



PROGRAMAÇÃO 101

dia 1

- Introdução
- Variáveis
- Operadores
- Boas Práticas

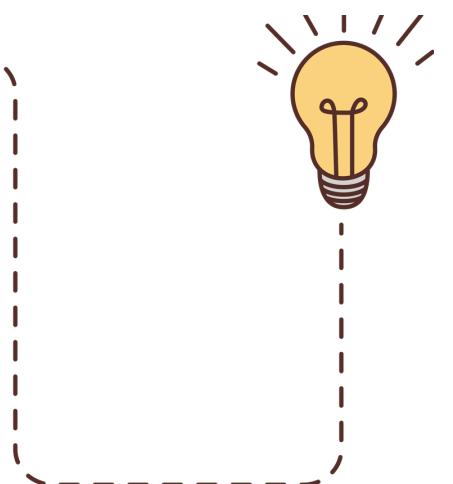
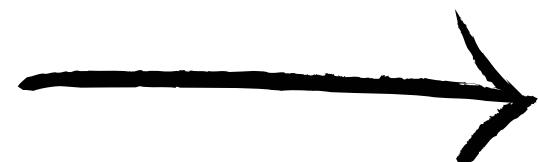
dia 1

Introdução

O que é programação?



Problema



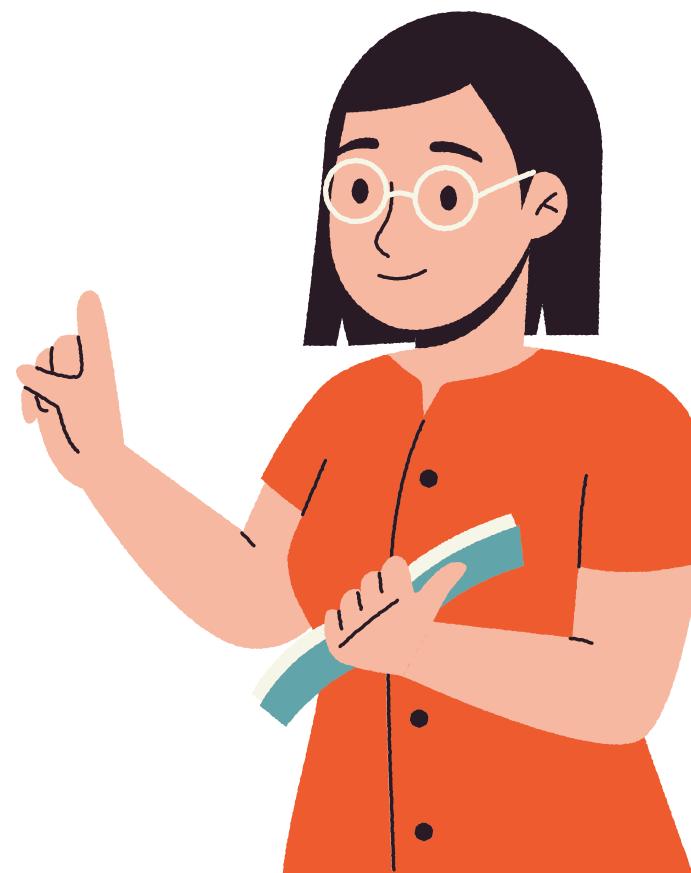
Solução (ou algoritmo)

O que é programação?



Mas o que é um algoritmo?

**Algortimo é uma sequência
de passos necessária para
atingir um objetivo**

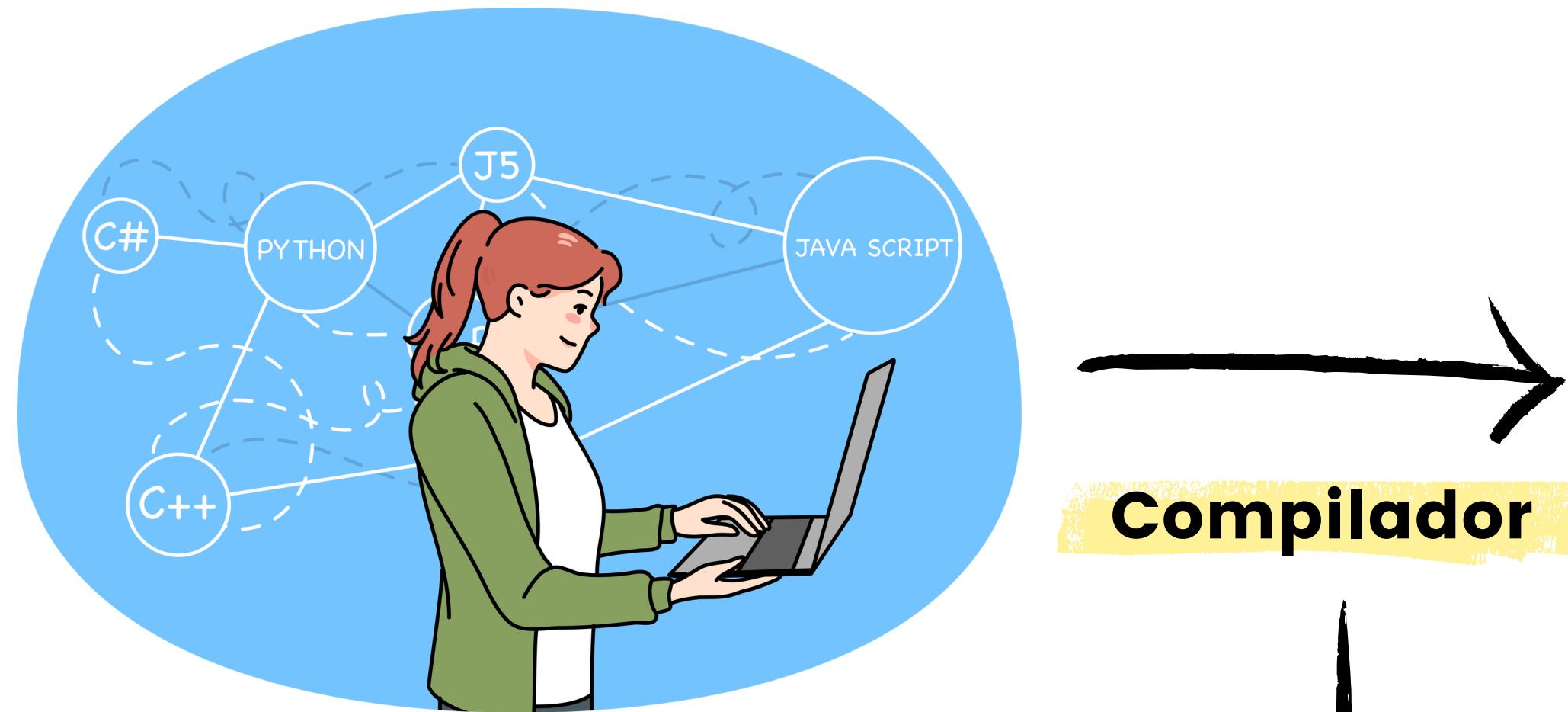


O que é programação?



- . Algoritmo de compras
- . Abrir a geladeira
- . **Se** há ovos
- . . . Não fazer nada
- . **Do contrário**
- . . . Pegar carteira
- . . . Ir até o supermercado
- . . . Comprar ovos
- . . . Voltar para casa
- . Fim se
- . Fim algoritmo

O que é programação?

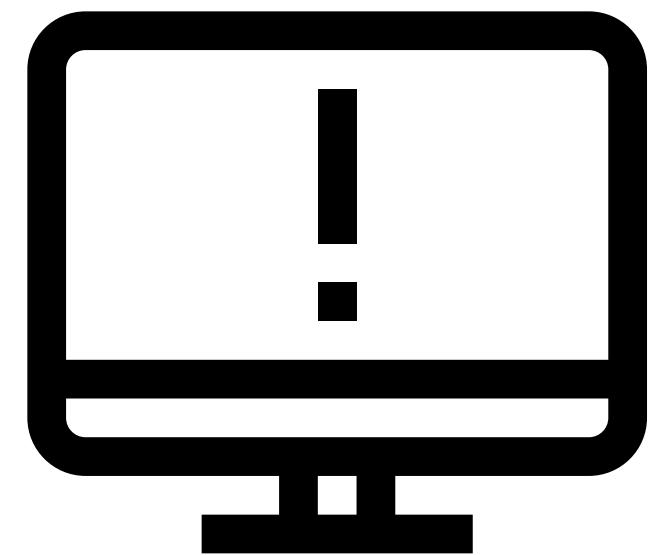


**algoritmo -> linguagem de
programação**

**Transforma o código
fonte para binário**

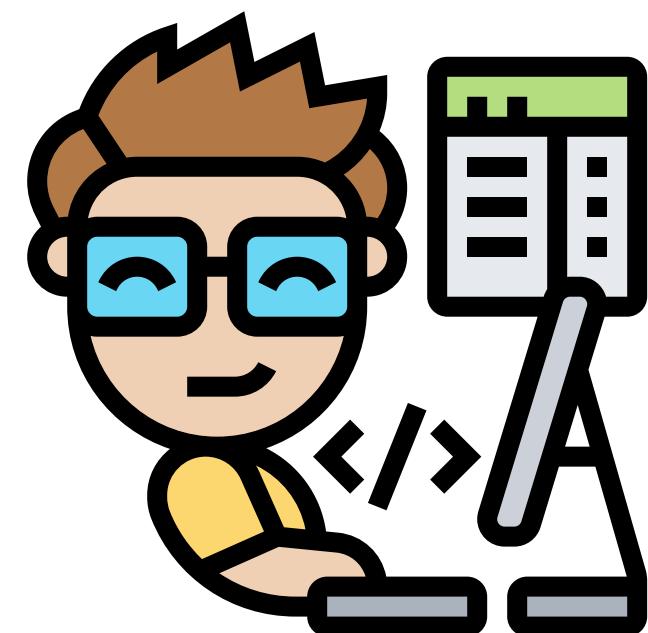
O que é programação?

Deu errado?



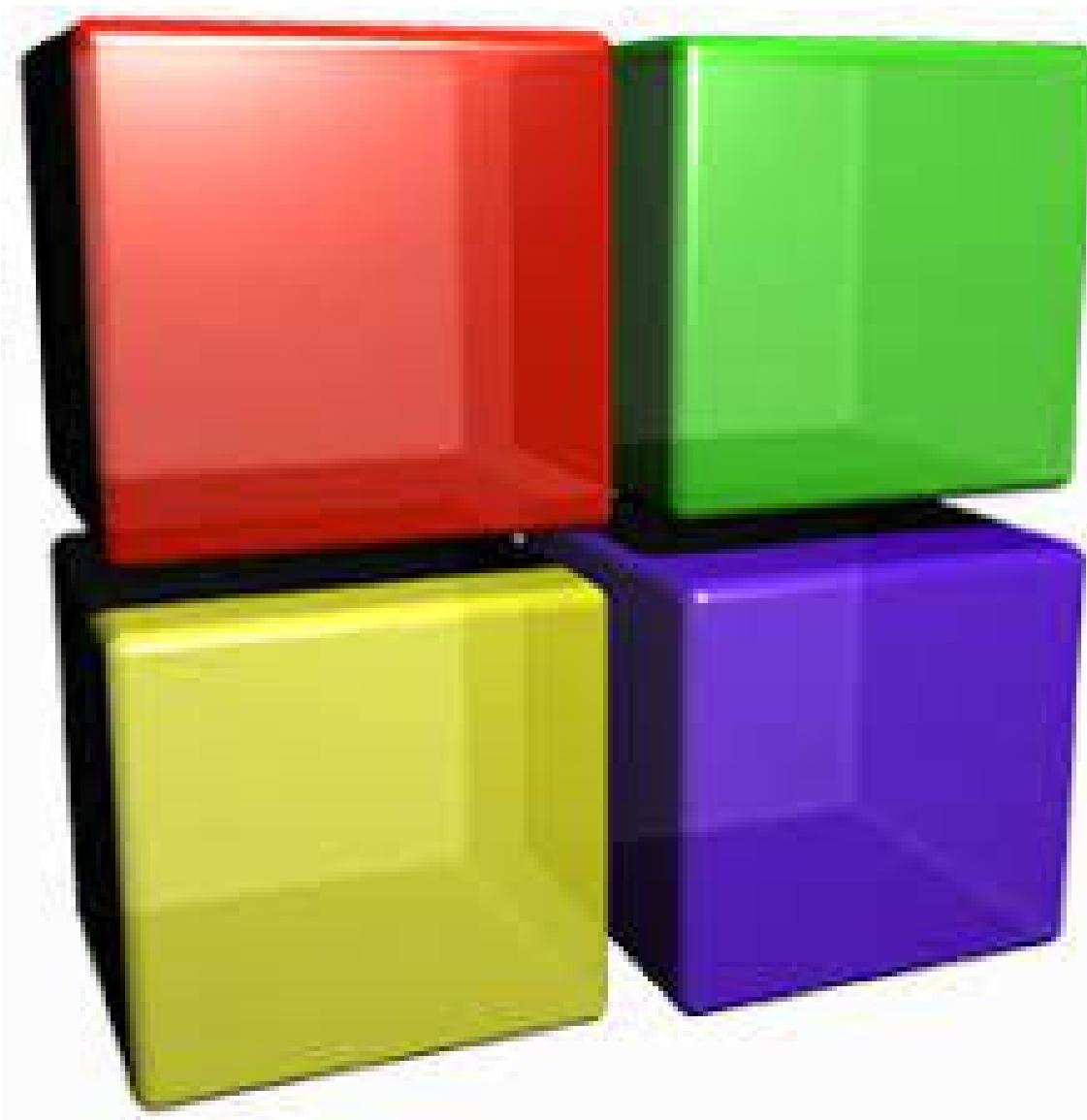
**Nem sempre o nosso
programa está correto**

Deu certo?



**Você vai poder usar o
programa no seu
computador**

O que vamos utilizar?

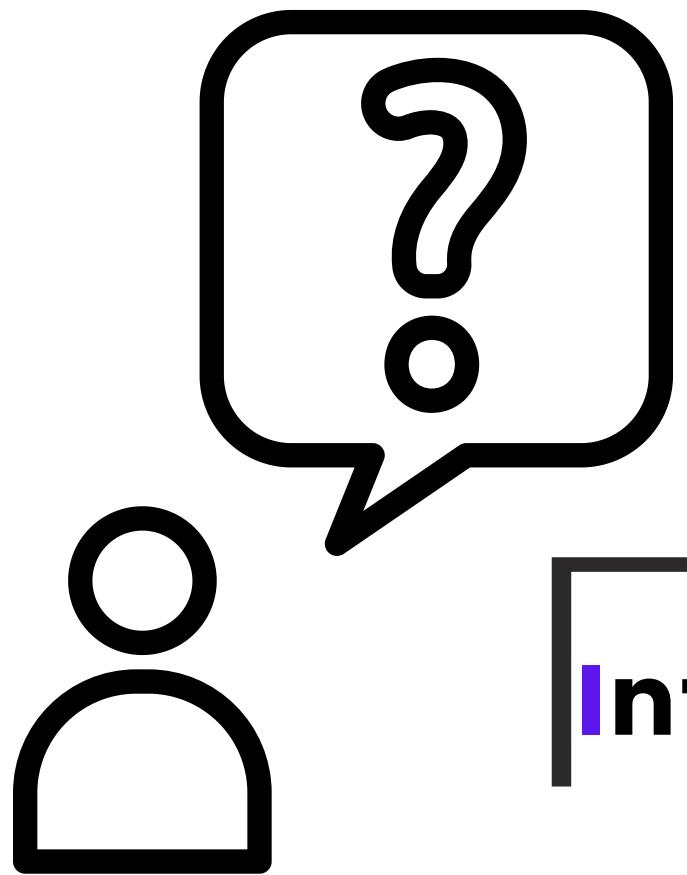


Codeblocks

- **tem um editor de texto**
 - **tem um compilador**
 - **tem um debugador**

IDE

Codeblocks



O que é uma IDE?

Integrated Development Environment

Auxilia no desenvolvimento de
aplicações

Ajuda a criar aplicações mais rápido

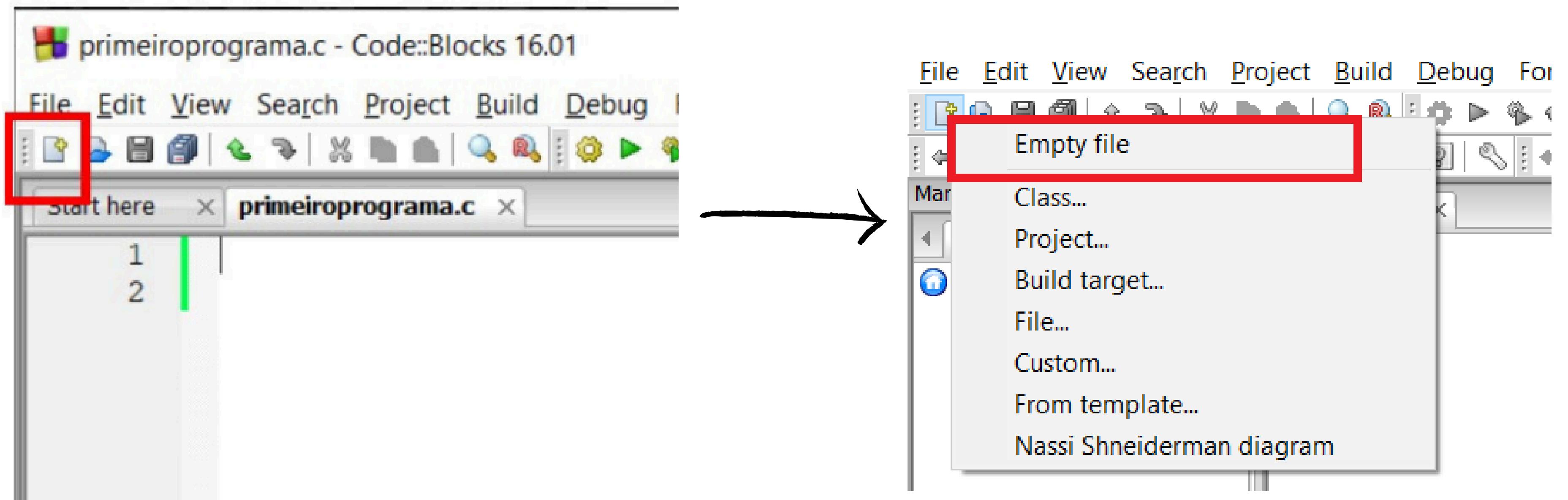


Mãos à obra

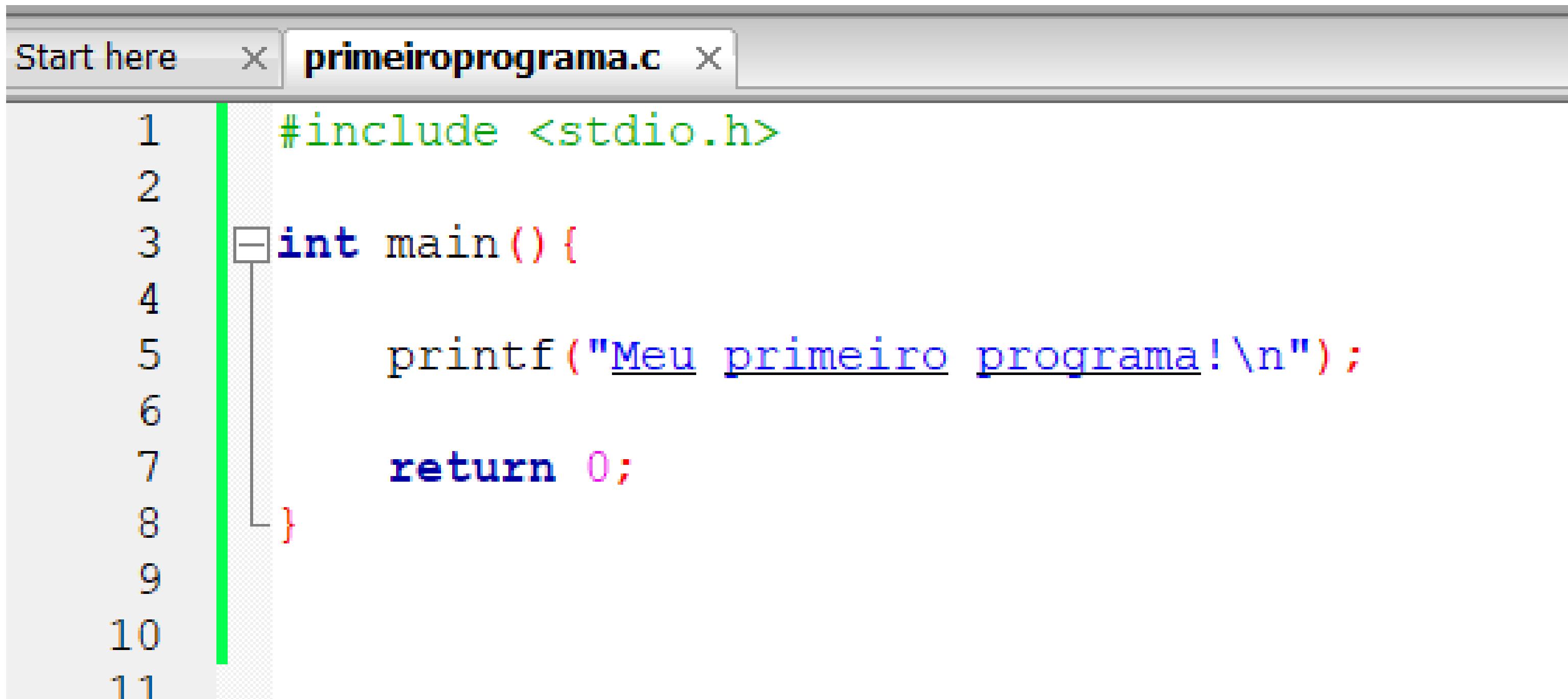
- 1. Faça login no seu computador**
- 2. Abra o Codeblocks**



Mãos à obra



Mãos à obra

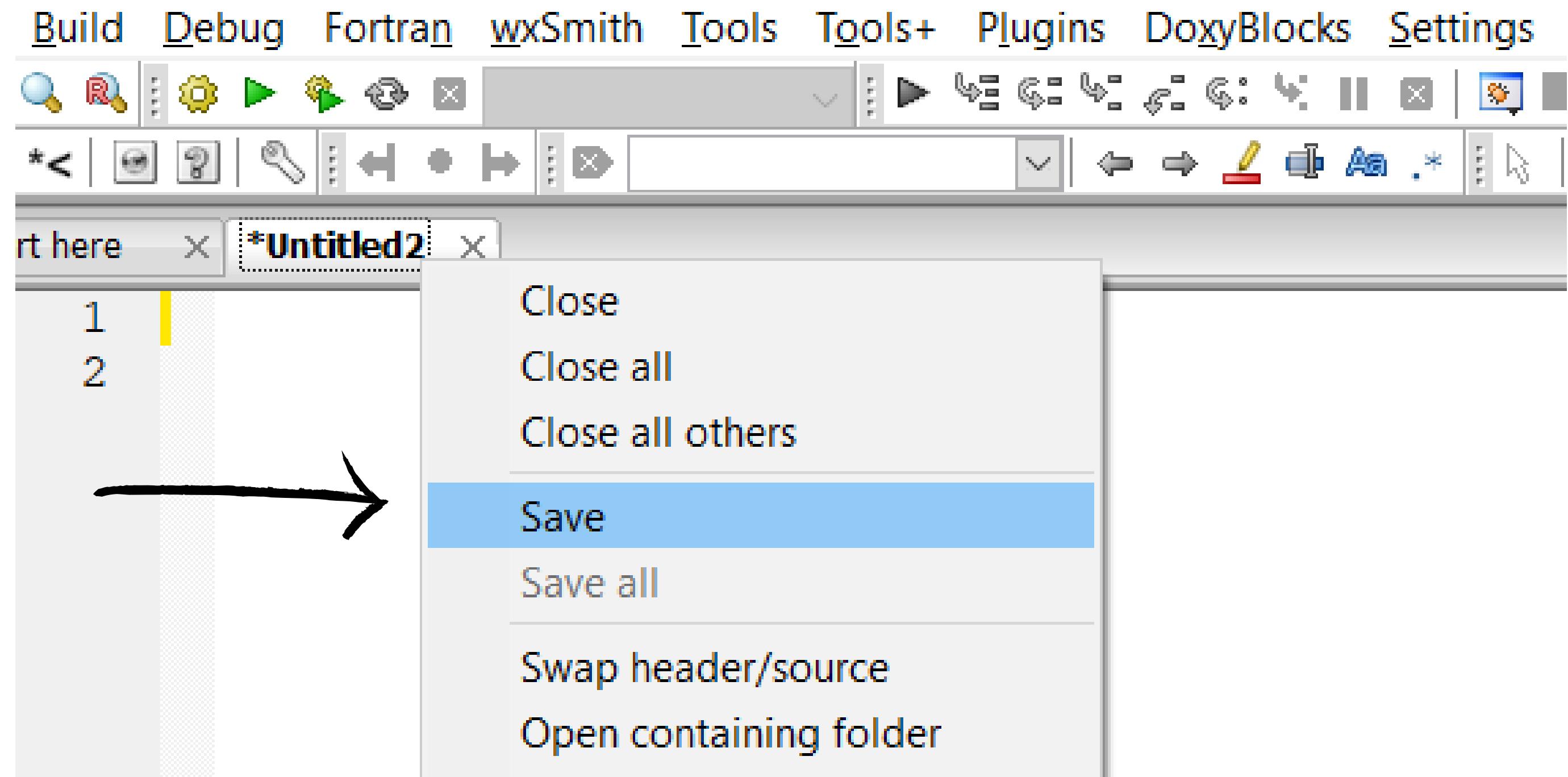


The screenshot shows a code editor window with the title bar "Start here" and "primeiroprograma.c". The code editor displays the following C program:

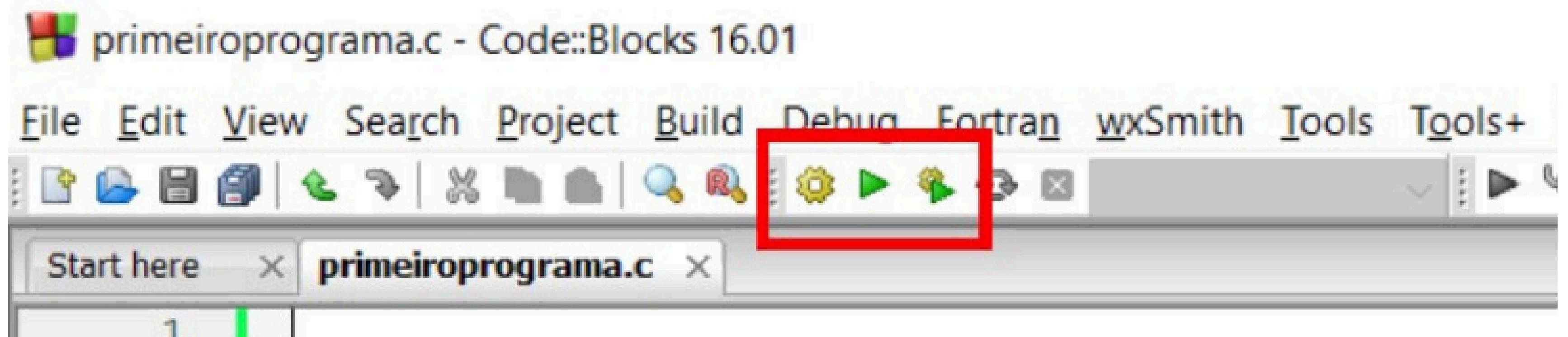
```
1 #include <stdio.h>
2
3 int main() {
4
5     printf("Meu primeiro programa!\n");
6
7     return 0;
8 }
```

The code is color-coded: green for the file extension ".c", blue for the include directive, red for the keyword "int", orange for "main()", blue for "printf", and purple for the string "Meu primeiro programa!\n". The line numbers are on the left, and a vertical green bar highlights the first line of the code.

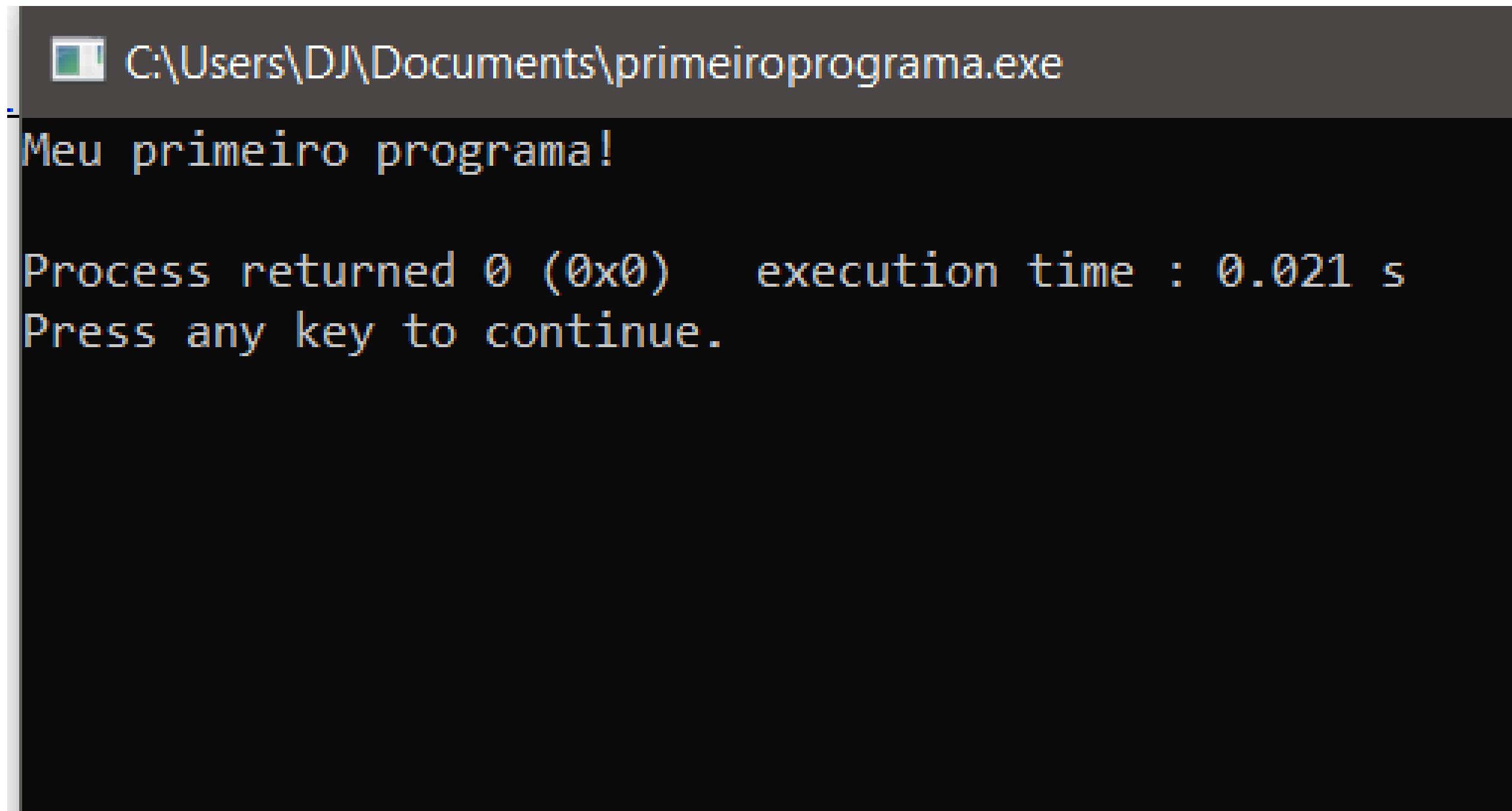
Mãos à obra



Mãos à obra



Mãos à obra



A screenshot of a terminal window with a dark background and light-colored text. The title bar shows the path "C:\Users\DJ\Documents\primeiroprograma.exe". The main area displays the following text:
Meu primeiro programa!

Process returned 0 (0x0) execution time : 0.021 s
Press any key to continue.

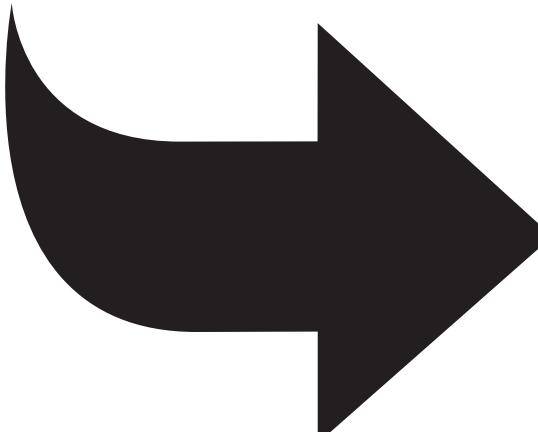
dia 1

Variáveis

O que é uma variável?

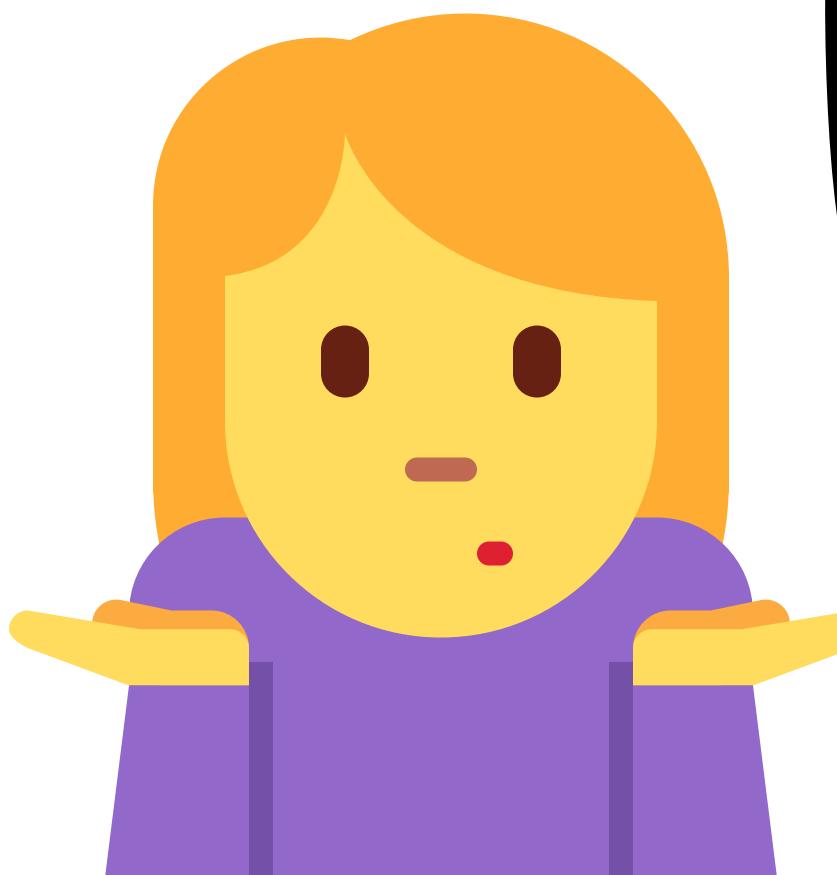


O que é uma
variável?



- é um lugar para
armazenar um valor
- esses valores
podem mudar

Declaração de variáveis



Mas como eu
posso criar
uma variável?

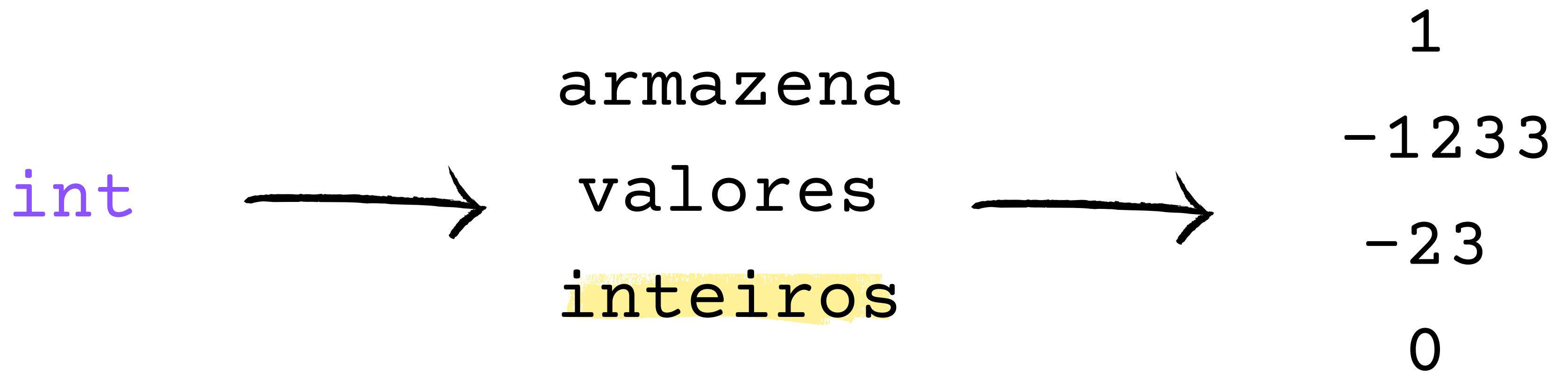
tipo da variável

int numero;

nome da
variável

sempre colocar
um ponto e
vírgula no final
da declaração

Tipos de Variáveis



```
int numero = 1234;
```



Tipos de Variáveis

float



armazena

valores

de ponto

flutuante

Em C, números
“com vírgula” são
representados
com “.”

1 . 5

0 . 432

124 . 34

```
float numero = 3.14;
```



Tipos de Variáveis

`double`



armazena

valores

de ponto

flutuante

(com maior precisão)

```
double numero = 3.14153;
```

A diferença entre float e double é a quantidade de valores que eles representam

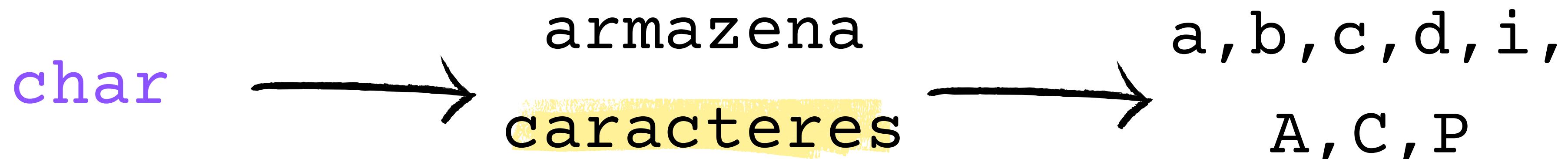
1.5

0.432

124.34



Tipos de Variáveis



```
char caractere = 'A';
```



Regras de nomeação

O nome de uma variável pode conter:

- letras
- números
- undercores (_)

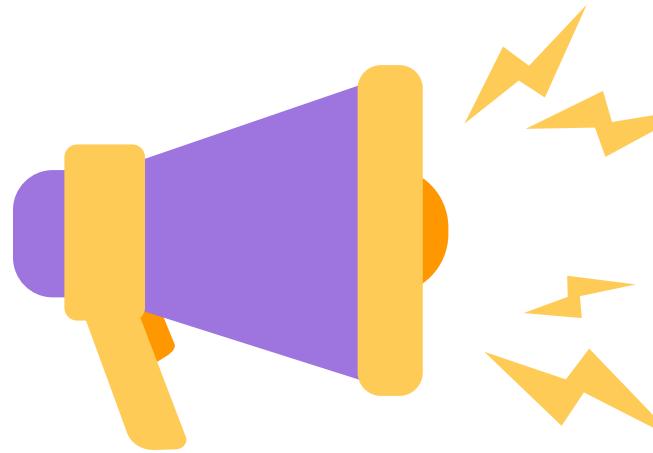


O nome de uma variável não pode:

- começar com números
- ter espaços em branco
- usar uma palavra reservada da linguagem



Regras de nomeação

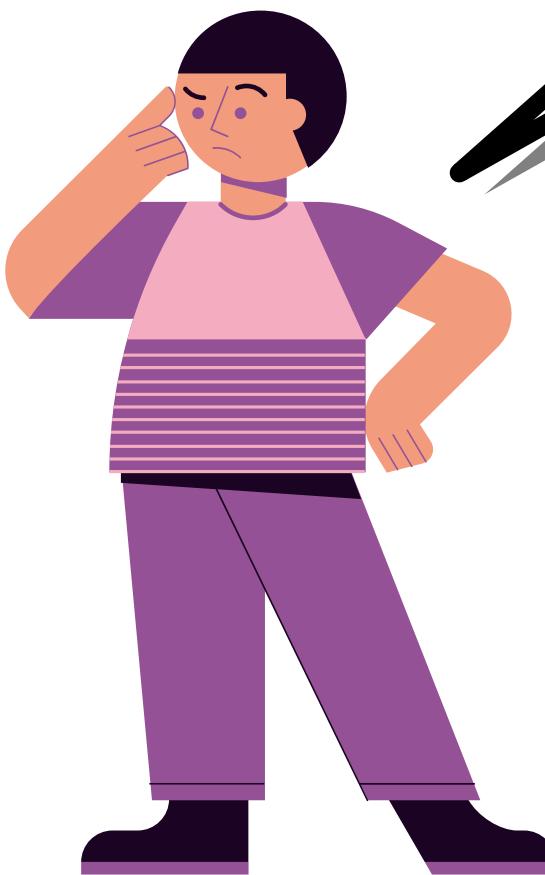


**Em C, há diferença entre letras
maiúsculas e minúsculas**

`int numero` **X** `int NUMERO` **X** `int Numero`

Atribuição de variáveis

**E como eu dou
um valor a
uma variável?**



Procure sempre atribuir valor à uma variável recém criada, mesmo que ele vá mudar. É uma boa prática!

O símbolo “=” é operador de atribuição, em que você armazena um valor na variável desejada

```
int numero = 1;
```



O “=” NÃO funciona como na matemática! Nós ainda vamos ver como é feita a comparação de igualdade

Escrevendo na tela



```
printf("Olá! \\n");
```



- **imprime algo na tela**
- **podemos imprimir variáveis também!**

Imprimindo variáveis

```
int numero = 10;  
  
printf("O valor do número é %d \n", numero);
```

A expressão “\n” indica para o programa que você quer pular para a próxima linha



Imprimindo variáveis



Quando queremos imprimir variáveis , precisamos formatar a saída de acordo com o tipo da variável

Imprimindo variáveis

```
float b = 2.5;
```

```
printf("%f\n" , b);
```

Você também pode formatar a quantidade de casas decimais que deseja imprimir

`printf("%.2f\n" , b);` indica que você quer imprimir somente até 2 casas decimais, por exemplo



Imprimindo variáveis

```
char c = 'w';
```

```
printf("%c\n" , c);
```



Imprimindo variáveis

```
double d = 3.1415326;
```

```
printf("%lf\n" , d);
```

A formatação de casas decimais também funciona aqui!

```
printf("%.5lf\n" , d); imprime até 5 casas decimais
```



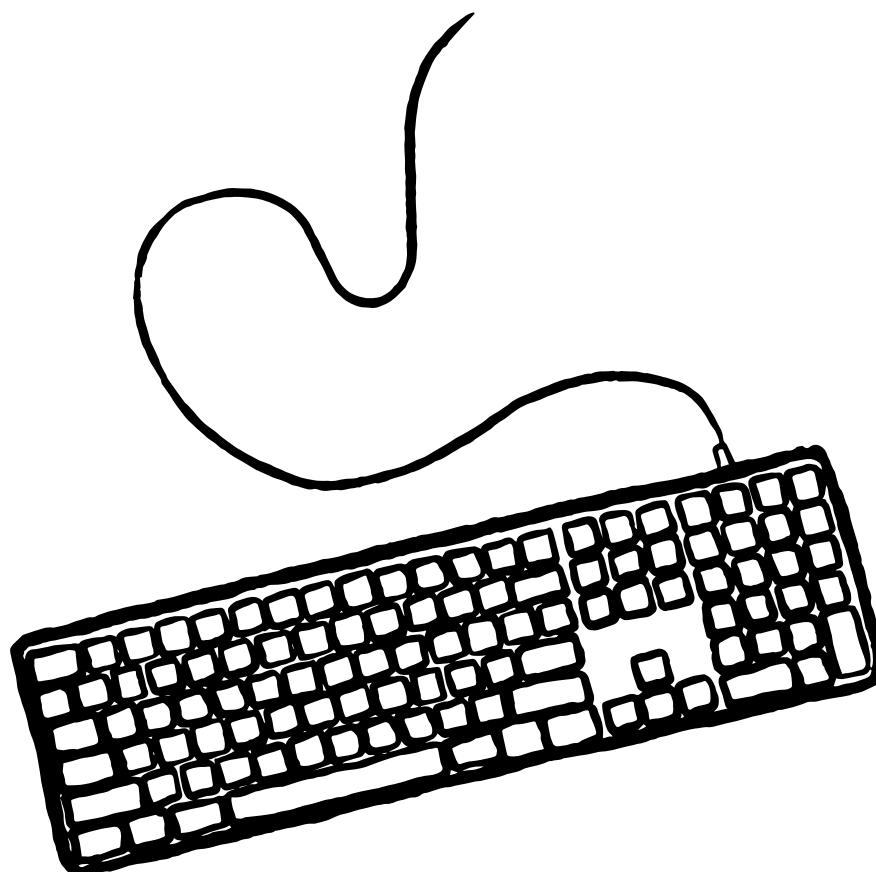
Imprimindo variáveis

Você pode imprimir diversas variáveis
em conjunto!

```
int a = 1;  
float b = 3.14;  
char c = 'A' ;  
double d = 3.1415326;  
  
printf("%d\n %f\n %c\n %lf\n", a, b, c, d);
```

Lendo Variáveis

E quando precisamos ler um valor que o usuário digita?



```
scanf("%d", &variável);
```

precisamos colocar um “&” antes do nome da variável quando estamos lendo ela

Lendo Variáveis



```
scanf("%d", &inteiro);
```

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

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

```
scanf("%lf", &duplaPrecisao);
```

Dia 1

Operadores

Operadores Aritméticos



**Operadores aritméticos são
utilizados para realizar
operações matemáticas**

Operadores Aritméticos

Operadores + e -

```
int a = 2 + 2;
```

```
int a = 7 - 3;
```



Operadores Aritméticos

Operador / e *

```
int a = 5 * 4;
```

```
int a = 20 / 5;
```



Operadores Aritméticos

Operador %

```
int a = 5 % 2;
```

Representa o resto de uma
divisão



Precedência de operadores

**Alguns operadores são
executados primeiro do
que outros**

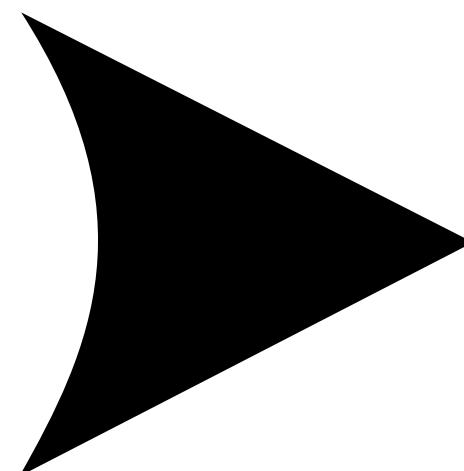


Ordem de precedência



- 1 . Parênteses
- 2 . *, /, %
- 3 . +, -

**quando tem a mesma
precedência, prevalece a ordem
da esquerda para a direita**

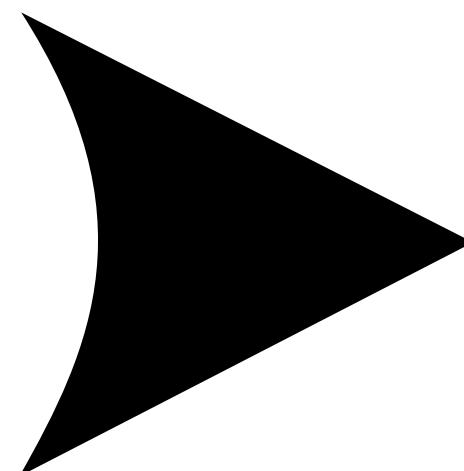


Ordem de precedência



- 1 . Parênteses
- 2 . *, /, %
- 3 . +, -

**quando tem a mesma
precedência, prevalece a ordem
da esquerda para a direita**



Operadores Relacionais



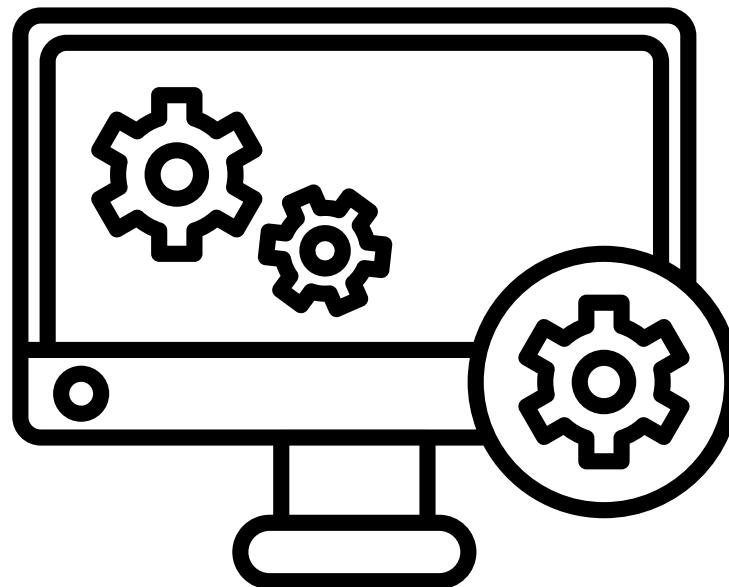
Operadores relacionais são usados para comparar dois valores (ou variáveis)

Operadores Relacionais

- > maior que
- >= maior ou igual
- < menor que
- <= menor ou igual
- == igual
- != diferente



Operadores Relacionais



**O resultado de uma
comparação sempre é 0
(falso) ou 1 (verdadeiro)**

Operadores Relacionais

```
int a = 10 > 5;
```

```
int b = 5 > 10;
```

```
int c = 2 == 2;
```

```
int d = 2 != 2;
```



Precedência

**Operadores aritméticos
têm precedência sobre os
operadores relacionais**

$3+5 < 6*2$ é o mesmo que $(3+5) < (6*2)$



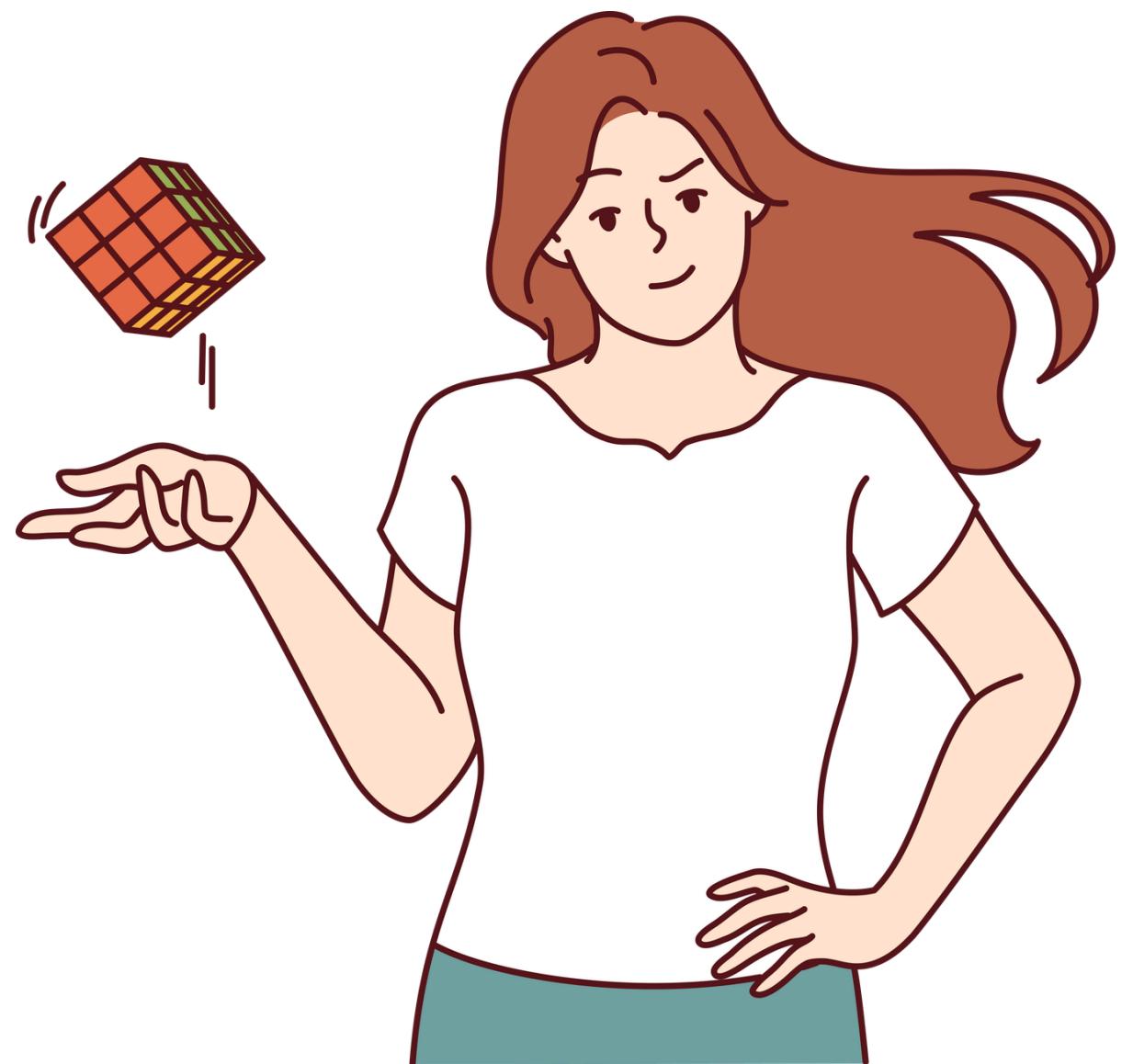
Operadores Relacionais

E se fizermos $3 + (5 < 6) * 2$?

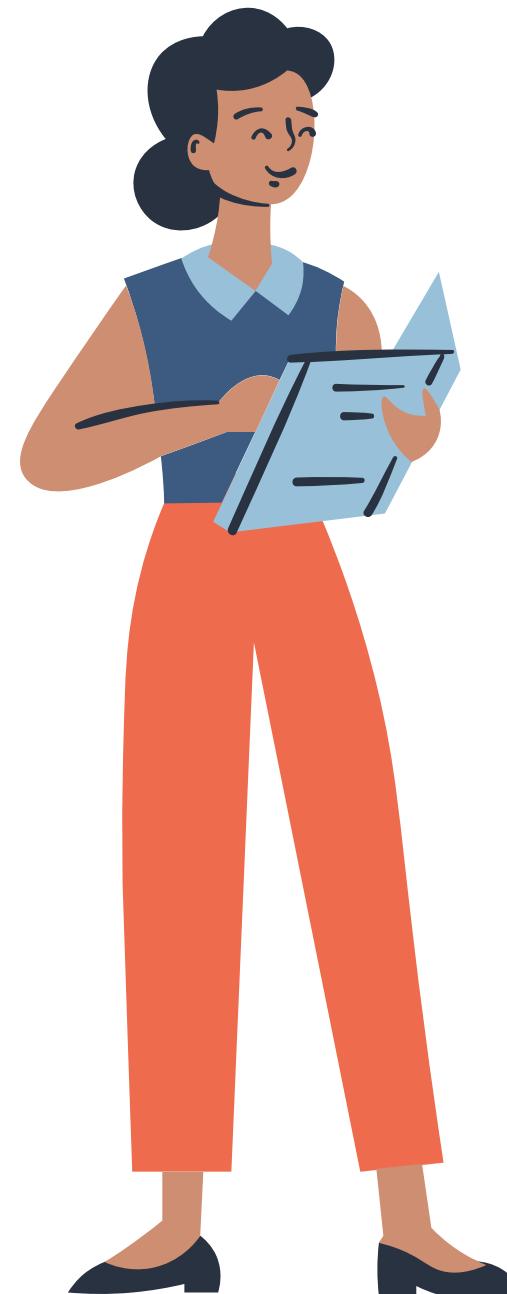


Operadores Lógicos

**Operadores lógicos servem
pra determinar a lógica
entre dois valores ou
variáveis**



Operadores Lógicos



Operador **&&** (and)

**As duas expressões
precisam ser verdadeiras
para o resultado ser 1
(verdadeiro)**

Operadores Lógicos

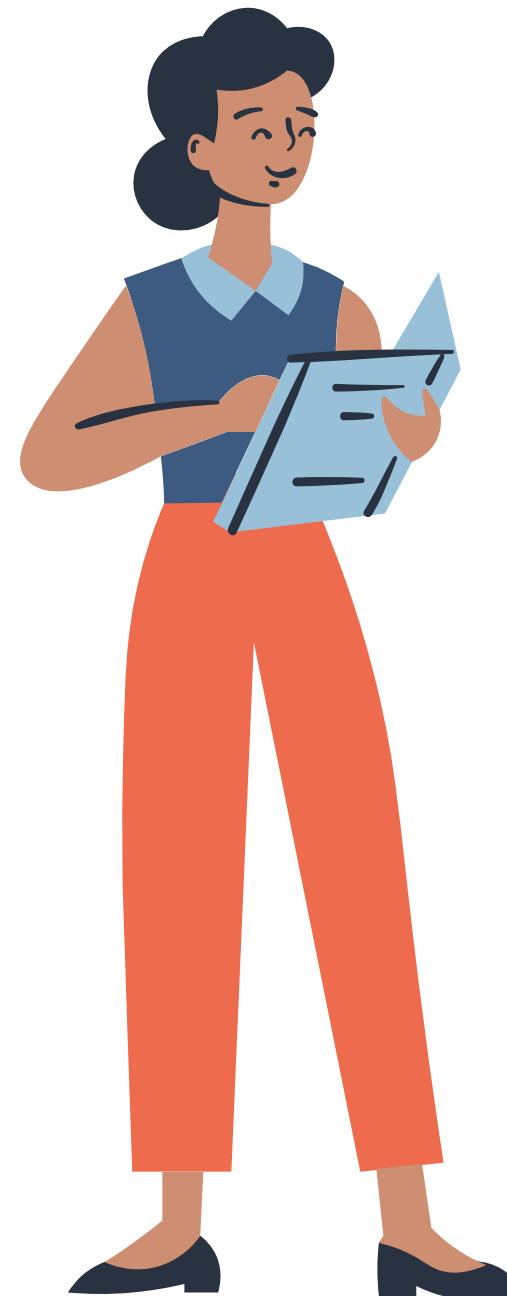
```
int a = (10 > 5) && (3 > 2);
```

```
int b = (5 > 10) && (3 > 2);
```

```
int c = (2 <= 3) && (5 >= 5);
```



Operadores Lógicos



Operador || (or)

**Pelo menos uma das duas
expressões precisa ser
verdadeira**

Operadores Lógicos

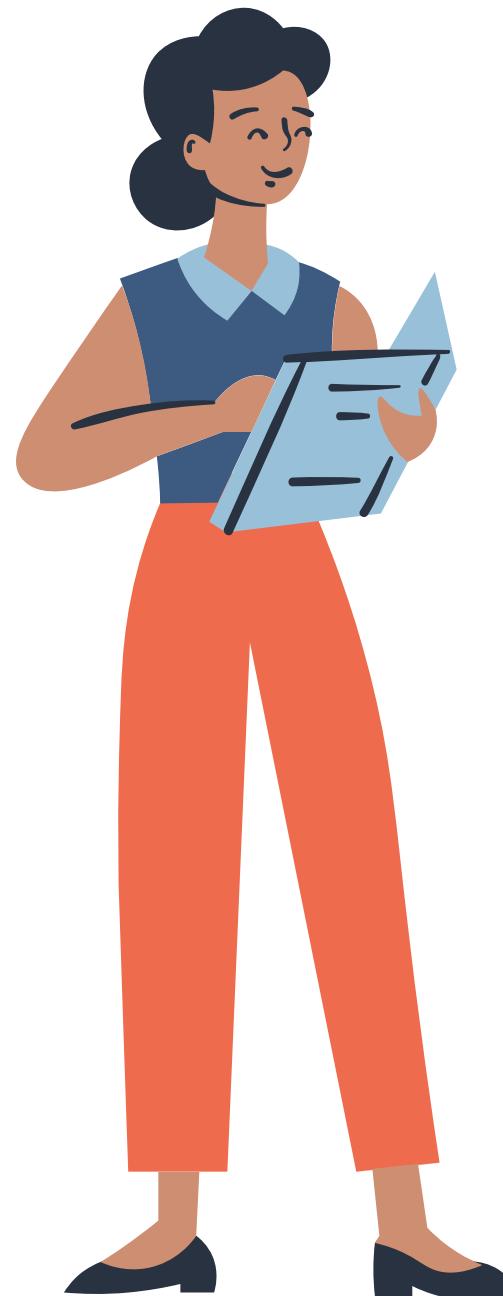
```
int a = (7 > 5) || (5 < 2);
```

```
int b = (5 > 10) || (3 > 2);
```

```
int c = (5 <= 2) || (2 >= 3);
```



Operadores Lógicos



Operador ! (not)

**Reverte o resultado da
expressão**

Operadores Lógicos

```
int a = !(7 > 5);
```

```
int b = !(10 < 2);
```

```
int c = !(8 >= 4);
```



Operadores Lógicos

IMPORTANTE

'=' é diferente de '=='

'=' serve para **atribuir valores**

'==' serve para **comparações**



dia 1

Boas práticas

O que são?



Boas práticas são um conjunto de convenções adotadas pelos programadores durante a escrita do código

Não são obrigatórias para o funcionamento do programa, mas ajudam muito na legibilidade e, se necessário, no conserto de erros!

Indentação de Códigos



Indentação é um **espaçamento antes
de cada linha**

Facilita a **leitura e **compreensão** do
código**

Melhora a **organização do código**

Indentação de Códigos

```
#include<stdio.h>

int main ( ) {
    int a;
    scanf ("%d", &a);
    while (a < 100)
        a++;
    printf ("%d", a);
    return 0;
}
```

X

```
#include<stdio.h>

int main ( ) {
    int a;
    scanf ("%d", &a);

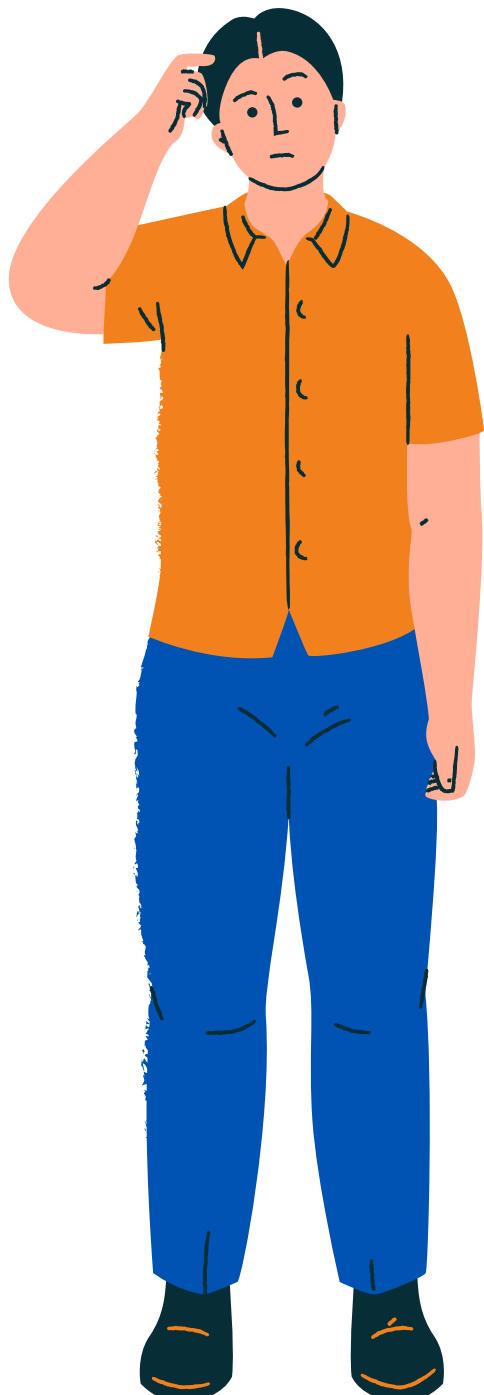
    while (a < 100)
        a++;

    printf ("%d", a);

    return 0;
}
```



Comentários



**Nem sempre as pessoas conseguem
entender o que o nosso código faz**

**Colocar comentários explicando o
que o código faz pode ajudar**

Comentários

Os comentários podem começar com //

```
#include <stdio.h>

int main()
{
    // O seguinte trecho mostra uma mensagem na tela
    // Essas linhas comentadas não irão aparecer na tela
    printf("Meu primeiro programa em C!\n");
}
```



Comentários

Ou podem começar com /* e terminar com */

```
#include <stdio.h>

int main()
{
    /* Os comandos abaixo imprimem 2 linhas
       no terminal. Os comentários escritos são ignorados
       pelo compilador */
    printf("Meu primeiro programa em C!\n");
    printf("Estou muito feliz com isso!\n");
}
```





PROGRAMAÇÃO 101

dia 2

- Comandos Condicionais
- Comandos de Repetição

dia 2

Comandos Condicionais

Contexto

Em diversos casos, queremos que nosso programa tenha comportamentos diferentes em situações diferentes

Em outras palavras, queremos que ele faça operações distintas a depender das condições em que é executado



Definição

Para esses casos, podemos utilizar os
Comandos Condicionais

O **bloco de código** de um comando condicional
será executado somente se uma determinada
condição for verdadeira



Definição

De forma geral:

```
Se (condição) {  
    executo esses  
    comandos  
}
```

} bloco de código



Condições

Você pode interpretar condições como perguntas que devem ser respondidas com **sim** ou **não**.

Mais formalmente, uma condição é uma expressão que resulte em uma resposta do tipo verdadeiro (**sim**) ou falso (**não**)



Condições

Pergunta	Resposta
Vocês vieram para o Programação 101 hoje?	SIM
Vocês ficaram em casa hoje?	NÃO
Vocês utilizaram o computador hoje?	SIM



Condições

Expressões condicionais podem ser construídas utilizando **variáveis** e **valores em conjunto** com os **operadores** que vimos anteriormente:

Aritméticos: +, -, *, /, %

Relacionais: >, <, >=, <=, ==, !=

Lógicos: &&, ||, !



Condições

Beleza, então como fica uma pergunta em escrita de código?

Pergunta	Condição	Resp. Natural	Resp. Comp.
5 é maior que 3?	$5 > 3$	SIM	VERDADEIRO
4 é menor que 2?	$4 < 2$	NÃO	FALSO
1 é maior que 5 ou 7 é menor que 8?	$(1 > 5) \text{ } (7 < 8)$	SIM	VERDADEIRO
2 e 3 são pares?	$(2 \% 2 == 0) \text{ \&\& } (3 \% 2 == 0)$	NÃO	FALSO



Condições

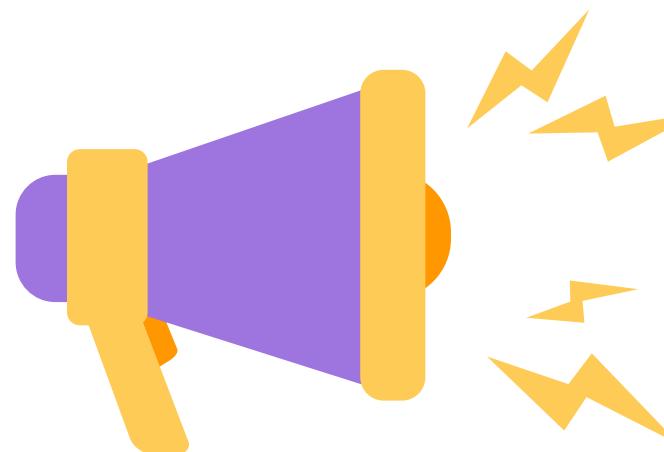
Na maioria dos casos, vamos criar condicionais que envolvem variáveis. Suponha uma variável “num” do tipo inteiro (int) que recebeu o valor 5:

Pergunta	Condição	Resp. Natural	Resp. Comp.
num é par?	<code>num % 2 == 0</code>	NÃO	FALSO
num é ímpar?	<code>num % 2 != 0</code>	SIM	VERDADEIRO
num é não-negativo?	<code>num >= 0</code>	SIM	VERDADEIRO



Condições

Quando usamos valores em uma condição, o único valor interpretado como **FALSO** é o 0.



Qualquer valor diferente de 0 é interpretado como **VERDADEIRO**

Condições

Condição	Resp. Comp.
1	VERDADEIRO
75 && 100	VERDADEIRO
0	FALSO
!0	VERDADEIRO



If

O primeiro comando condicional que veremos é o if. Ele possui a seguinte sintaxe:

```
6  if(condição) {  
7      // executo este  
8      // bloco de código  
9  }  
10 }  
11 }
```

If

Caso a condição seja **VERDADEIRA**, a sequência de comandos dentro do bloco de códigos será executada.

Caso seja **FALSA**, o programa irá ignorar os comandos existentes dentro do bloco do if e continuar sua execução normalmente



If

```
1 #include <stdio.h>
2
3 int main()
4 {
5
6     int condicao = 0; //condicao FALSA
7
8     if(condicao) {
9         printf("Estou no if!\n");
10    }
11
12    printf("Continuando meu programa");
13
14    return 0;
15 }
16
```

```
Continuando meu programa
```

```
...Program finished with exit code 0
Press ENTER to exit console.█
```

```
1 #include <stdio.h>
2
3 int main()
4 {
5
6     int condicao = 1; //condicao VERDADEIRA
7
8     if(condicao) {
9         printf("Estou no if!\n");
10    }
11
12    printf("Continuando meu programa");
13
14    return 0;
15 }
16
```

```
Estou no if!
Continuando meu programa
```

```
...Program finished with exit code 0
Press ENTER to exit console.█
```



Else if

Um comando complementar ao if é o else if, muito útil para quando você deseja testar mais de uma condição possível para a mesma variável

```
6  if(primeira condição) {  
7      |     executo este código  
8  }  
9  
10 else if(segunda condição) {  
11     |     executo este outro código  
12 }  
13  
14 else if(terceira condição) {  
15     |     executo este terceiro código  
16 }
```



Else if

Nesse caso, o programa irá sequencialmente verificar as condições e, caso alguma delas seja **VERDADEIRA**, executar o respectivo bloco de código.

Se todas forem **FALSAS**, ele apenas seguirá normalmente a execução.



Else if

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int meuNumero = 0;
6
7     if(meuNumero == 0) {
8         printf("Eh zero!\n");
9     }
10
11    else if(meuNumero == 1) {
12        printf("Eh um!\n");
13    }
14
15    else if(meuNumero == 2) {
16        printf("Eh dois!\n");
17    }
18
19    printf("Continuando meu programa");
20
21    return 0;
22 }
```

```
Eh zero!
Continuando meu programa
...Program finished with exit code 0
Press ENTER to exit console.
```

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int meuNumero = 1;
6
7     if(meuNumero == 0) {
8         printf("Eh zero!\n");
9     }
10
11    else if(meuNumero == 1) {
12        printf("Eh um!\n");
13    }
14
15    else if(meuNumero == 2) {
16        printf("Eh dois!\n");
17    }
18
19    printf("Continuando meu programa");
20
21    return 0;
22 }
```

```
Eh um!
Continuando meu programa
...Program finished with exit code 0
Press ENTER to exit console.[]
```

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int meuNumero = 2;
6
7     if(meuNumero == 0) {
8         printf("Eh zero!\n");
9     }
10
11    else if(meuNumero == 1) {
12        printf("Eh um!\n");
13    }
14
15    else if(meuNumero == 2) {
16        printf("Eh dois!\n");
17    }
18
19    printf("Continuando meu programa");
20
21    return 0;
22 }
```

```
Eh dois!
Continuando meu programa
...Program finished with exit code 0
Press ENTER to exit console.[]
```



Else

Finalmente, temos o **else**, que ao contrário dos outros dois comandos não avalia condição. Ele será executado quando todas as condições anteriores forem falsas.

```
6      if(primeira condicao) {  
7          | executo este código  
8      }  
9  
10     else if(segunda condicao) {  
11         | executo este outro código  
12     }  
13  
14     else if(terceira condicao) {  
15         | executo este terceiro código  
16     }  
17  
18     else {  
19         | se nenhuma condição foi verdadeira  
20         | até agora, executo este último  
21         | código  
22 }
```



Else

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int meuNumero = 4;
6
7     if(meuNumero % 2 == 0) {
8         printf("Divide por 2!\n");
9     }
10
11    else if(meuNumero % 3 == 0) {
12        printf("Divide por 3!\n");
13    }
14
15    else if(meuNumero % 5 == 0) {
16        printf("Divide por 5!\n");
17    }
18
19    else {
20        printf("Nao divide por nenhum.");
21    }
22
23    return 0;
24 }
```

```
Divide por 2!
...
Program finished with exit code 0
Press ENTER to exit console.[]
```

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int meuNumero = 25;
6
7     if(meuNumero % 2 == 0) {
8         printf("Divide por 2!\n");
9     }
10
11    else if(meuNumero % 3 == 0) {
12        printf("Divide por 3!\n");
13    }
14
15    else if(meuNumero % 5 == 0) {
16        printf("Divide por 5!\n");
17    }
18
19    else {
20        printf("Nao divide por nenhum.");
21    }
22
23    return 0;
24 }
```

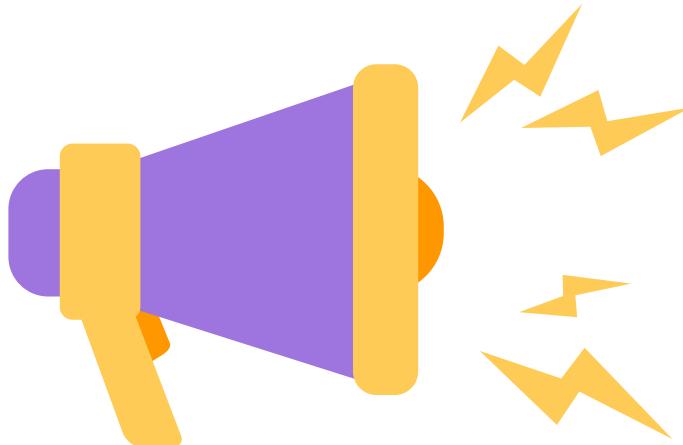
```
Divide por 5!
...
Program finished with exit code 0
Press ENTER to exit console.[]
```

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int meuNumero = 7;
6
7     if(meuNumero % 2 == 0) {
8         printf("Divide por 2!\n");
9     }
10
11    else if(meuNumero % 3 == 0) {
12        printf("Divide por 3!\n");
13    }
14
15    else if(meuNumero % 5 == 0) {
16        printf("Divide por 5!\n");
17    }
18
19    else {
20        printf("Nao divide por nenhum.");
21    }
22
23    return 0;
24 }
```

```
Nao divide por nenhum.
...
Program finished with exit code 0
Press ENTER to exit console.[]
```



Comandos Condicionais



Em um conjunto de if,
else if's e else, quando
uma das condicionais
é satisfeita todas as
subsequentes
serão ignoradas!

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int meuNumero = 30;
6
7     if(meuNumero % 2 == 0) {
8         printf("Divide por 2!\n");
9     }
10
11    else if(meuNumero % 3 == 0) {
12        printf("Divide por 3!\n");
13    }
14
15    else if(meuNumero % 5 == 0) {
16        printf("Divide por 5!\n");
17    }
18
19    else {
20        printf("Nao divide por nenhum.");
21    }
22
23
24    return 0;
25 }
```

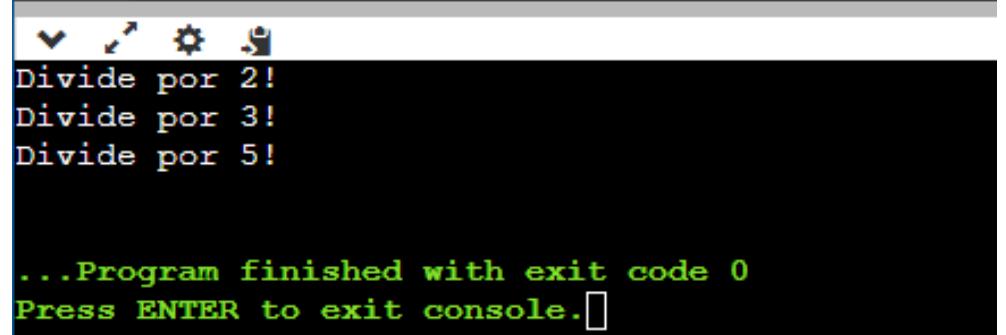
```
Divide por 2!

...Program finished with exit code 0
Press ENTER to exit console.
```

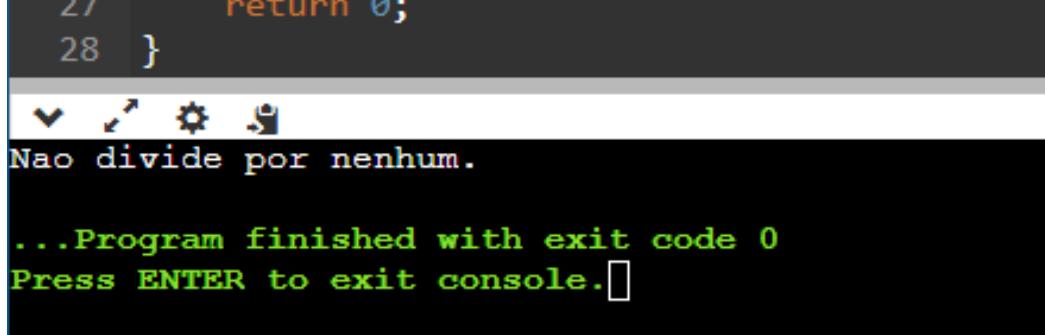
Comandos Condicionais

Nos casos em que várias condições podem ser satisfeitas ao mesmo tempo, a solução é avaliar cada uma em um if

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int meuNumero = 30;
6     int naoDividiu = 1;
7
8     if(meuNumero % 2 == 0) {
9         naoDividiu = 0;
10        printf("Divide por 2!\n");
11    }
12
13    if(meuNumero % 3 == 0) {
14        naoDividiu = 0;
15        printf("Divide por 3!\n");
16    }
17
18    if(meuNumero % 5 == 0) {
19        naoDividiu = 0;
20        printf("Divide por 5!\n");
21    }
22
23    if(naoDividiu == 1) {
24        printf("Nao divide por nenhum.");
25    }
26
27    return 0;
28 }
```



```
1 #include <stdio.h>
2
3 int main()
4 {
5     int meuNumero = 7;
6     int naoDividiu = 1;
7
8     if(meuNumero % 2 == 0) {
9         naoDividiu = 0;
10        printf("Divide por 2!\n");
11    }
12
13    if(meuNumero % 3 == 0) {
14        naoDividiu = 0;
15        printf("Divide por 3!\n");
16    }
17
18    if(meuNumero % 5 == 0) {
19        naoDividiu = 0;
20        printf("Divide por 5!\n");
21    }
22
23    if(naoDividiu == 1) {
24        printf("Nao divide por nenhum.");
25    }
26
27    return 0;
28 }
```



Comandos Condicionais

Todo conjunto de condicionais DEVE TER um único if, PODE TER quantos else if's for desejado e PODE TER um único else.



Caso não seja especificado a qual if um else if/else pertence, ele pertencerá ao if imediatamente anterior

Switch

Outro comando condicional que veremos é o **switch**. Ele avalia se uma variável possui valor equivalente a algum valor constante

```
6 switch(variável) {  
7     case primeiro_valor:  
8         executo este código  
9         break;  
10    case segundo_valor:  
11        executo este  
12        outro código  
13        break;  
14    case terceiro_valor:  
15        executo este  
16        terceiro código  
17        break;  
18 }
```

Falaremos sobre o “break” em breve!



Switch

```
1 #include <stdio.h>
2
3 int main()
4 {
5
6     int a = 7;
7     int b = 3;
8     int opcao = 1;
9
10    switch(opcao){
11        case 1:
12            printf("A soma dos numeros eh %d\n", a+b);
13            break;
14
15        case 2:
16            printf("A subtração dos numeros eh %d\n", a-b);
17            break;
18    }
19
20    return 0;
21 }
22
```

```
A soma dos numeros eh 10

...Program finished with exit code 0
Press ENTER to exit console.[]
```

```
1 #include <stdio.h>
2
3 int main()
4 {
5
6     int a = 7;
7     int b = 3;
8     int opcao = 2;
9
10    switch(opcao){
11        case 1:
12            printf("A soma dos numeros eh %d\n", a+b);
13            break;
14
15        case 2:
16            printf("A subtração dos numeros eh %d\n", a-b);
17            break;
18    }
19
20    return 0;
21 }
22
```

```
A subtração dos numeros eh 4

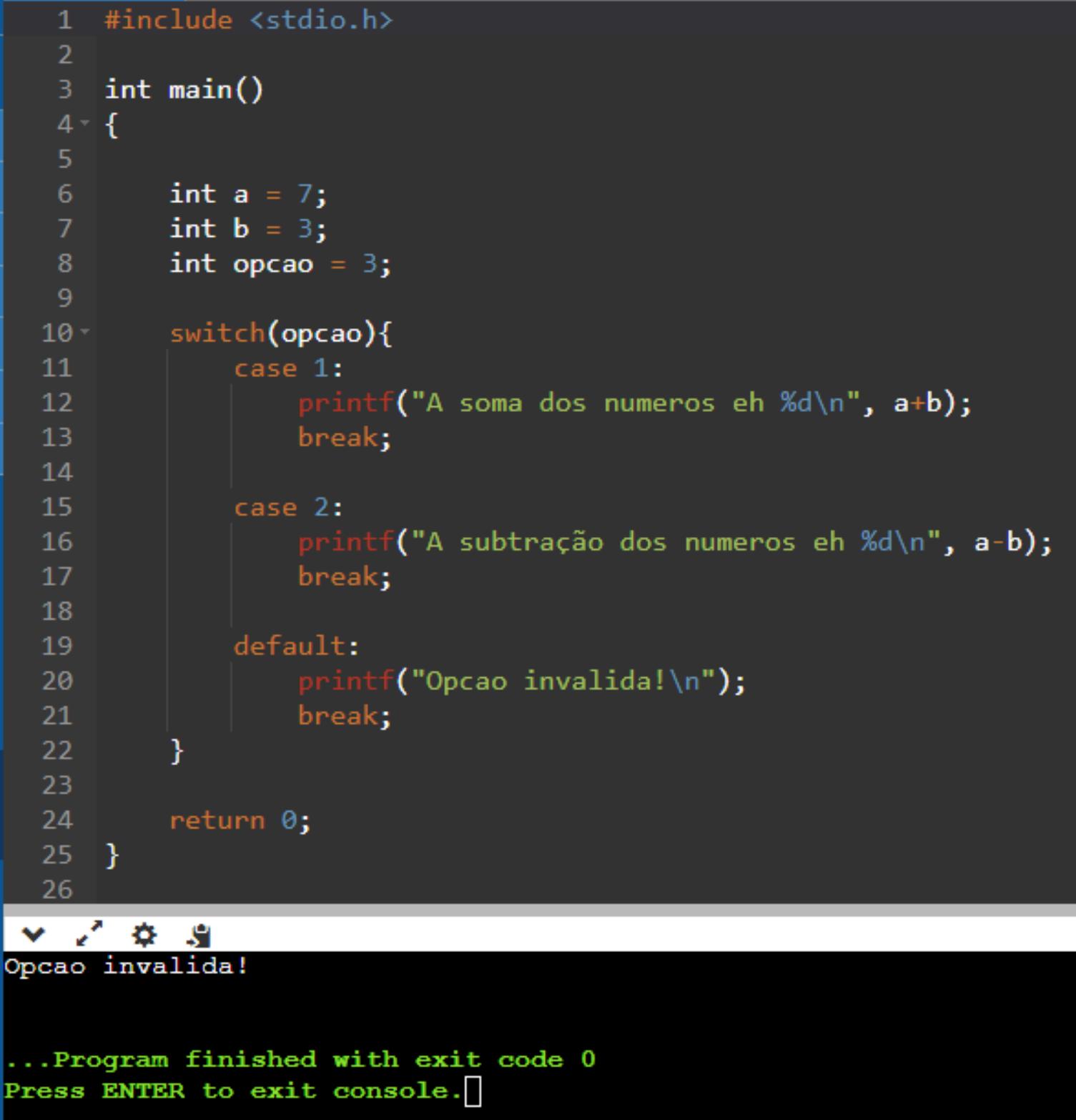
...Program finished with exit code 0
Press ENTER to exit console.[]
```



Switch

Opcionalmente, você pode adicionar um caso **default**, que será executado apenas se a variável comparada não possuir nenhum dos valores verificados no switch

```
1 #include <stdio.h>
2
3 int main()
4 {
5
6     int a = 7;
7     int b = 3;
8     int opcao = 3;
9
10    switch(opcao){
11        case 1:
12            printf("A soma dos numeros eh %d\n", a+b);
13            break;
14
15        case 2:
16            printf("A subtração dos numeros eh %d\n", a-b);
17            break;
18
19        default:
20            printf("Opcao invalida!\n");
21            break;
22    }
23
24    return 0;
25 }
26
```



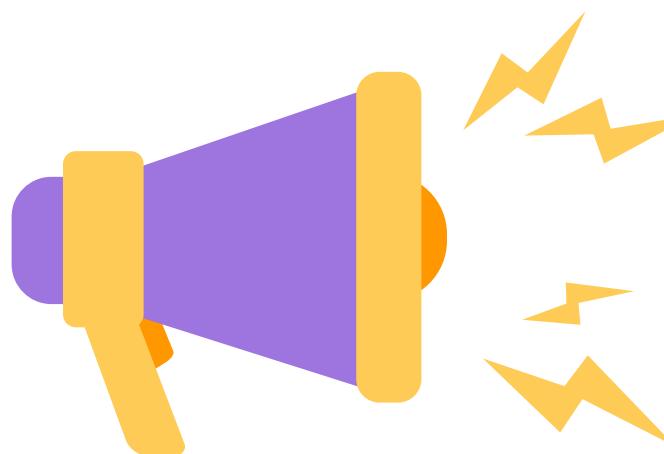
```
Opcao invalida!
```

```
...Program finished with exit code 0
Press ENTER to exit console.
```



Switch

O comando **break** que apareceu nas últimas imagens serve para **interromper a execução de um bloco**



Precisamos dele pois, na verdade, o switch funciona com efeito cascata, em que se um dos cases for satisfeito, todos os seguintes são executados

Switch

```
1 #include <stdio.h>
2
3 int main()
4 {
5
6     int salario = 0;
7     int cargo = 1;
8
9     switch(cargo) {
10         case 5:
11             salario += 2000;
12
13         case 4:
14             salario += 1000;
15
16         case 3:
17             salario += 500;
18
19         case 2:
20             salario += 250;
21
22         case 1:
23             salario += 125;
24
25     default:
26         salario += 100;
27     }
28
29     printf("De acordo com o cargo, seu salario eh %d\n", salario);
30
31     return 0;
32 }
```

```
De acordo com o cargo, seu salario eh 225
...Program finished with exit code 0
Press ENTER to exit console.[]
```

```
1 #include <stdio.h>
2
3 int main()
4 {
5
6     int salario = 0;
7     int cargo = 3;
8
9     switch(cargo) {
10         case 5:
11             salario += 2000;
12
13         case 4:
14             salario += 1000;
15
16         case 3:
17             salario += 500;
18
19         case 2:
20             salario += 250;
21
22         case 1:
23             salario += 125;
24
25     default:
26         salario += 100;
27     }
28
29     printf("De acordo com o cargo, seu salario eh %d\n", salario);
30
31     return 0;
32 }
```

```
De acordo com o cargo, seu salario eh 975
...Program finished with exit code 0
Press ENTER to exit console.
```

```
1 #include <stdio.h>
2
3 int main()
4 {
5
6     int salario = 0;
7     int cargo = 5;
8
9     switch(cargo) {
10         case 5:
11             salario += 2000;
12
13         case 4:
14             salario += 1000;
15
16         case 3:
17             salario += 500;
18
19         case 2:
20             salario += 250;
21
22         case 1:
23             salario += 125;
24
25     default:
26         salario += 100;
27     }
28
29     printf("De acordo com o cargo, seu salario eh %d\n", salario);
30
31     return 0;
32 }
```

```
De acordo com o cargo, seu salario eh 3975
...Program finished with exit code 0
Press ENTER to exit console.
```

Ao utilizar dessa forma, o caso default é sempre executado!



dia 2

Comandos de Repetição

Contexto

Suponha que você queira imprimir uma mensagem na tela 50 vezes, ou ler 100 números da entrada de dados do usuário.

Escrever o mesmo comando manualmente em diversas linhas é uma solução pouco prática, certo?



Definição

Nessas situações, fazemos uso dos
Comandos de Repetição

O **bloco de código** será repetidamente
executado enquanto uma determinada
condição for verdadeira, o que pode ocorrer
por diversas vezes

Definição

De forma geral:



While

O **while** é o primeiro comando de repetição que veremos e possui a seguinte sintaxe:

```
6  while(condição) {  
7  
8      executo este  
9      bloco de código  
10  
11 }
```



While

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int contador = 0;
6
7     while(contador < 3) {
8         printf("O contador esta em: %d\n", contador);
9         contador++;
10    }
11
12    return 0;
13 }
14
```

```
0 contador esta em: 0
0 contador esta em: 1
0 contador esta em: 2

...Program finished with exit code 0
Press ENTER to exit console.[]
```

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int contador = 0;
6
7     while(contador < 5) {
8         printf("O contador esta em: %d\n", contador);
9         contador++;
10    }
11
12    return 0;
13 }
14
```

```
0 contador esta em: 0
0 contador esta em: 1
0 contador esta em: 2
0 contador esta em: 3
0 contador esta em: 4

...Program finished with exit code 0
Press ENTER to exit console.[]
```



While

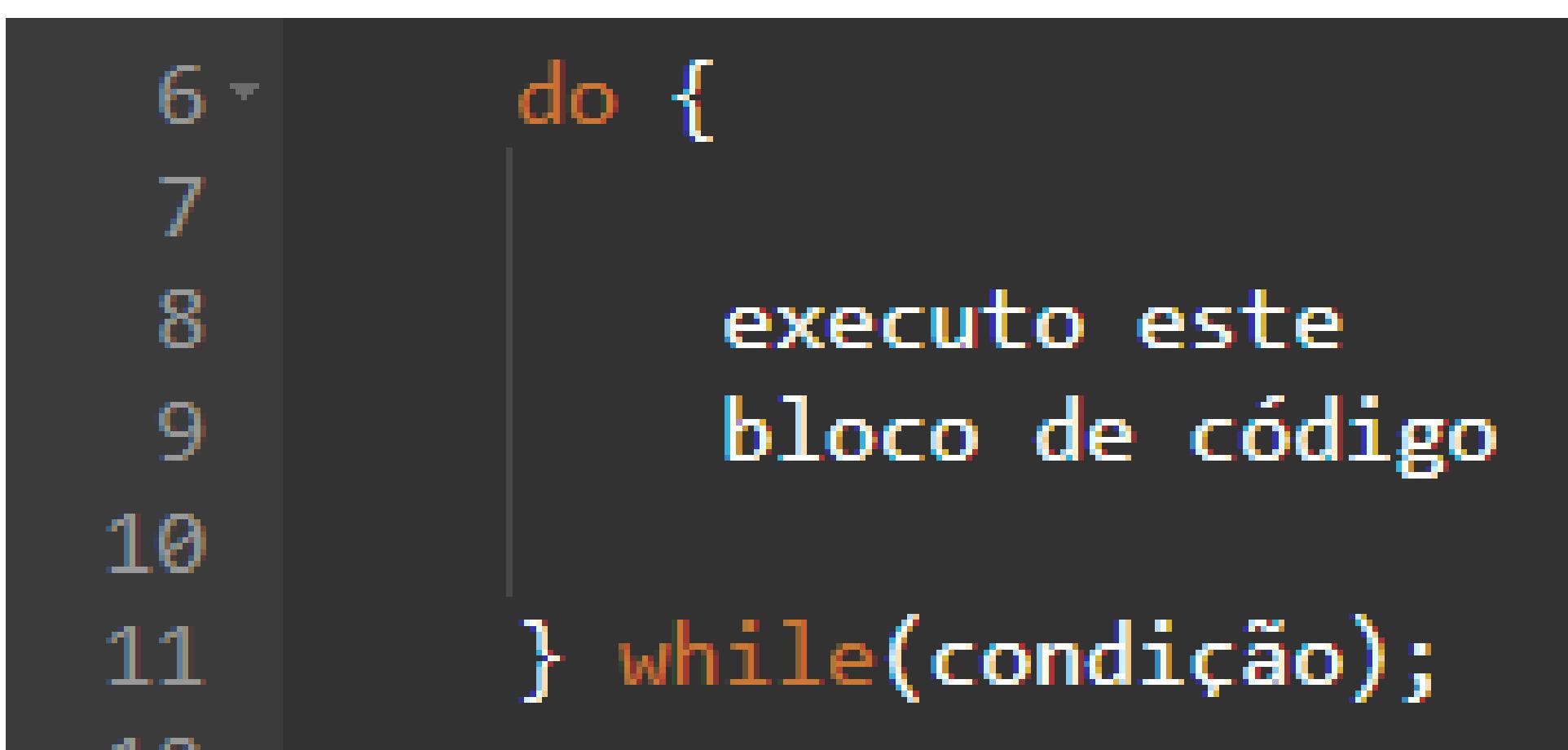
Primeiro, o while verifica se a condição é **VERDADEIRA**. Se for, executa o bloco de código e continua o processo de verificação e execução repetidamente.

Sua execução só será interrompida quando a condição se tornar **FALSA**.



Do-while

Muito semelhante ao comando anterior, mas com a principal diferença de que o **do-while** faz a validação da condição somente DEPOIS de executar pela primeira vez



A screenshot of a code editor showing a do-while loop. The code is:

```
6 do {  
7     // execute este  
8     // bloco de código  
9 } while(condição);  
10  
11
```

The numbers 6 through 11 are on the left, likely indicating line numbers. The code itself is on the right.



Do-while

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int contador = 0;
6
7     do {
8
9         printf("O contador esta em: %d\n", contador);
10        contador++;
11
12    } while(contador < 5);
13
14    return 0;
15 }
```

```
▼ ↻ ⚙ ⌂
O contador esta em: 0
O contador esta em: 1
O contador esta em: 2
O contador esta em: 3
O contador esta em: 4

...Program finished with exit code 0
Press ENTER to exit console.□
```

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int contador = 0;
6
7     do {
8
9         printf("O contador esta em: %d\n", contador);
10        contador++;
11
12    } while(contador > 1);
13
14    return 0;
15 }
```

```
▼ ↻ ⚙ ⌂
O contador esta em: 0

...Program finished with exit code 0
Press ENTER to exit console.□
```



Do-while

Primeiro, o do-while executa o bloco de código, e só então ele verifica se a condição é **VERDADEIRA**. Se for, ele continua o processo de execução e verificação

Novamente, sua execução só será interrompida quando a condição se tornar **FALSA**.



For

O **for** possui sintaxe um pouco distinta dos comandos vistos até agora, mas seu funcionamento segue a mesma ideia

```
6  for(inicializador; condição; incremento) {  
7  
8      |     executo este  
9      |     bloco de código  
10  
11 }
```



For

Como visto até agora, nos comandos de repetição temos uma variável de controle para garantir que, eventualmente, a condição seja falsa e o laço seja interrompido. Esse é o papel do **inicializador** dentro do **for**.

Já o **incremento** define qual mudança será feita nessa variável de controle a cada iteração.



For

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int contador = 0;
6
7     for(contador = 0; contador < 5; contador++) {
8
9         printf("O contador esta em: %d\n", contador);
10    }
11
12    return 0;
13 }
14
15
```

```
↙ ↘ ⚙ ⚙
O contador esta em: 0
O contador esta em: 1
O contador esta em: 2
O contador esta em: 3
O contador esta em: 4

...Program finished with exit code 0
Press ENTER to exit console.[]
```

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int contador = 0;
6
7     for(contador = 1; contador < 50; contador *= 2) {
8
9         printf("O contador esta em: %d\n", contador);
10    }
11
12    return 0;
13 }
14
15
```

```
↙ ↘ ⚙ ⚙
O contador esta em: 1
O contador esta em: 2
O contador esta em: 4
O contador esta em: 8
O contador esta em: 16
O contador esta em: 32

...Program finished with exit code 0
Press ENTER to exit console.[]
```



For

Primeiro, o for realiza a atribuição do inicializador e depois verifica se a condição é **VERDADEIRA**, executando o bloco de código em caso positivo.

Ele então incrementa o inicializador e continua o processo de verificação e execução, até que a condição se torne **FALSA**.



Comandos de Repetição

Em alguns casos, queremos que a repetição seja interrompida caso determinada condição seja obedecida.

Pode ser também que a quantidade de iterações seja incerta e não sabemos determinar um limite para a variável de controle.



Comandos de Repetição

Nessas ocasiões, o comando **break** é muito útil. Ao ser executado, ele irá imediatamente interromper a repetição.

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int contador = 1;
6
7     while(1) {
8
9         contador *= 2;
10
11        if(contador > 100) {
12            break;
13        }
14    }
15
16    printf("O contador esta em: %d\n", contador);
17
18    return 0;
19 }
```

O contador esta em: 128

...Program finished with exit code 0
Press ENTER to exit console.█





PROGRAMAÇÃO 101

dia 3

- Aula Prática
- Exercícios



PROGRAMAÇÃO 101

Dia 4

- Vetores
- Matrizes

dia 4

Vetores

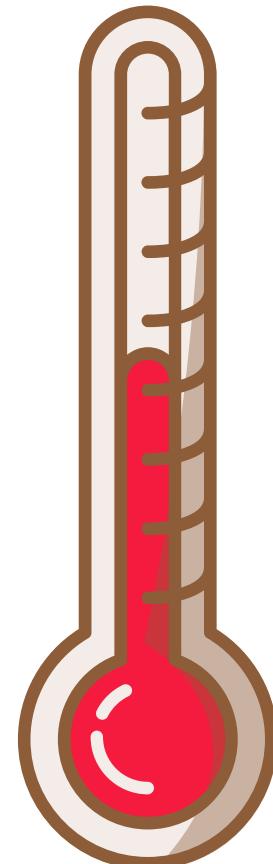
Motivação

Até agora, vimos variáveis que são capazes de armazenar um único valor por vez

Se precisarmos de muitos valores ao mesmo tempo, vamos precisar de várias variáveis



Motivação



Pense no seguinte problema:

Você precisa escrever um programa que leia as temperaturas dos últimos 5 dias e determine qual é a temperatura média nesse intervalo de tempo

```
#include <stdio.h>

int main() {
    float t1, t2, t3, t4, t5;
    printf("Digite as temperaturas dos ultimos 5 dias: ");

    scanf("%f", &t1);
    scanf("%f", &t2);
    scanf("%f", &t3);
    scanf("%f", &t4);
    scanf("%f", &t5);

    float temperaturaMedia = (t1 + t2 + t3 + t4 + t5) / 5;
    printf("A temperatura media eh %f\n", temperaturaMedia);

    return 0;
}
```



Motivação

Qual é o problema da solução do slide anterior?



Motivação

Qual é o problema da solução do slide anterior?

Imagine se quiséssemos obter a média de um ano. Precisaríamos de 365 variáveis!



Motivação

Felizmente, existe um recurso em C que permite associar múltiplos valores a um mesmo nome/identificador, ou seja, a uma mesma variável



Vetores

Vetores (também chamados de **arranjos** ou **arrays**) são um tipo de estruturas de dados utilizado para armazenar coleções de elementos de um mesmo tipo



Vetores

Um vetor é associado a um nome, da mesma forma que uma variável, e cada elemento dentro de um vetor pode ser acessado por meio de índices



Declarando um vetor

Forma mais simples:

```
tipo_do_vetor nome_do_vetor[tamanho];
```

Exemplos:

```
float temperaturas[365];
```

```
int notas[1000];
```

```
char nome[6];
```



Tamanho

O tamanho de um vetor deve ser um valor inteiro, constante e positivo!



1
10000
365



0
-1
2.5

Declarando um vetor

Inicializando os valores na declaração:

```
tipo_do_vetor nome_do_vetor[tamanho] = {dados};
```

Exemplos:

```
float temperaturas[3] = {29.0, 30.5, 26.7};
```

```
int notas[5] = {10, 10, 9, 5, 6};
```

```
char nome[6] = {'P', '1', '0', '1'};
```



Declarando um vetor

Podemos omitir o tamanho em declarações assim:

```
tipo_do_vetor nome_do_vetor[] = {dados};
```

Exemplos:

```
float temperaturas[] = {29.0, 30.5, 26.7};
```

```
int notas[] = {10, 10, 9, 5, 6};
```

```
char nome[] = {'P', '1', '0', '1'};
```



Acessando elementos

Vetores armazenam diversos valores simultaneamente. Para acessar e editar valores individualmente, podemos utilizar índices:

```
nome_do_vetor[indice];
```



Acessando elementos

Exemplo:

```
float temperaturas[365];
temperaturas[0] = 20.0;

// Imprime: 20.0
printf("%f\n", temperaturas[0]);
```



IMPORTANTE

Assim como na grande maioria das linguagens de programação, os índices em C se iniciam em 0 ao invés de 1.

De forma mais geral, devemos usar o índice $N-1$ para acessar o elemento na posição N .



Operações com vetores

Para realizar operações utilizando todos os elementos de um vetor, precisamos realizar um mesmo comando repetidas vezes. Como podemos lidar com isso?



Operações com vetores

Para realizar operações utilizando todos os elementos de um vetor, precisamos realizar um mesmo comando repetidas vezes. Como podemos lidar com isso?

Comandos de repetição!!



```
#include <stdio.h>

int main() {
    float temperaturas[365];
    printf("Informe as temperaturas dos ultimos 365 dias:");

    for (int i = 0; i < 365; i++) {
        scanf("%f", &temperaturas[i]);
    }

    return 0;
}
```



```
#include <stdio.h>

int main() {
    float temperaturas[365];
    printf("Informe as temperaturas dos ultimos 365 dias:");

    for (int i = 0; i < 365; i++) {
        scanf("%f", &temperaturas[i]);
    }

    for (int i = 0; i < 365; i++) {
        printf("%f ", temperaturas[i]);
    }
    printf("\n");

    return 0;
}
```



Dia 4

Matrizes

Motivação

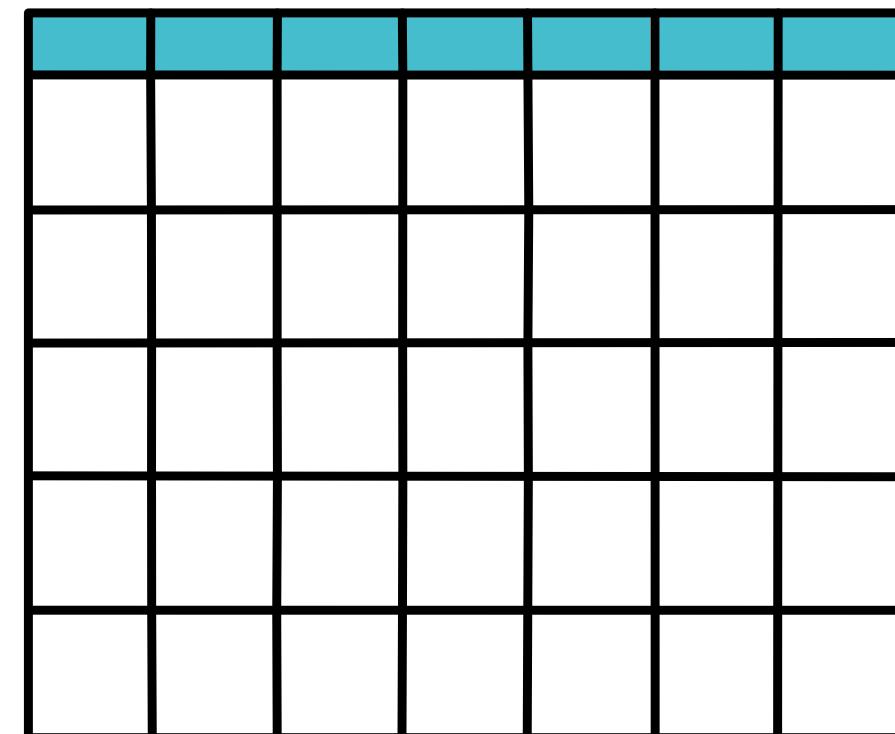
Os vetores vistos até agora funcionam como listas unidimensionais de elementos.

Contudo, existem situações em que é útil tratar uma coleção de dados em mais de uma dimensão, com linhas e colunas (ex.: tabelas).



Matrizes

Matriz é o nome dado a um vetor de duas dimensões. De certa forma, uma matriz pode ser interpretada como um vetor de vetores



Declarando uma matriz

Forma mais simples:

```
tipo nome[num_linhas][num_colunas];
```

Exemplos:

```
int matriz_1[100][20];
```

```
float matriz_2[3][4];
```

```
int matriz_3[2][2];
```



Declarando uma matriz

Inicializando os valores na declaração:

```
tipo nome[num_linhas][num_colunas] = {dados};
```

Exemplos:

```
int matriz[2][4] = {{1, 2, 3, 4}, {5, 6, 7, 8}};
```

```
int matriz[2][4] = {1, 2, 3, 4, 5, 6, 7, 1};
```



Declarando uma matriz

Podemos omitir apenas o número de linhas:

```
tipo nome[ ][num_colunas] = {dados};
```

Exemplos:

```
int matriz[ ][4] = {{1, 2, 3, 4}, {5, 6, 7, 8}};
```

```
int matriz[ ][4] = {1, 2, 3, 4, 5, 6, 7, 8};
```



Acessando elementos

Os elementos de matrizes também são acessados por meio de índices. Porém, como temos duas dimensões, precisamos usar dois índices distintos:

```
nome_matriz[indice_linha][indice_col];
```



Acessando elementos

Exemplo:

```
int matriz[2][3] = {{1, 2, 3}, {4, 5, 6}};
```

// Qual valor eh impresso?

```
printf("%d\n", matriz[1][2]);
```



Acessando elementos

Exemplo:

```
int matriz[2][3] = {{1, 2, 3}, {4, 5, 6}};
```

```
// Imprime: 6
```

```
printf("%d\n", matriz[1][2]);
```



Operações com matrizes

Também podemos usar comandos de repetição para realizar operações com matrizes.

Porém, como precisamos acessar elementos em duas dimensões distintas, precisamos aninhar os comandos de repetição



```
#include <stdio.h>

int main() {
    int matriz[3][3];

    for (int linha = 0; linha < 3; linha++) {
        for (int col = 0; col < 3; col++) {
            scanf("%d", &matriz[linha][col]);
        }
    }

    return 0;
}
```



```
#include <stdio.h>

int main() {
    int matriz[3][3];

    // Qual eh a diferenca?
    for (int col = 0; col < 3; col++) {
        for (int linha = 0; linha < 3; linha++) {
            scanf("%d", &matriz[linha][col]);
        }
    }

    return 0;
}
```



```
#include <stdio.h>

int main() {
    int matriz[2][2] = {{1, 2}, {3, 4}};

    for (int linha = 0; linha < 2; linha++) {
        for (int col = 0; col < 2; col++) {
            printf("%d ", matriz[linha][col]);
        }
        printf("\n");
    }
    // Resultado:
    // 1 2
    // 3 4

    return 0;
}
```





PROGRAMAÇÃO 101

dia 5

- Strings
- Funções
- Apêndice

Dia 5 Strings

Vetor de caracteres: String

String é definido como um vetor de caracteres, logo usamos o tipo char.

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

int main() {
    char texto[6] = "Ola";
    //um vetor de caracteres de 6 posições
    return 0;
}
```



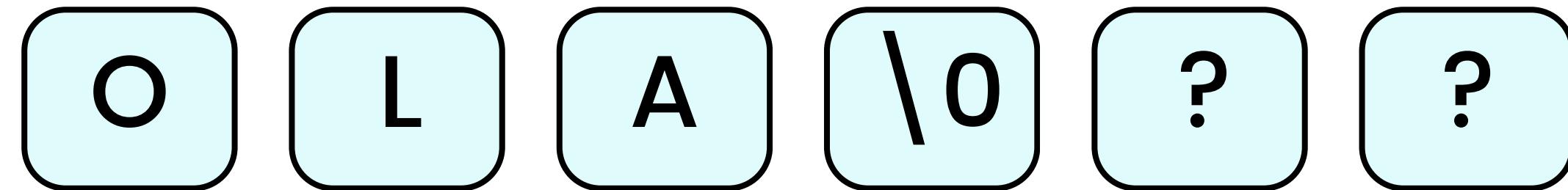
IMPORTANTE !

As strings têm no elemento seguinte a última letra da palavra/frase armazenada, um caractere '\0' que indica o fim da sequência de caracteres.

Leia-se: o caractere especial é definido por - contrabarra \ e o número 0

```
..  
char texto[6] = "ola";  
..
```

A string Ola definida acima está armazenada na memória da seguinte forma:

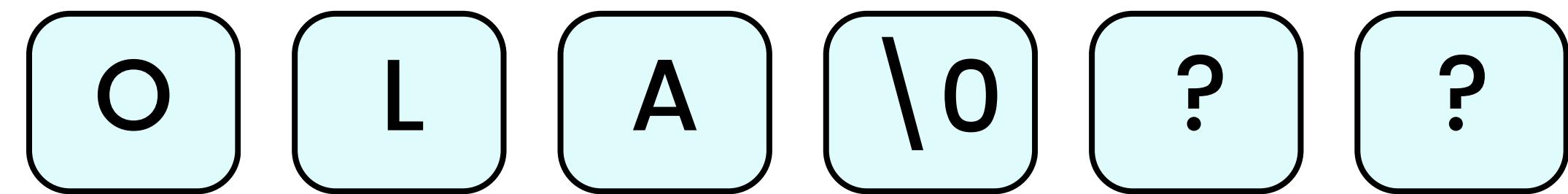


IMPORTANTE !

Sempre que definimos uma string em C, precisamos considerar o tamanho do vetor da seguinte forma:

char texto[6] -> isso significa que a string que iremos armazenar precisa conter ATÉ 5 CARACTERES, pois o caractere especial \0 também ocupa um espaço desse vetor.

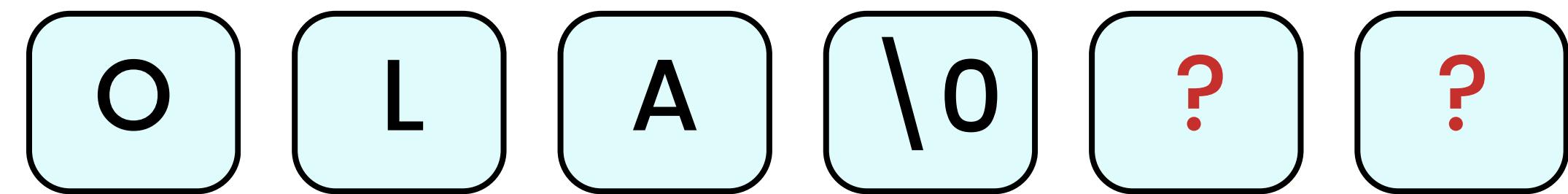
```
char texto[6] = "ola";
```



Nesse caso armazenamos uma string de 3 letras (ola) + o caracter especial representando o final da string \0 resultando em 4 posições ocupadas.



```
char texto[6] = "ola";
```



Como criamos uma variável com 6 posições, as duas últimas ficaram “vazias”. Você ainda pode acessá-las porém resultará em valores desconhecidos o que chamamos de lixo de memória.



String também é um vetor

Podemos utilizar todo nosso conhecimento aprendido em vetores para manipular as strings, pois ela também é um vetor.



String também é um vetor

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

int main() {
    char texto[6] = "Ola";
    texto[0] = 'P'; //usamos 'aspas simples'
    //a string texto que antes armazenava Ola passa a armazenar Pla

    return 0;
}
```



IMPORTANTE !

Existe uma diferença entre “aspas duplas” e ‘aspas simples’:

“aspas duplas”: é usada para inicializar uma **STRING**.
‘aspas simples’: é usada para inicializar um **CARACTERE**.

Manipulando strings

Em C utilizamos a biblioteca `string.h` para manipular strings, como copiar, ler, escrever, comparar, encontrar o comprimento entre outros.

```
#include <stdio.h>
#include <string.h> //incluimos essa biblioteca

int main() {
    ...
    return 0;
}
```



Copiando uma string

Você pode copiar uma string de duas formas:

- Não utilizando a biblioteca string.h
- Utilizando a biblioteca string.h



Sem usar <string.h>

```
#include <stdio.h>

int main() {
    char texto[6] = "Ola";
    char texto2[6];
    for (int pos = 0; texto[pos] != '\0'; pos++) {
        texto2[pos] = texto[pos]; //copiando para texto2
    }
    texto2[pos] = '\0'; //lembre-se de adicionar o caractere especial

    return 0;
}
```



Usando <string.h>

Para copiar usando a biblioteca string.h utilizaremos a função strcpy que recebe dois parâmetros:

strcpy(char *destino, char *fonte);

destino: variável que irá armazenar a string a ser copiada.

fonte: variável que já está armazenando a string que será copiada.



Usando <string.h>

```
#include <stdio.h>
#include <string.h> //é necessário importar para usar a função strcpy

int main() {
    char texto[6] = "Ola", texto2[6];

    strcpy(texto2, texto); //função presente na biblioteca string.h

    return 0;
}
```



Lendo uma string

Para ler uma string utilizaremos a seguinte função

fgets(char *string, int tamanho, FILE *fp);

string: a string a ser lida.

tamanho: um número inteiro que representa o máximo de caracteres a serem lidos.

fp: o local de onde iremos ler a string, usaremos, **stdin**



IMPORTANTE !

```
fgets(char *string, int tamanho, FILE *fp);
```

O campo `*fp` representa de onde a string será lida. Como nesse curso usaremos o dispositivo de entrada (input) padrão (standard), no geral o teclado, nomearemos o terceiro campo com o termo `stdin`.

`stdin`: standard input

Regras do fgets

- A função fgets irá ler a string até o que o caractere de nova linha seja lido, '\n' , ou até que **tamanho-1** seja alcançado.
- O caractere de nova linha, '\n' , fará parte da string caso seja lido.
- A leitura máxima é de **tamanho-1** pois o caractere especial '\0' sempre é adicionado ao final da string.



Lendo uma string

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

int main() {
    char texto[6];
    fgets(texto, 6, stdin);
    //usamos stdin pra representar a entrada padrão do dispositivo
    //na verdade temos que armazenar uma string de no máximo 5 caracteres
    //a última posição ficará armazenado o caractere especial \0

    return 0;
}
```



Escrevendo uma string na tela

Para escrever uma string utilizaremos a seguinte função

```
fputs(char *string, FILE *fp);
```

string: a string que será escrita.

fp: o local para onde iremos escrever a string, usaremos,
stdout



IMPORTANTE !

```
fputs(char *string, FILE *fp);
```

No fgets, usamos o campo fp como o dispositivo de entrada padrão, já no fputs usaremos como dispositivo de saída (output) padrão (standard), que geralmente é a tela do computador.

stdout: standard output

Escrevendo uma string na tela

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

int main() {
    char texto[6] = "Ola";
    fputs(texto, stdout);
    //usamos stdout pra representar a saída padrão do dispositivo

    return 0;
}
```



Comparando strings

Para comparar duas strings utilizaremos a função

strcmp(char *string1, char *string2);

string1 e string2: as strings a serem comparadas.

Retorno:

- Se as strings forem iguais, strcmp retorna o número 0.
- Qualquer outro número, strings diferentes.



Comparando strings

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

int main() {
    char texto1[6] = "Ola", texto2[6] = "Ola";

    if(strcmp(texto1, texto2) == 0){ //caso seja igual strcmp retorna 0
        printf("Strings iguais.\n");
    }else{
        printf("Strings diferentes.\n");
    }
    return 0;
}
```



Comprimento da string

Para encontrar o comprimento da string, utilizaremos

strlen(char *string);

string: string que devemos encontrar o tamanho

Retorno:

- Retorna o número de caracteres da string (o caractere especial \0 não é adicionado a esse tamanho)



Comprimento da string

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

int main() {
    char texto[6] = "Teste";
    printf("%li", strlen(texto));
    //retorna o número 5, pois a palavra Teste tem 5 letras.
    //usamos %li ao invés de %d pois strlen retorna um tipo chamado size_t

    return 0;
}
```



Explore a «string.h»

Existem ainda diversas outras funções na biblioteca **string.h**. Sempre que necessário você pode consultar alguma documentação sobre a biblioteca.

[Documentação para biblioteca <string.h>](#)



dia 5

Funções

Motivação

Nesse ponto do aprendizado já sabemos o suficiente para desenvolver um sistema simples.

Mas se além disso, pudessemos desenvolver um programa organizado e de fácil compreensão?



Problema das repetições

```
...
for(int i = 0; i < 10; i++){
    printf("%c", '*');
}
printf("%c", '\n');
printf("Números entre 1 a 5\n");
for(int i = 0; i < 10; i++){
    printf("%c", '*');
}
printf("%c", '\n');
for(int i = 1; i <= 5; i++){
    printf("%d ", i);
}
...
...
```

Como vemos nesse trecho ao lado, um grande problema no início do nossa aprendizado com programação é a repetição excessiva de algumas partes do código.



Problema das repetições

Repetições de código podem causar alguns problemas quando usados em projetos de médio a grande porte:

- Dificuldade de compreensão do programa pois teremos diversas repetições ao longo do programa.
- Dificuldade de manutenção pois se alterarmos um trecho repetido, teremos que replicar a mudança em todos os locais.



Problema das repetições

```
...  
for(int i = 0; i < 10; i++){  
    printf("%c", '*');  
}  
printf("%c", '\n');  
printf("Números entre 1 a 5\n");  
for(int i = 0; i < 10; i++){  
    printf("%c", '*');  
}  
printf("%c", '\n');  
for(int i = 1; i <= 5; i++){  
    printf("%d ", i);  
}  
...  
...
```

O ideal é que esses trechos possam ser definidos uma única vez e ser chamados em todos os pontos necessários ao longo do código.



Criando a função linha

```
#include <stdio.h>
//função propriamente dita
void linha(){
    for(int i = 0; i < 10; i++){
        printf("%c", '*');
    }
    printf("%c", '\n');
}

int main(){
    linha(); //chamada da função
    printf("Números entre 1 a 5\n");
    linha(); //chamada da função
    for(int i = 1; i <= 5; i++){
        printf("%d ", i);
    }
    return 0;
}
```

Removemos os dois trechos de código repetido e definimos a função `linha()`.

Melhoramos a compreensão, a manutenção e também possibilitamos a reutilização em outros pontos do código.



Função: Definição

Função é um bloco de código nomeado e que pode ser chamado em diversas partes do programa que realiza uma tarefa específica dentro do sistema.



Já usamos funções!

Desde o primeiro dia de curso utilizamos diversas funções que ate então não sabíamos como chamava

- `printf()` - função para escrever na tela.
- `scanf()` - função para ler dados do teclado.
- `main()` - função principal do código onde o programa começa a ser executado.



Declarando funções

Todas as funções possuem uma estrutura padrão para ser declarada que deve ser respeitado.

```
tipo nome(parametros){  
    corpo  
}
```



Declarando funções

```
tipo nome(parametros){  
    corpo  
}
```

tipo: Você define qual tipo a função irá retornar (se retornar). Inicialmente sua função pode ser dos tipos, int, float, double, char ou void.

nome: O nome da sua função.

parametros: Conjunto de dados que sua função poderá receber para realizar a tarefa destinada a ela.

corpo: Contém a lógica para realizar a tarefa.



Declarando funções

```
tipo nome(parametros){  
    corpo  
}
```

tipo: Você define qual tipo a função irá retornar (se retornar).

nome: O nome da sua função.

parametros: Conjunto de dados que sua função poderá receber para realizar a tarefa destinada a ela.

corpo: Contém a lógica para realizar a tarefa.



Tipo da função

```
tipo nome(parametros){  
    corpo  
}
```

tipo: Você define qual tipo a função irá retornar (se retornar).

Uma função pode retornar qualquer tipo, isto é, após ser executada ela irá devolver um valor resultante da computação realizada dentro do corpo.

Alguns tipos válidos para funções são: int, double, float, char, void entre outros.



IMPORTANTE !

O tipo **void** significa que a sua função não tem retorno, ou seja, após realizada a tarefa ela não devolve um resultado para quem a chamou.

Caso sua função tenha **QUALQUER OUTRO TIPO** que não seja **void**, é OBRIGATÓRIO ter o comando **return** indicando o valor a ser retornado.

Tipo da função: tipo void

```
#include <stdio.h>  
  
//função propriamente dita  
void linha(){  
    for(int i = 0; i < 10; i++){  
        printf("%c", '*');  
    }  
    printf("%c", '\n');  
}  
  
int main(){  
    linha(); //chamada da função  
    ...  
    return 0;  
}
```

A função `linha()` é chamada dentro do `main` e por ser do tipo `void`, ela não retorna nenhum valor para quem a chamou.

Por ser do tipo `void`, **não usamos** o comando `return`.



Tipo da função: outros tipos

```
...  
int quadrado(){  
    return 4 * 4;  
    //retorna 16 pra quem chamou  
}  
  
int main(){  
    int numero = quadrado();  
    //numero recebe o valor 16  
    ...  
    return 0;  
}
```

A função `quadrado()` ao lado é definida com o tipo `int`, assim ela necessariamente precisa ter o comando `return` para indicar o valor do resultado a ser retornado.

Veja que a variável `numero` presente na função `main` irá armazenar o valor 16, resultado retornado da função `quadrado()`



Nome da função

```
tipo nome(parametros){  
    corpo  
}
```

nome: O nome da sua função.

Forma de identificar a sua função no sistema.



Parâmetros

```
tipo nome(parametros){  
    corpo  
}
```

parametros: Conjunto de dados que sua função poderá receber para realizar a tarefa destinada a ela.

É por meio dos parâmetros que uma função **recebe informações do programa principal**, isto é, de quem o chamou.

Caso sua função tenha parâmetros, a declaração deve seguir a seguinte estrutura:

(tipo1 nome1, tipo2 nome2, ..., tipoN nomeN)



Sem parâmetros

Caso sua função não tenha parâmetros, você deve deixar o conteúdo entre os parênteses vazio, assim você indica que nenhum dado será passado pelo programa principal.

```
...
//função sem parâmetros
int quadrado(){
    return 4 * 4;
}

int main(){
    int numero = quadrado();
    //numero recebe o valor 16
    ...
    return 0;
}
```



Passagem de parâmetros

```
...  
int quadrado(int a){  
    return a * a;  
    //retorna 16 pra quem chamou  
}  
  
int main(){  
    int numero = 4, resultado;  
    resultado = quadrado(numero);  
    //resultado recebe o valor 16  
    ...  
    return 0;  
}
```

A função quadrado agora recebe o parâmetro “int a”.

Veja na função main, que ao chamar quadrado estamos passando a variável numero definida no programa principal.

Assim, “a” contém o valor que foi passado para a função (o valor da variavel numero).



IMPORTANTE !

Na linguagem C, os parâmetros de uma função são sempre passados como valor, isso significa que, uma cópia do valor do parâmetro passado é criado para ser usado na função, assim o valor desse parâmetro é alterado somente dentro da função.

Nesse curso não abordaremos passagem por referência que é uma outra forma de passagem de parâmetros sem usar cópias. Isto será aprofundado durante a disciplina do seu curso.

Passagem por valor

```
...  
int quadrado(int a){  
    return a * a;  
}  
  
int main(){  
    int numero = 4, resultado;  
    resultado = quadrado(numero);  
    //numero permanece com valor 4  
    //a e numero são variáveis diferentes  
    ...  
    return 0;  
}
```

A variável **numero** passada como parâmetro da função **quadrado** permanece com o **mesmo valor** que lhe foi atribuído na função **main**.

É criado uma cópia do valor 4 e atribuído para variável “a”.



IMPORTANTE !

Vetores, matrizes, strings, que estudamos anteriormente são passados como referência para uma função, o que necessita um ferramental mais aprofundado sobre alguns conceitos que não serão abordados nesse curso por ser apenas introdutório.

Assim, todos os nossos exemplos terão o uso de parâmetros apenas definidos pelos tipos primitivos da linguagem C: int, float, double, char.

Função - corpo

```
tipo nome(parametros){  
    corpo  
}
```

corpo: Contém a lógica para realizar a tarefa.



Função - corpo

Dentro do corpo de uma função (que não seja a função main), é recomendado seguir algumas boas práticas e não cometer alguns erros:

- Mantenha dentro do corpo da função somente a lógica necessária para realizar a tarefa.
- As operações de entrada e saída (`printf` e `scanf`) devem ser feitos na função main.
- Isso garante que a função tenha generalidade, ou seja, possa ser usada em diversos casos.



O último passo: Escopo

Para finalizar nosso aprendizado introdutório de funções vamos aprender sobre o conceito de **escopos**.

Escopo define o local onde as variáveis podem ou não ser acessadas e usadas.

Aqui, vamos aprender o conceito de **variáveis locais** muito utilizado nas funções.



Variáveis locais

Variáveis locais são aquelas que só podem ser acessadas e usadas dentro do bloco de código no qual foi declarado.

O bloco de código é definido por todo o conteúdo que está entre as chaves {}.

Assim seja um if/else, for/while ou em uma função, quaisquer variáveis declaradas dentro desse bloco só podem ser acessadas e usadas naquele ponto.



Variáveis locais

```
...  
int main(){  
    int numero = 4;  
    if(numero == 4){  
        int i = 2;  
        printf("%d", i);  
    }  
    printf("%d", numero);  
    printf("%d", i); //erro  
    return 0;  
}
```

Definimos a variável `i` dentro do `if`, que foi delimitado pelas chaves, sendo assim essa variável só pode ser usada dentro desse escopo.

Caso tente usar `printf` na variável `i` fora do bloco do `if`, você terá um erro no código, pois `i` pertence somente ao escopo de `if`.



Variáveis locais

```
...  
int quadrado(){  
    int numero = 4;  
    return numero * numero;  
}  
  
int main(){  
    int numero = quadrado();  
    ...  
    return 0;  
}
```

Variáveis locais também estão presentes em funções.

No exemplo ao lado, temos a definição de `numero` em dois locais diferentes, na função `main` e na função `quadrado`.

Apesar de terem o mesmo nome, são variáveis diferentes, pois estão em escopos diferentes.



Dia 5

Apêndice

O que faltou no curso?

O intuito do curso é ser introdutório, assim alguns temas não foram cobertos aqui por serem avançados mas serão abordados durante a disciplina de programação do seu curso:

- Passagem por referência
- Ponteiros
- Alocação dinâmica
- Struct
- Recursividade
- Arquivos





PROGRAMAÇÃO 101