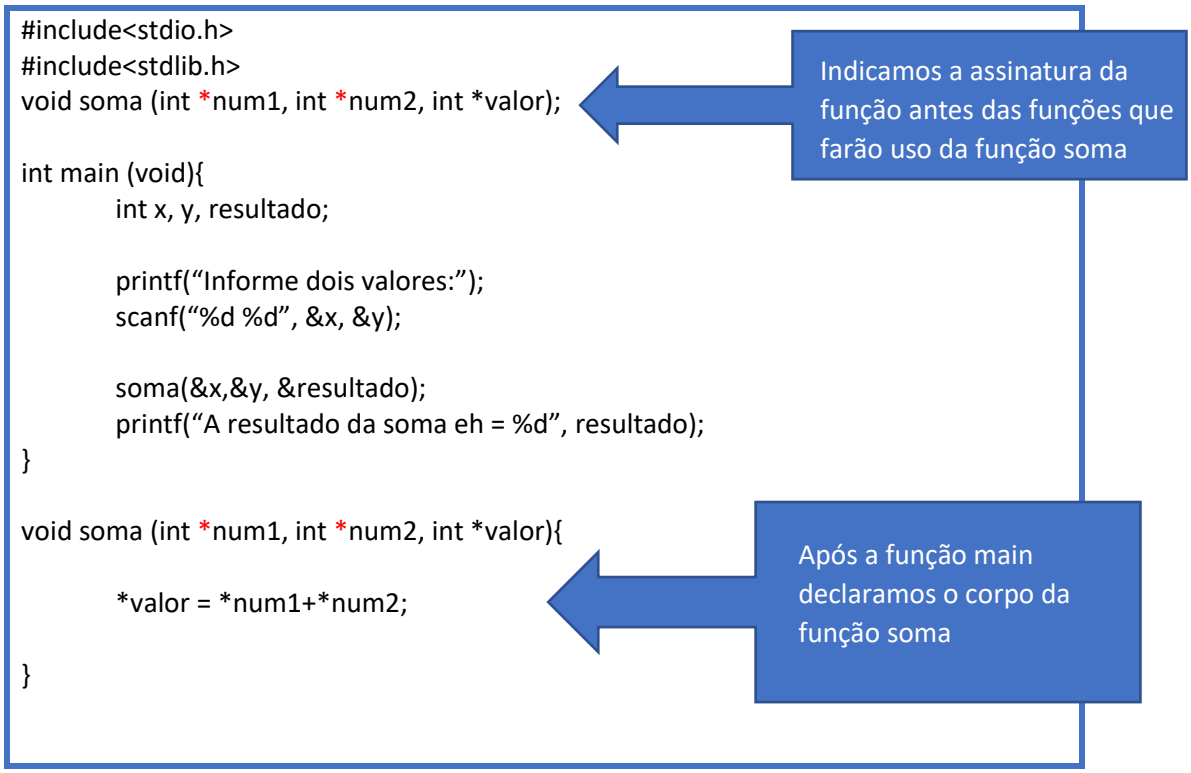
*num1, *num2 e *valor indicam "valores apontados por", neste caso x, y e resultado

Passando os endereços de x, y e resultado, pois a assinatura da função indica que o tipo dos parâmetros são ponteiros

Esse tipo de processo em que não passamos os valores para uma função e sim os endereços das variáveis, para que seus valores sejam consultados ou alterados (como foi o caso da variável resultado), chamamos de **PASSAGEM POR REFERÊNCIA**.

Agora sim, faz todo o sentido o uso de ponteiros. Super prático, né? Sem o uso de ponteiros, uma função só pode retornar um único valor, com uso da instrução *return*. Porém, com uso de ponteiros, podem retornar mais de um valor, modificando as variáveis apontadas.

Bom, mas vamos fechar o processo de "chamada" de uma função no mesmo arquivo do main? Ué, não acabamos de fazer uso? Não, há mais uma maneira. Lembra da assinatura? Vamos ver uma aplicação dela. Mas já adianto, há outras aplicações (em bibliotecas e projetos).



Em linhas gerais as funções são isto, módulos que nos permitem delegar funcionalidades e responsabilidades, que proporcionam melhor manutenção, identificação de erros e gerência de memórias.

Mas antes de finalizamos, agora que vamos estruturar nossos programas em funções, vamos entender mais sobre variáveis?

Variáveis Locais e Globais

Variáveis Locais são aquelas visíveis no bloco de comando que foram declaradas. Ou seja, posso ter duas funções diferentes em que cada uma possua uma variável inteira x, por exemplo. E não teremos problemas, os x's serão distintos e serão *variáveis locais* de cada função. E o mesmo conceito cabe para os outros blocos que já conhecemos (if, else, for, while, do-while):

```
void funcao1(void){
        int x, y, z;

        [...]
}
```

```
void funcao2(void){
        int x, k;

        for (...){
          float x;
              ...
        }
}
```

Variáveis Globais, como o nome já indica, são conhecidas por todas as funções. Logo, devem ser declaradas *fora* de todas elas (incluindo a função principal main):

```
#include<stdio.h>
#include<stdlib.h>
int t, z;
void funcao1(void){
{
        int x,y;
        [...]
}

void funcao2(void){
{
        int x,y;
        [...]
        z=10;
        ...
}

int main (void)
{
        int i;
        [...]
}
```

É isso, fizemos aqui um tour rápido sobre o universo das funções, agora é visitar os vídeos sugeridos lá no SIGAA e praticar bastante! Vamos lá, nos encontramos no fórum de discursão.