

Introdução à Linguagem C

Prof. Paulo Henrique Pisani

fevereiro/2022

Tópicos

- Linguagem C: Tipos de dados, Entrada/Saída, Operadores aritméticos, condicionais, lógicos, Estruturas de repetição;
- Vetores;
- Funções;
- Passando vetores como argumento;
- Indentação;
- Strings;
- Matrizes.

Linguagem C

Linguagem C

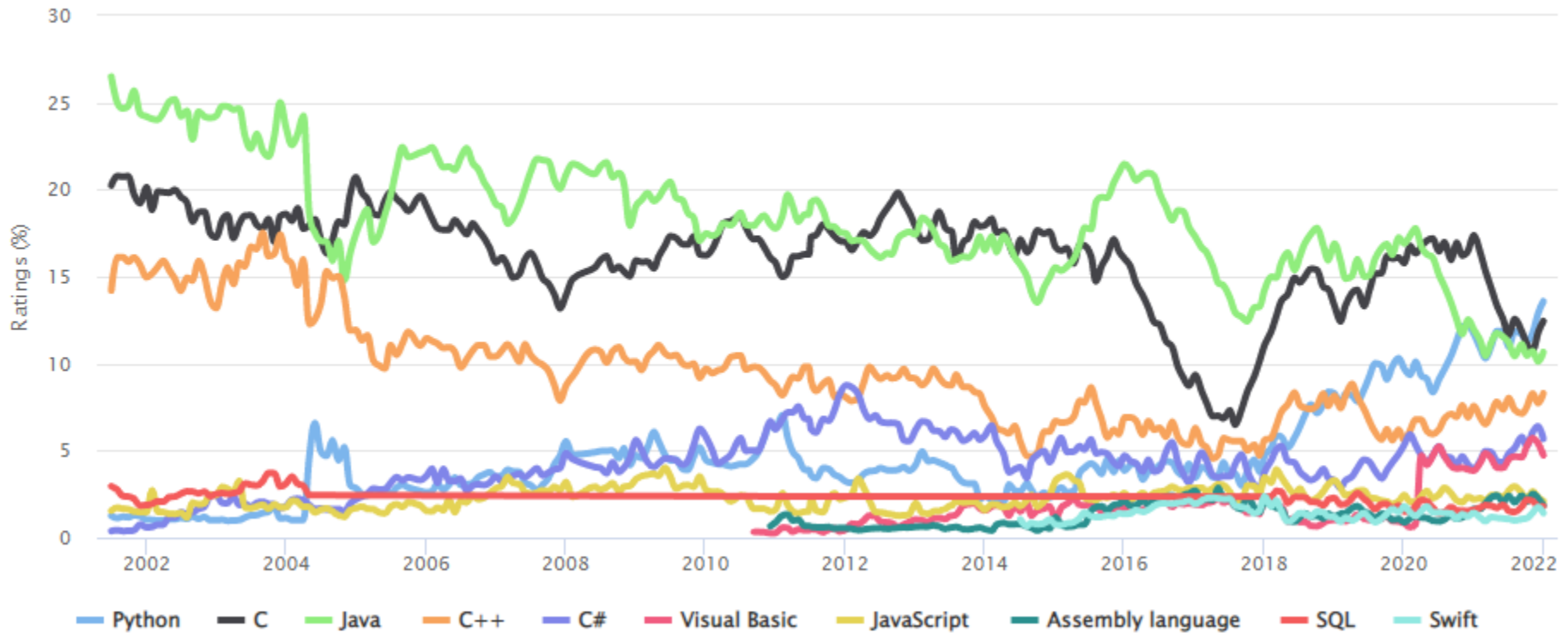
- Foi criada por Dennis Ritchie;
- Diversas variantes surgiram:
 - K & R (1978);
 - C89 (ANSI X3.159-1989 "Programming Language C.")
 - C99 (ISO/IEC 9899:1999)
 - C11 (ISO/IEC 9899:2011)
 - C18 (ISO/IEC 9899:2018)

Rank	Language	Type	Score
1	Python▼	  	100.0
2	Java▼	  	95.4
3	C▼	  	94.7
4	C++▼	  	92.4
5	JavaScript▼		88.1
6	C#▼	   	82.4
7	R▼		81.7
8	Go▼	 	77.7
9	HTML▼		75.4
10	Swift▼	 	70.4

<https://spectrum.ieee.org/top-programming-languages/> (21/01/2022)

TIOBE Programming Community Index

Source: www.tiobe.com



<https://www.tiobe.com/tiobe-index/> (21/01/2022)

Programa mínimo

```
#include <stdio.h>

int main() {
    printf("ABC");
    return 0;
}
```

- Para compilar:

gcc teste.c -o teste.exe

Código-fonte

Programa objeto

- Para executar:

./teste.exe

Programa (realmente) mínimo

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

- Para compilar:

gcc teste.c -o teste.exe

Código-fonte

Programa objeto

- Para executar:

./teste.exe

Tipos de dados

int	Atualmente, é o long int
short int	Inteiro de 2 bytes (-32.768 a 32.767)
long int	Inteiro de 4 bytes (-2^{31} a $2^{31}-1$)
long long int	Inteiro de 8 bytes (-2^{63} a $2^{63}-1$)
unsigned int	Versões dos tipos inteiro “sem sinal”
unsigned short int	
unsigned long int	
unsigned long long int	

float	Ponto flutuante de 4 bytes
double	Precisão dupla de 8 bytes

Tipos de dados

Importante! Atenção aos limites dos tipos!

int	Atualmente, é o long int
short int	Inteiro de 2 bytes (-32.768 a 32.767)
long int	Inteiro de 4 bytes (-2^{31} a $2^{31}-1$)
long long int	Inteiro de 8 bytes (-2^{63} a $2^{63}-1$)
unsigned int	Versões dos tipos inteiro “sem sinal”
unsigned short int	
unsigned long int	
unsigned long long int	

float	Ponto flutuante de 4 bytes
double	Precisão dupla de 8 bytes

Tipos de dados

`char` 1 byte (-128 a 127)

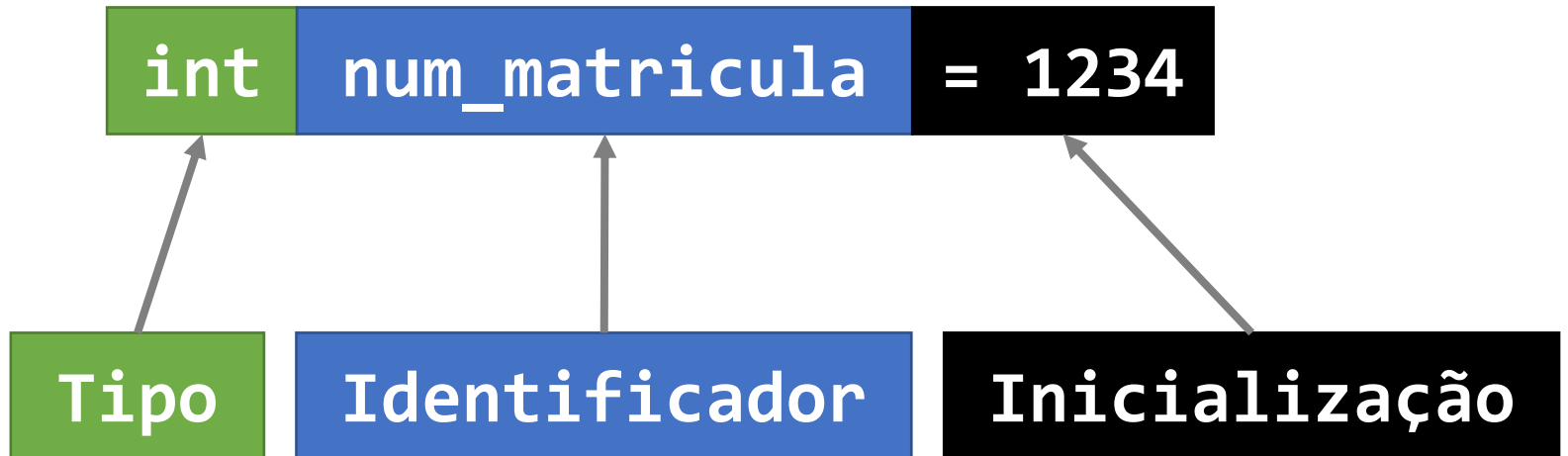
`unsigned char` 1 byte (0 a 255)

Não há um tipo `String`. Strings são representadas por vetores de `char` (e o último `char` é o `'\0'`).

Também não há tipo booleano! Podemos usar `int` ou `char`:
Valor = 0 -> FALSO
Valor != 0 -> VERDADEIRO

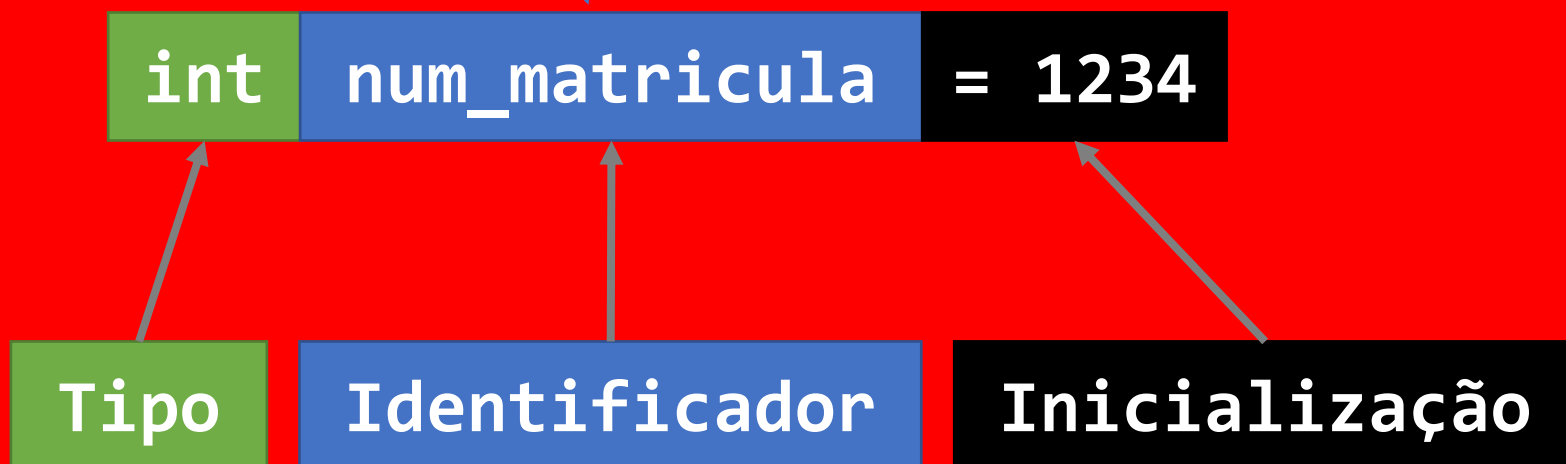
Declaração de variáveis

```
int main() {  
    int num_matricula = 1234;  
    return 0;  
}
```



Cuidado! A linguagem C é *case-sensitive*, ou seja:
A identificador “num_matricula” é diferente de
“Num_matricula”!

```
int main() {  
    int num_matricula = 1234;  
    return 0;  
}
```



Entrada e Saída


- Na linguagem C, são utilizadas funções para as operações de entrada e saída;
- As funções básicas ficam na biblioteca padrão: a **stdio**!
- Para usar essa biblioteca, adicionamos a seguinte linha:



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

Entrada e Saída

- Durante o curso, utilizaremos esta biblioteca em praticamente todos os programas, talvez em todos mesmo!
- Portanto, nosso programa mínimo torna-se:



```
#include <stdio.h>

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

Saída

- Para imprimir um valor, usamos a função **printf**

```
int printf( const char * format, ... );
```

Formato



The diagram illustrates the components of the printf function signature. A blue arrow points from the label 'Formato' to the 'const char * format' parameter. Another blue arrow points from the label 'Valores' to the '...' parameter.

Valores

Saída

- Exemplos:

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

```
int main() {
```

```
    printf("ABC");
```

```
    int num = 507;
```

```
    printf("%d", num);
```

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

```
    printf("A sala do professor eh a %d\n", num);
```

```
    printf("%c + %c = %d\n", 'A', 'B', num);
```

```
    return 0;
```

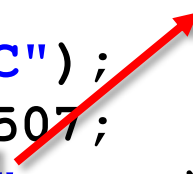
```
}
```

Saída

- Exemplos:

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

```
int main() {  
    printf("ABC");  
    int num = 507;  
    printf("%d", num);  
    printf("%d\n", num);  
    printf("A sala do professor eh a %d\n", num);  
    printf("%c + %c = %d\n", 'A', 'B', num);  
    return 0;  
}
```



%d	int
%ld	long int
%lld	long long int
%f	float
%lf	double
%c	char
%s	String (vetor de char)
%p	Ponteiro (endereço de memória)

Entrada

- Para ler um valor usamos a função **scanf**

```
int scanf( const char * format, ... );
```

Formato



Endereços
das variáveis

Entrada

- Exemplos:

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

```
int main() {  
    printf("Digite um numero inteiro:\n");  
    int num_int;  
    scanf("%d", &num_int);  
    return 0;  
}
```

Entrada

- Exemplos:

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

```
int main() {  
    printf("Digite um numero inteiro:\n");  
    int num_int;  
    scanf("%d", &num_int);  
    return 0;  
}
```



&num_int

Há um “&” antes do identificador da variável!!!

Entrada

- Exemplos:

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

```
int main() {  
    printf("Digite um numero inteiro:\n");  
    int num_int;  
    scanf("%d", &num_int);  
    return 0;  
}
```



&num_int

Há um “&” antes do identificador da variável!!!

O scanf recebe os endereços de memória das variáveis. O “&” serve para obter o endereço de memória da variável “num_int”. Quando trabalharmos com ponteiros, veremos que nem sempre é necessário usar o “&”.

Entrada

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

```
int main() {  
    printf("Digite um numero inteiro:\n");  
    int num_int;  
    scanf("%d", &num_int);  
  
    printf("Digite um numero fracionario:\n");  
    double num_frac;  
    scanf("%lf", &num_frac);  
  
    printf("INT=%d DOUBLE=%lf\n", num_int, num_frac);  
  
    int a, b;  
    printf("Digite dois numeros:\n");  
    scanf("%d %d", &a, &b);  
    printf("A=%d B=%d\n", a, b);  
  
    return 0;  
}
```

Entrada e saída

Lembrar da incluir a biblioteca `stdio`!

- **printf**: recebe **valores** como argumento;
- **scanf**: recebe **endereços de memória** como argumento.



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

```
int main() {  
    printf("Digite um numero inteiro:\n");  
    int num_int;  
    scanf("%d", &num_int);  
    printf("O numero eh %d", num_int);  
    return 0;  
}
```


Tamanhos dos tipos de dados (sizeof)

- Para saber quantos bytes um tipo de dados ocupa, usamos **sizeof** (o retorno é do tipo **long int**).

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

```
int main() {  
    int a;  
    long int b;  
    long long int c;  
    float d;  
    double e;  
    char f;
```

```
    printf("%ld %ld %ld %ld %ld %ld",  
           sizeof(a),  
           sizeof(b),  
           sizeof(c),  
           sizeof(d),  
           sizeof(e),  
           sizeof(f));
```

```
    return 0;
```

```
}
```

Conversão de tipo

- Expressões com diversos tipos de dados:

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

```
int main() {
```

```
    float num;
```

```
    num = 4 / 2;
```

```
    printf("%.2f\n", num);
```

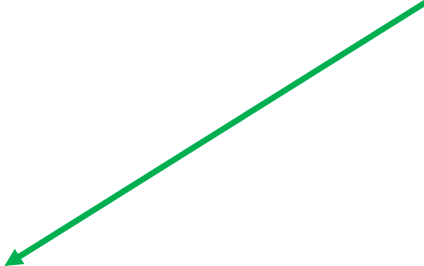
```
    num = 10 / 2.5;
```

```
    printf("%.2f\n", num);
```

```
    return 0;
```

```
}
```

Formata o número com
duas casas decimais



Saída

?

Conversão de tipo

- Expressões com diversos tipos de dados:

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

```
int main() {
```

```
    float num;
```

```
    num = 4 / 2;
```

```
    printf("%.2f\n", num);
```

```
    num = 10 / 2.5;
```

```
    printf("%.2f\n", num);
```

```
    return 0;
```

```
}
```

Saída

2.00

4.00

Conversão de tipo

- Expressões com diversos tipos de dados:

```
#include<stdio.h>

int main() {

    float num, a=5, b=2;

    num = a / b;

    printf("%.2f\n", num);

    return 0;
}
```

Saída

?

Conversão de tipo

- Expressões com diversos tipos de dados:

```
#include<stdio.h>

int main() {

    float num, a=5, b=2;

    num = a / b;

    printf("%.2f\n", num);

    return 0;
}
```

Saída

2.50

Conversão de tipo

- Expressões com diversos tipos de dados:

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

```
int main() {
```

```
    float num;
```

```
    num = 4 / 2;
```

```
    printf("%.2f\n", num);
```

```
    num = 5 / 2;
```

```
    printf("%.2f\n", num);
```

```
    return 0;
```

```
}
```

Saída

?

Conversão de tipo

- Expressões com diversos tipos de dados:

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

```
int main() {
```

```
    float num;
```

```
    num = 4 / 2;
```

```
    printf("%.2f\n", num);
```

```
    num = 5 / 2;
```

```
    printf("%.2f\n", num);
```

```
    return 0;
```

```
}
```

Saída

2.00

2.00

Conversão de tipo

- Atenção com operações envolvendo tipos fracionários!

```
float num;
```

<code>num = 5 / 2;</code>	→	2
<code>num = 5 / 2.0;</code>	→	2.5
<code>num = 5 / ((float) 2);</code>	→	2.5

Conversão de tipo

- Conversão entre tipos:

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

```
int main() {
```

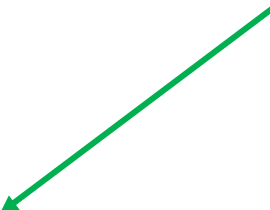
```
    int n1 = 507;  
    long long int n2 = n1;
```

```
    printf("%lld\n", n2);
```

```
    return 0;
```

```
}
```

Ok! Armazena valor (int) em
variável com tipo de dados que
possui faixa maior (long long int)



Saída

507

Conversão de tipo

- Conversão entre tipos:

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

```
int main() {
```

```
    int n1 = 507;
```

```
    short int n2 = n1;
```

```
    printf("%d\n", n2);
```

```
    return 0;
```

```
}
```

Perigoso! Armazena valor (int) em variável com tipo de dados que possui faixa MENOR (short int)



Saída

507

Conversão de tipo

- Conversão entre tipos:

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

```
int main() {
```

```
    int n1 = 50000;  
    short int n2 = n1;
```

```
    printf("%d\n", n2);
```

```
    return 0;
```

```
}
```

Perigoso! Armazena valor (int) em variável com tipo de dados que possui faixa MENOR (short int)



Saída

-15536

Conversão de tipo

- Conversão entre tipos:

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

```
int main() {
```

```
    int n1 = 50000;  
    short int n2 = n1;
```

```
    printf("%hd\n", n2);
```

```
    return 0;
```

```
}
```

Perigoso! Armazena valor (int) em variável com tipo de dados que possui faixa MENOR (short int)



Saída

-15536

Conversão de tipo

- Conversão entre tipos usando **cast**:

```
float n = (float) 507;  
int t = (int) n;
```



Força a conversão de tipo

Conversão de tipo

- Conversão entre tipos usando **cast**:

```
#include <stdio.h>
int main() {
    float r1 = 7 / 2;
    float r2 = ((float) 7) / 2;
    float r3 = 7 / ((float) 2);
    float r4 = ((float) 7) / ((float) 2);
    float r5 = (float) 7 / 2;
    float r6 = (float) (7 / 2);

    printf("%.1f %.1f %.1f\n", r1, r2, r3);
    printf("%.1f %.1f %.1f\n", r4, r5, r6);

    return 0;
}
```

Saída

?

Conversão de tipo

- Conversão entre tipos usando **cast**:

```
#include <stdio.h>
int main() {
    float r1 = 7 / 2;
    float r2 = ((float) 7) / 2;
    float r3 = 7 / ((float) 2);
    float r4 = ((float) 7) / ((float) 2);
    float r5 = (float) 7 / 2;
    float r6 = (float) (7 / 2);

    printf("%.1f %.1f %.1f\n", r1, r2, r3);
    printf("%.1f %.1f %.1f\n", r4, r5, r6);

    return 0;
}
```

Type cast tem precedência sobre o operador de divisão.

Saída

3.0 3.5 3.5
3.5 3.5 3.0

Inicialização de variáveis

- **Sempre inicie variáveis em C!**
- Quando uma variável local é declarada, o programa apenas reserva um espaço de memória para ela:
 - **Mas o espaço alocado não é inicializado!**
 - **Portanto, uma variável não inicializada pode ter qualquer valor!**

Inicialização de variáveis

- **Sempre inicie variáveis em C!**
- Quando uma variável local é declarada, o programa apenas reserva um espaço de memória para ela:
 - **Mas o espaço alocado não é inicializado!**
 - **Portanto, uma variável não inicializada pode ter qualquer valor!**

Sempre inicialize variáveis!

- Qual a saída deste programa?

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

```
int main() {
```

```
    int num;
```

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

```
    return 0;
```

```
}
```

Sempre inicialize variáveis!

- Qual a saída deste programa?

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

```
int main() {
```


```
    int num;
```

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

```
    return 0;
```

```
}
```

Variável não foi inicializada! A saída será o que estiver na área alocada! Pode ser qualquer valor!



Resultado da compilação usando -Wall:

```
teste.c: In function 'main':  
teste.c:9:5: warning: 'num' is used uninitialized in this function  
[-Wuninitialized]  
    printf("%d\n", num);
```

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

```
int main() {
```

```
    int num;
```

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

```
    return 0;
```

```
}
```

Operadores aritméticos

+	Soma
-	Subtração
*	Multiplicação
/	Divisão
%	Resto da divisão

Operadores relacionais

< Menor que

> Maior que

<= Menor ou igual a

>= Maior ou igual a

== Igual a

!= Diferente de

Operadores lógicos

&&	E
	Ou
!	Negação

Operadores de atribuição

++	Incremento unitário
--	Decremento unitário
+=	Atribuição por soma
-=	Atribuição por subtração
*=	Atribuição por multiplicação
/=	Atribuição por divisão
%=	Atribuição por resto da divisão

Operadores de atribuição

Não use esses operadores mais de uma vez na mesma linha!

++	Incremento unitário
--	Decremento unitário
+=	Atribuição por soma
-=	Atribuição por subtração
*=	Atribuição por multiplicação
/=	Atribuição por divisão
%=	Atribuição por resto da divisão

Estruturas condicionais

```
if (<condicao>) {  
  
}
```

```
if (<condicao>) {  
  
} else {  
  
}
```

```
<condicao> ? <retorno verdadeiro> : <retorno falso>
```

Atenção!

= é diferente de ==

- Qual a saída deste programa?

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

```
int main() {  
    int n = 507;  
  
    if (n = 5)  
        printf("n eh 5.\n");  
    else  
        printf("n nao eh 5.\n");  
  
    return 0;  
}
```

Atenção!


= é diferente de ==

- Qual a saída deste programa?

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

```
int main() {  
    int n = 507;  
    if (n = 5)  
        printf("n eh 5.\n");  
    else  
        printf("n nao eh 5.\n");  
  
    return 0;  
}
```

Deveria ser:
if (n == 5)



Estruturas de repetição

- while

```
while (<condicao>) {  
  
}
```

Exemplo

```
int i = 0;  
while (i < 3) {  
    i++;  
    printf("%d\n", i);  
}
```

Saída



Estruturas de repetição

- while

```
while (<condicao>) {  
  
}
```

Exemplo

```
int i = 0;  
while (i < 3) {  
    i++;  
    printf("%d\n", i);  
}
```

Saída

```
1  
2  
3
```

Estruturas de repetição

- while

```
while (<condicao>) {  
  
}
```

Exemplo

```
int i = 0;  
while (i < 3) {  
    printf("%d\n", i++);  
}
```

Saída



Estruturas de repetição

- while

```
while (<condicao>) {  
  
}
```

Exemplo

```
int i = 0;  
while (i < 3) {  
    printf("%d\n", i++);  
}
```

Saída

```
0  
1  
2
```


Estruturas de repetição

- while

```
while (<condicao>) {  
  
}
```

Exemplo

```
int i = 0;  
while (i < 3) {  
    printf("%d\n", ++i);  
}
```

Saída



Estruturas de repetição

- while

```
while (<condicao>) {  
  
}
```

Exemplo

```
int i = 0;  
while (i < 3) {  
    printf("%d\n", ++i);  
}
```

Saída

```
1  
2  
3
```

Estruturas de repetição

- do-while

```
do {  
  
} while (<condicao>);
```

Exemplo

```
int i = 0;  
do {  
    printf("1\n");  
    i++;  
} while (i < 3);
```

Saída



Estruturas de repetição

- do-while

```
do {  
  
} while (<condicao>);
```

Exemplo

```
int i = 0;  
do {  
    printf("1\n");  
    i++;  
} while (i < 3);
```

Saída

```
1  
1  
1
```

Estruturas de repetição

- for

```
for (<inicializacao>; <condicao>; <passo>) {  
  
}
```

Exemplo

```
int i;  
for (I = 0; I < 3; I++) {  
    printf("%d\n", i);  
}
```

Saída



Estruturas de repetição

- for

```
for (<inicializacao>; <condicao>; <passo>) {  
  
}
```

Exemplo

```
int i;  
for (I = 0; I < 3; I++) {  
    printf("%d\n", i);  
}
```

Saída

Não
compila!

Estruturas de repetição

- for

```
for (<inicializacao>; <condicao>; <passo>) {  
  
}
```

Exemplo

```
int i;  
for (i = 0; i < 3; i++) {  
    printf("%d\n", i);  
}
```

Saída



Estruturas de repetição

- for

```
for (<inicializacao>; <condicao>; <passo>) {  
  
}
```

Exemplo

```
int i;  
for (i = 0; i < 3; i++) {  
    printf("%d\n", i);  
}
```

Saída

```
0  
1  
2
```


Importante!

Observe o **estilo de codificação** adotado nos slides:

- Indentação;
- Posição das chaves;
- Nomenclatura de variáveis.

Também use nomes representativos para variáveis!

Evite usar tmp1, tmp2, aux1, aux2, aux50, etc.

Vetores

Vetores

- É um conjunto de variáveis do mesmo tipo:
 - Referenciada por um mesmo identificador;
 - Cada elemento é acessado por meio de um índice.
- **Exemplo:** ler a idade de 10 pessoas e contar quantas estão acima da idade média.
 - Uma alternativa seria criar 10 variáveis;
 - Outra (bem melhor), seria criar um vetor/array de comprimento 10.

Vetores

- Declarar vetor:

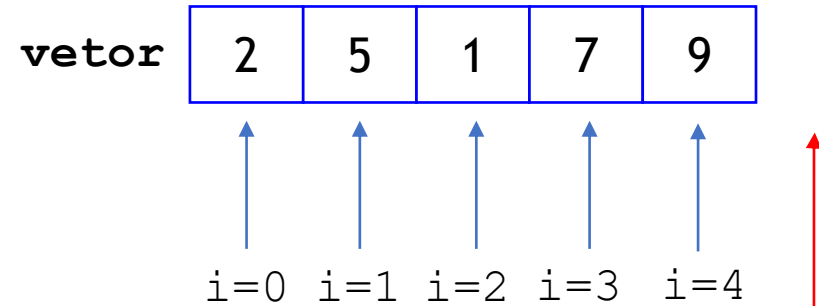
`<tipo> <nome>[<tamanho>];`

Exemplos

```
int idades[10];  
double vetor2[5];  
int valores[3] = {10, 20, 30};
```

Vetores

Índices começam no 0 (zero)



- **Acessar valores em um vetor:**

```
int vetor[5];  
vetor[4] Acessa elemento de índice 4  
vetor[3] = 7; Atribui valor no elemento de índice 3  
if (vetor[3] == 8) Acessa elemento de índice 3  
    return 0;  
vetor[5] Elemento não existe!
```

- **Ler elementos em um vetor:**

```
int idade[10];  
int i;  
for (i = 0; i < 10; i++)  
    scanf("%d", &idade[i]);
```

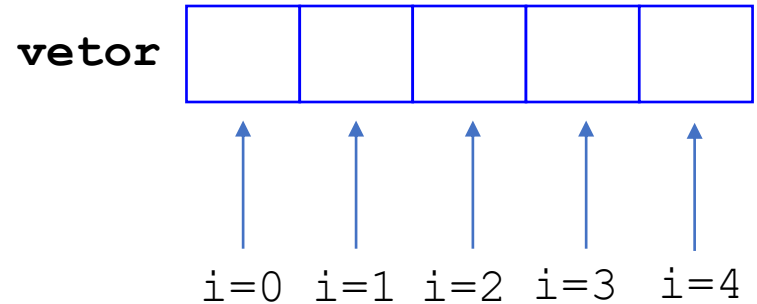
Vetores

- Percorrer um vetor:

```
int vetor[5];  
int i;  
for (i = 0; i < 5; i++) {  
    vetor[i];  
}
```

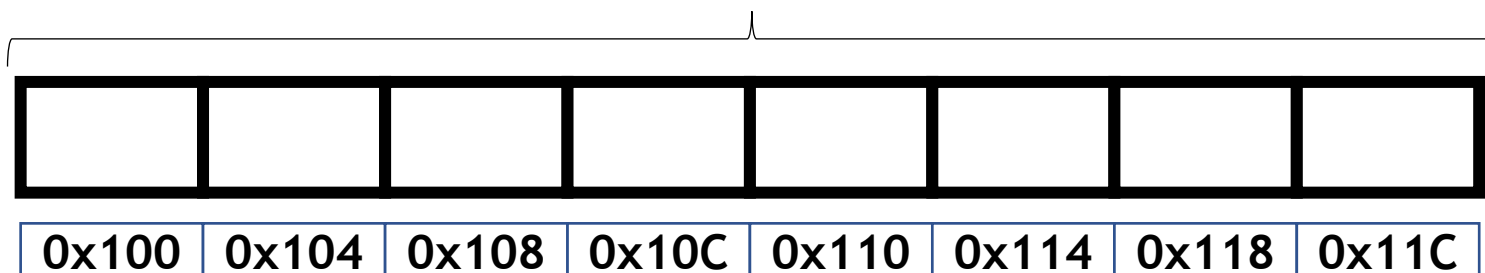
- O que faz este código?

```
int vetor[5];  
int i;  
for (i = 0; i < 5; i++) {  
    vetor[i] = (i+1)*(i+1);  
}
```



Vetores são armazenados em posições consecutivas na memória!

`int vetor[8];`



Endereço de memória do primeiro elemento do vetor.

Observe que cada elemento tem 4 bytes
(tamanho de um int = `sizeof(int)`)

Este vetor ocupa $8 * 4 = 32$ bytes
Ou seja, `qtd_elementos * sizeof(<tipo>)`

Tamanhos dos tipos de dados (sizeof)

- Para saber quantos bytes um tipo de dados ocupa, usamos **sizeof** (o retorno é do tipo **long int**).

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

```
int main() {  
    int a;  
    long int b;  
    long long int c;  
    float d;  
    double e;  
    char f;
```

```
    printf("%ld %ld %ld %ld %ld %ld",  
           sizeof(a),  
           sizeof(b),  
           sizeof(c),  
           sizeof(d),  
           sizeof(e),  
           sizeof(f));
```

```
    return 0;
```

```
}
```


Funções

Funções

- Usadas para **modularizar** o código;
- Uma função executa uma operação e retorna um valor;
- Estrutura de uma função em C:

```
tipo_retorno nome_funcao(Lista de parâmetros) {  
    ...  
}
```

Funções

Estilo de código: nomes de funções são escritos em letras minúsculas e cada palavra é separada por “_”.

- Exemplo

```
int calcula_quadrado(int num) {  
    return num * num;  
}
```

Usado para retornar o valor da função.

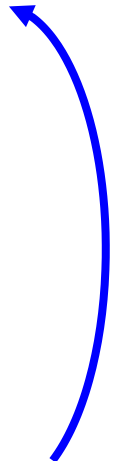
Em C, apenas um valor pode ser retornado.

Chamando uma função

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


```
int calcula_quadrado(int num) {  
    return num * num;  
}
```

```
int main() {  
  
    int n1;  
    scanf("%d", &n1);  
  
    int quad = calcula_quadrado(n1);  
  
    printf("%d\n", quad);  
  
    return 0;  
}
```



Procedimentos

- Um procedimento é uma função que não retorna um valor;
- Usamos **void** para indicar isso.



```
void mostra_quadrado(int num) {  
    printf("%d\n", num * num);  
}
```

Declaração de funções

- Declaramos as funções antes de suas chamadas:
 - Por exemplo, o código da função `calcula_quadrado` está antes da função `main`.

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

```
int calcula_quadrado(int num) {  
    return num * num;  
}
```

```
int main() {  
    int n1;  
    scanf("%d", &n1);  
    int quad = calcula_quadrado(n1);  
    printf("%d\n", quad);  
    return 0;  
}
```

Declaração de funções

- Mas podemos declarar as funções em qualquer ordem se declararmos seus **protótipos** no início:

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

```
int calcula_quadrado(int num);
```



```
int main() {  
    int n1;  
    scanf("%d", &n1);  
    int quad = calcula_quadrado(n1);  
    printf("%d\n", quad);  
    return 0;  
}
```

```
int calcula_quadrado(int num) {  
    return num * num;  
}
```

Passagem de parâmetros

- Em C, todo parâmetro de função é passado **por valor**;
- Para passar um argumento **por referência**, precisamos passar o valor do endereço de memória (ponteiro) - veremos isso em outras aulas...

Parâmetros e argumentos são a mesma coisa?

main é uma função também!

- A função main é chamada quando o programa é iniciado;
- A função retorna um inteiro:
 - Retorna 0 quando o programa termina como esperado;
 - Retorna outro valor em caso de erro!

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

Escopo de variáveis

- O escopo de uma variável determina de onde ela pode ser acessada;
- Em C, existem dois escopos principais:
 - Variáveis **locais**: acessíveis apenas localmente (pela função);
 - Variáveis **globais**: acessíveis a partir de qualquer parte do código.

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

```
double resultado; ←
```

```
void calcula_quadrado(double num) {  
    resultado = num * num;  
}
```

```
double calcula_soma(double n1, double n2) {  
    double r; ←  
    r = n1 + n2;  
    return r;  
}
```

```
int main() {  
    int a = 2, b = 3; ←  
    resultado = calcula_soma(a, b);  
    printf("%.2lf\n", resultado);  
    calcula_quadrado(resultado);  
    printf("%.2lf\n", resultado);  
    calcula_quadrado(resultado);  
    printf("%.2lf\n", resultado);  
  
    return 0;  
}
```

Variável global

A variável resultado pode ser acessada a partir de qualquer função!

Variável local da função
calcula_soma

Variáveis locais da
função main

Variáveis locais são acessíveis apenas da função onde foram declaradas.

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

```
double resultado;
```

```
void calcula_quadrado(double num) {  
    resultado = num * num;  
}
```

```
double calcula_soma(double n1, double n2) {  
    double r;  
    r = n1 + n2;  
    return r;  
}
```

```
int main() {  
    int a = 2, b = 3;  
    resultado = calcula_soma(a, b);  
    printf("%.2lf\n", resultado);  
    calcula_quadrado(resultado);  
    printf("%.2lf\n", resultado);  
    calcula_quadrado(resultado);  
    printf("%.2lf\n", resultado);  
  
    return 0;  
}
```

O que será impresso?

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

```
double resultado;
```

```
void calcula_quadrado(double num) {  
    resultado = num * num;  
}
```

```
double calcula_soma(double n1, double n2) {  
    double r;  
    r = n1 + n2;  
    return r;  
}
```

```
int main() {  
    int a = 2, b = 3;  
    resultado = calcula_soma(a, b);  
    printf("%.2lf\n", resultado);  
    calcula_quadrado(resultado);  
    printf("%.2lf\n", resultado);  
    calcula_quadrado(resultado);  
    printf("%.2lf\n", resultado);  
  
    return 0;  
}
```

O que será impresso?

```
5.00  
25.00  
625.00
```

Veja que o uso de variáveis globais dificulta a leitura do código (e pode levar a erros de programação).

Portanto, evite variáveis globais ao máximo!

Escopo de variáveis

```
#include<stdio.h>

void funcao_teste(int param1) {
    int a = param1;
    if (param1 > 0) {
        int b = 0;
        int i;
        for (i = 0; i < 10; i++) {
            int c = i * i;
            b += c;
            printf("%d %d %d", a, b, c);
        }
        printf("%d %d %d", a, b, c);
    }
    printf("%d %d %d", a, b, c);
}

int main() {
    funcao_teste(507);

    return 0;
}
```

O que será impresso?

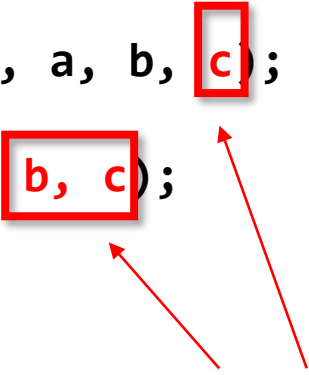
Escopo de variáveis

```
#include<stdio.h>

void funcao_teste(int param1) {
    int a = param1;
    if (param1 > 0) {
        int b = 0;
        int i;
        for (i = 0; i < 10; i++) {
            int c = i * i;
            b += c;
            printf("%d %d %d", a, b, c);
        }
        printf("%d %d %d", a, b, c);
    }
    printf("%d %d %d", a, b, c);
}

int main() {
    funcao_teste(507);

    return 0;
}
```



Erro de compilação!

Escopo de variáveis

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

```
void funcao_teste(int param1) {
```

```
    int a = param1;
```

```
    if (param1 > 0) {
```

```
        int b = 0;
```

```
        int i;
```

```
        for (i = 0; i < 10; i++) {
```

```
            int c = i * i;
```

```
            b += c;
```

```
            printf("%d %d %d", a, b, c);
```

```
        }
```

```
        printf("%d %d %d", a, b, c);
```

```
    }
```

```
    printf("%d %d %d", a, b, c);
```

```
}
```

```
int main() {
```

```
    funcao_teste(507);
```

```
    return 0;
```

```
}
```

Variável c existe apenas aqui!

Escopo de variáveis

```
#include<stdio.h>

void funcao_teste(int param1) {
    int a = param1;
    if (param1 > 0) {
        int b = 0;
        int i;
        for (i = 0; i < 10; i++) {
            int c = i * i;
            b += c;
            printf("%d %d %d", a, b, c);
        }
        printf("%d %d %d", a, b, c);
    }
    printf("%d %d %d", a, b, c);
}

int main() {
    funcao_teste(507);

    return 0;
}
```

Variável b existe apenas aqui!

Escopo de variáveis

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

```
void funcao_teste(int param1) {
```

```
    int a = param1;
```

```
    if (param1 > 0) {
```

```
        int b = 0;
```

```
        int i;
```

```
        for (i = 0; i < 10; i++) {
```

```
            int c = i * i;
```

```
            b += c;
```

```
            printf("%d %d %d", a, b, c);
```

```
        }
```

```
        printf("%d %d %d", a, b, c);
```

```
    }
```

```
    printf("%d %d %d", a, b, c);
```

```
}
```

```
int main() {
```

```
    funcao_teste(507);
```

```
    return 0;
```

```
}
```

Variável a existe apenas aqui!

Passando vetores
como argumento

Variáveis são passadas por valor

```
#include <stdio.h>

void muda_valor(int parametro) {
    parametro = 507;

    printf("%d\n", parametro);
}

int main() {

    int n = 1000;

    muda_valor(n);

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

    return 0;
}
```

Qual a saída
desse programa?

Variáveis são passadas por valor

```
#include <stdio.h>

void muda_valor(int parametro) {
    parametro = 507;

    printf("%d\n", parametro);
}

int main() {

    int n = 1000;

    muda_valor(n);

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

    return 0;
}
```

Qual a saída
desse programa?

507
1000

Ok, variáveis são
passadas por valor!

```
#include <stdio.h>

void muda_valor(int vetor[]) {
    vetor[0] = 90;

    printf("%d\n", vetor[0]);
}

int main() {

    int v[3] = {200, 500, 300};

    muda_valor(v);

    printf("%d %d %d\n", v[0], v[1], v[2]);

    return 0;
}
```

Qual a saída
desse programa?

Mas vetores são passados **por referência!**

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

```
void muda_valor(int vetor[]) {  
    vetor[0] = 90;  
  
    printf("%d\n", vetor[0]);  
}
```

```
int main() {  
  
    int v[3] = {200, 500, 300};  
  
    muda_valor(v);  
  
    printf("%d %d %d\n", v[0], v[1], v[2]);  
  
    return 0;  
}
```

Qual a saída
desse programa?

90

90 500 300

Mas por-que é assim?

Variáveis

```
int matricula = 123;
```

- O identificador de uma variável é usado para acessar seu valor;

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

- O endereço de memória da variável é acessado com o operador address-of &

```
printf("%p\n", &matricula);
```

Vetores

```
int vetor[3] = {20, 500, 7};
```

- O identificador de um vetor representa o **endereço do primeiro elemento!**

```
printf("%p\n", vetor);  
printf("%p\n", &vetor[0]);
```



Retorna o mesmo valor
nos dois casos!

Vetores são passados **por referência!**

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

```
void muda_valor(int vetor[]) {  
    vetor[0] = 90;  
  
    printf("%d\n", vetor[0]);  
}
```

```
int main() {  
  
    int v[3] = {200, 500, 300};  
  
    muda_valor(v);  
  
    printf("%d %d %d\n", v[0], v[1], v[2]);  
  
    return 0;  
}
```

Qual a saída
desse programa?

90

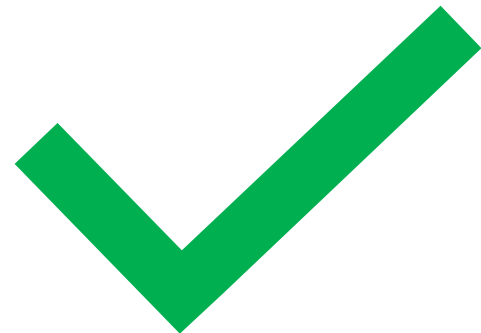
90 500 300

Indentação

Indentação

- Cuidado ao indentar vários blocos de código!!!

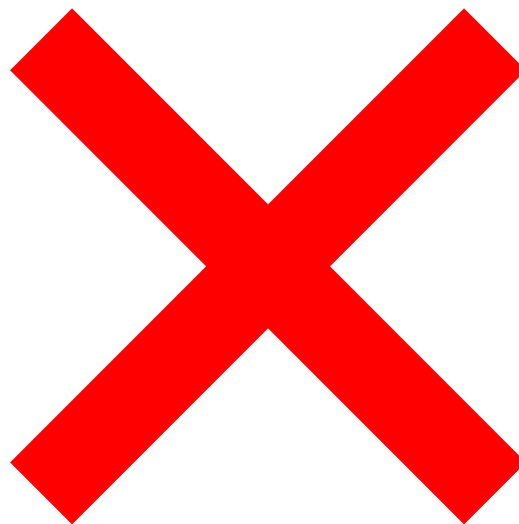
```
void funcao_teste(int param1) {  
→ int a = param1;  
→ if (param1 > 0) {  
→   int b = 0;  
→   int i;  
→   for (i = 0; i < 10; i++) {  
→     int c = i * i;  
→     b += c;  
→     printf("%d %d %d", a, b, c);  
→   }  
→   printf("%d %d %d", a, b, c);  
→ }  
→ printf("%d %d %d", a, b, c);  
}
```



Indentação

- Cuidado ao indentar vários blocos de código!!!

```
void funcao_teste(int param1) {  
    int a = param1;  
    if (param1 > 0) {  
        int b = 0;  
        int i;  
        for (i = 0; i < 10; i++) {  
            int c = i * i;  
            b += c;  
            printf("%d %d %d", a, b, c);  
        }  
        printf("%d %d %d", a, b, c);  
    }  
    printf("%d %d %d", a, b, c);  
}
```




Indentação

- Cuidado ao indentar vários blocos de código!!!

```
void funcao_teste(int param1) {  
    int a = param1;  
    if (param1 > 0) {  
        int b = 0;  
        int i;  
        for (i = 0; i < 10; i++) {  
            int c = i * i;  
            b += c;  
            printf("%d %d %d", a, b, c);  
        }  
        printf("%d %d %d", a, b, c);  
    }  
    printf("%d %d %d", a, b, c);  
}
```

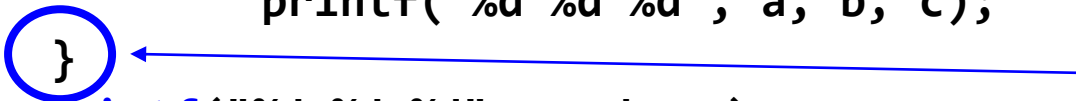
Este printf será chamado
se param1 <= 0?



Indentação

- Cuidado ao indentar vários blocos de código!!!

```
void funcao_teste(int param1) {  
    int a = param1;  
    if (param1 > 0) {  
        int b = 0;  
        int i;  
        for (i = 0; i < 10; i++) {  
            int c = i * i;  
            b += c;  
            printf("%d %d %d", a, b, c);  
        }  
        printf("%d %d %d", a, b, c);  
    }  
}
```



Esta chave aqui dá a impressão que o segundo **printf** está fora do bloco if!

Indentação

- Cuidado ao indentar vários blocos de código!!!

```
void funcao_teste(int param1) {  
    int a = param1;  
    if (param1 > 0) {  
        int b = 0;  
        int i;  
        for (i = 0; i < 10; i++) {  
            int c = i * i;  
            b += c;  
            printf("%d %d %d", a, b, c);  
        }  
        printf("%d %d %d", a, b, c);  
    }  
    printf("%d %d %d", a, b, c);  
}
```




Strings

Strings

- Não há o tipo String em C;
- Para representar uma String em C, usamos um **vetor de char**.

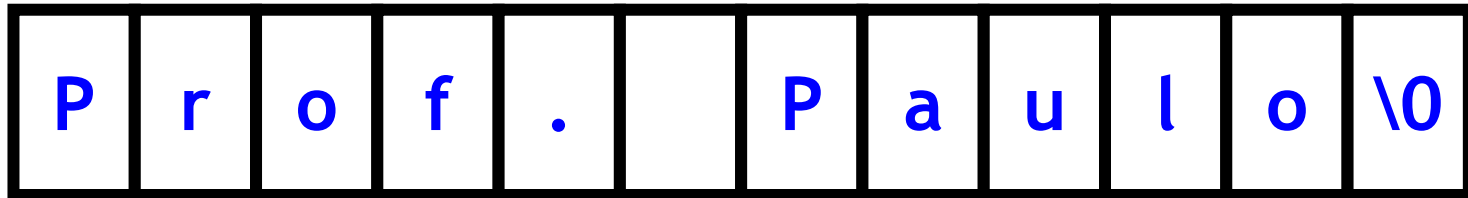
```
#include <stdio.h>
```

```
int main() {  
    char nome[50] = "Prof. Paulo";  
    printf("%s\n", nome);  
  
    return 0;  
}
```



Strings

- Representação da String:



Vamos criar uma string caractere a caractere...

```
#include<stdio.h>

int main() {
    char universidade[50];
    universidade[0] = 'U';
    universidade[1] = 'F';
    universidade[2] = 'A';
    universidade[3] = 'B';
    universidade[4] = 'C';

    printf("%s\n", universidade);

    return 0;
}
```

Veja que usamos aspas simples para representar caracteres!



O que será impresso?

Vamos criar uma string caractere a caractere...

```
#include<stdio.h>

int main() {
    char universidade[50];
    universidade[0] = 'U';
    universidade[1] = 'F';
    universidade[2] = 'A';
    universidade[3] = 'B';
    universidade[4] = 'C';

    printf("%s\n", universidade);

    return 0;
}
```

O que será impresso?

UFABCw \a █ ||k1w¡11wL@



Vamos criar uma string caractere a caractere...

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

```
int main() {  
    char universidade[50];  
    universidade[0] = 'U';  
    universidade[1] = 'F';  
    universidade[2] = 'A';  
    universidade[3] = 'B';  
    universidade[4] = 'C';  
    universidade[5] = '\\0';  
  
    printf("%s\\n", universidade);  
  
    return 0;  
}
```

Faltou colocarmos o caractere
indicando o final da String!



O que será impresso?

Lendo Strings

- Para ler Strings, passamos o vetor para o `scanf`:

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

```
int main() {  
    char universidade[50];  
  
    scanf("%s", universidade);  
    printf("%s\n", universidade);  
  
    return 0;  
}
```

AH! Cadê o “&” no
scanf???



Lendo Strings

- Para ler Strings, passamos o vetor para o **scanf**:

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

```
int main() {  
    char universidade[50];  
  
    scanf("%s", universidade); ←  
    printf("%s\n", universidade);  
  
    return 0;  
}
```

Apesar de ser o scanf, observe que aqui não usamos o “&”



Lendo Strings

- Para ler Strings, passamos o vetor para o `scanf`:

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

```
int main() {  
    char universidade[50];  
  
    scanf("%s", universidade);  
    printf("%s\n", universidade);  
  
    return 0;  
}
```



Isso ocorre, porque o quando usamos o identificador do vetor sem os colchetes, ele representa o **endereço do primeiro elemento.**

Lendo Strings


- Para ler Strings, passamos o vetor para o **scanf**:

```
#include<stdio.h>

int main() {
    char universidade[50];

    scanf("%s", &universidade[0]);
    printf("%s\n", universidade);

    return 0;
}
```



Podemos ler uma string assim também, passando o endereço do primeiro elemento explicitamente.

Endereço do primeiro elemento

- Veja que o mesmo endereço de memória é impresso!

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

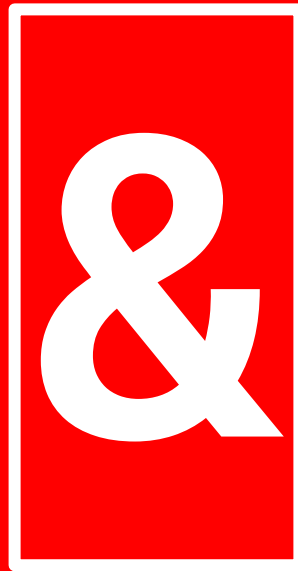
```
int main() {  
    char universidade[50];  
  
    printf("%p\n", universidade);  
  
    printf("%p\n", &universidade[0]);  
  
    return 0;  
}
```

```
int scanf( const char * format, ... );
```

Formato



Endereços
das variáveis



Este é o operador address-of!
Ele retorna o endereço do item a
sua direita!

Por exemplo:

&temp retorna o endereço de temp

&soma retorna o endereço de soma

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

scanf

```
int main() {  
    float numero;  
    scanf("%f", &numero); ✓
```

```
    scanf("%f", numero); ✗
```

```
    char universidade[50];  
    scanf("%s", universidade); ✓
```

O identificador do vetor
representa o endereço
do primeiro elemento!

```
    scanf("%s", &universidade[0]); ✓
```

```
    return 0;
```

```
}
```

gets e puts

- **gets**: lê uma string;
- **puts**: imprime uma string e quebra a linha.

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

```
int main() {  
    char texto[20];
```

```
    gets(texto);
```

```
    puts(texto);
```

```
    return 0;
```

```
}
```


fgets

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

```
int main() {
```

```
    printf("Digite uma frase: ");
```

```
    char frase[6];  
    fgets(frase, 6, stdin);  
    puts(frase);
```

```
    return 0;
```

```
}
```

É recomendável utilizar o **fgets** ao invés do **gets**! o **fgets** limita a quantidade de caracteres lida.

```
Digite uma frase: UFABC  
UFABC
```

Pegou apenas os 5 primeiros caracteres! →

```
Digite uma frase: Universidade  
Unive
```

fgets e puts

- **fgets**: lê uma string;
- **puts**: imprime uma string e quebra a linha.


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

```
int main() {  
    char texto[20];
```

```
    fgets(texto, 20, stdin);  
    puts(texto);
```

```
    return 0;
```


```
}
```



OK, mas qual a
diferença para o
scanf e printf????

fgets e puts

- **fgets**: lê uma string;
- **puts**: imprime uma string.



Vamos ver no exemplo a seguir...


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

```
int main() {  
    char texto[20];
```

```
    fgets(texto, 20, stdin);  
    puts(texto);
```

```
    return 0;
```

```
}
```



OK, mas qual a diferença para o scanf e printf????

Há alguma diferença entre esses dois programas?

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

```
int main() {  
    char texto[20];  
  
    scanf("%s", texto);  
    printf("%s\n", texto);  
  
    return 0;  
}
```

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

```
int main() {  
    char texto[20];  
  
    fgets(texto, 20, stdin);  
    puts(texto);  
  
    return 0;  
}
```

Há alguma diferença entre esses dois programas?

```
#include <stdio.h>

int main() {
    char texto[20];

    scanf("%s", texto);
    printf("%s\n", texto);

    return 0;
}
```

```
#include <stdio.h>

int main() {
    char texto[20];

    fgets(texto, 20, stdin);
    puts(texto);

    return 0;
}
```

Sim! O `scanf("%s", texto)` para de ler a string quando encontra um caractere espaço, mas o `fgets` não!!!

Comprimento de uma String

- **Exercício:** leia uma String e calcule o comprimento da String.

Matrizes

Matrizes

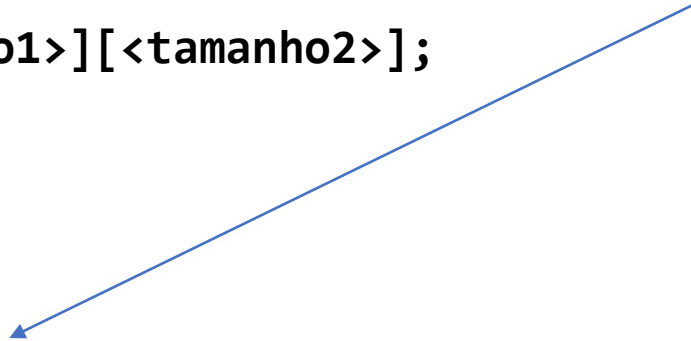
- Declarar matriz:

`<tipo> <nome>[<tamanho1>][<tamanho2>];`

Exemplos

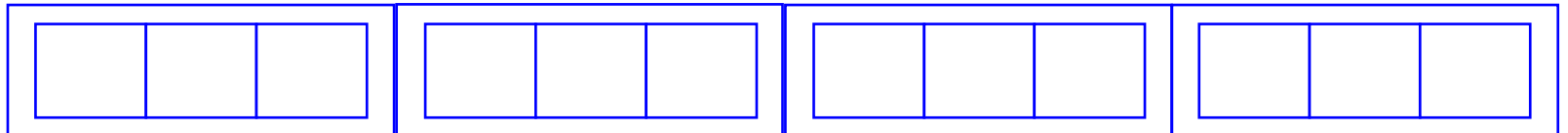
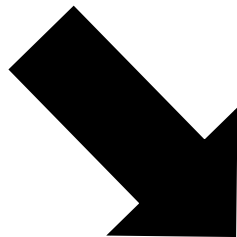
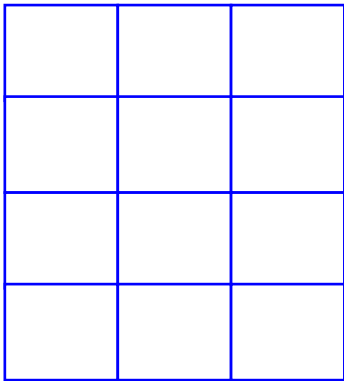
```
int matriz[4][3];
```

```
double matriz2[4][3];
```

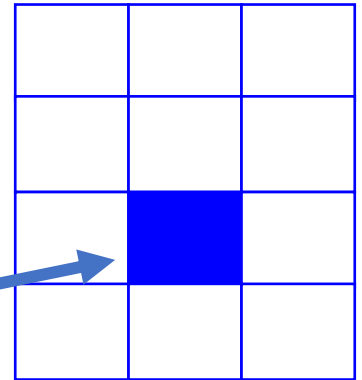


Matriz é um vetor de vetores

- Internamente, a matriz é um vetor unidimensional, em que cada elemento é um vetor unidimensional.



Matrizes



- **Acessar** valores em uma matriz.

`vetor[2][1]`

Índices começam no 0 (zero)

- **Ler/Imprimir** elemento de matriz:

```
int matriz[4][3];
```

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

Matrizes

O que faz esse programa?

```
#include<stdio.h>

int main() {
    float matriz[4][3];
    int i, j, c = 0;

    for (i = 0; i < 4; i++)
        for (j = 0; j < 3; j++)
            matriz[i][j] = c++;

    return 0;
}
```

E este outro?

```
#include<stdio.h>

int main() {
    float matriz[4][3];
    int i, j, c = 0;

    for (i = 0; i < 4; i++)
        for (j = 0; j < 3; j++)
            matriz[i][j] = ++c;

    return 0;
}
```

- Quando o operador ++ está DEPOIS da variável (c++), primeiro ele retorna o valor *e depois incrementa*;
- Quando está ANTES (++c), primeiro incrementa *e depois retorna*.

O que faz esse programa?

```
#include<stdio.h>

int main() {
    float matriz[4][3];
    int i, j, c = 0;

    for (i = 0; i < 4; i++)
        for (j = 0; j < 3; j++)
            matriz[i][j] = c++;

    return 0;
}
```

E este outro?

```
#include<stdio.h>

int main() {
    float matriz[4][3];
    int i, j, c = 0;

    for (i = 0; i < 4; i++)
        for (j = 0; j < 3; j++)
            matriz[i][j] = ++c;

    return 0;
}
```

Passagem de matriz como parâmetro

- Para passar uma matriz como parâmetro, é necessário especificar ao menos a segunda dimensão;
 - Mas é possível especificar as duas dimensões também;

```
void funcao(int n_colunas, double m[][n_colunas])
```

```
void funcao(int n_linhas, int n_colunas, double m[n_linhas][n_colunas])
```

Referências

- Slides do Prof. Fabrício Olivetti:
 - <http://folivetti.github.io/courses/ProgramacaoEstruturada/>
- Slide do Prof. Monael Pinheiro Ribeiro:
 - <https://sites.google.com/site/aed2018q1/>
- CELES, W.; CERQUEIRA, R.; RANGEL, J. L. Introdução a Estruturas de Dados. Elsevier/Campus, 2004.

Bibliografia básica

- CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. Algoritmos: teoria e prática. 2. ed. Rio de Janeiro, RJ: Campus, 2002.
- KNUTH, D. E. The art of computer programming. Upper Saddle River, USA: Addison-Wesley, 2005.
- SEDGEWICK, R. Algorithms in C: parts 1-4 (fundamental algorithms, data structures, sorting, searching). Reading, USA: Addison-Wesley, 1998.

Bibliografia complementar

- DROZDEK, A. Estrutura de dados e algoritmos em C++. São Paulo, SP: Thomson Learning, 2002.
- RODRIGUES, P.; PEREIRA, P.; SOUSA, M. Programação em C++: conceitos básicos e algoritmos. Lisboa, PRT: FCA de Informática, 2000.
- SZWARCFITER, J. L.; MARKENZON, L. Estruturas de dados e seus algoritmos. 3. ed. Rio de Janeiro, RJ: LTC, 2010.
- TENENBAUM, A. M.; LANGSAM Y.; AUGENSTEIN M. J. Estruturas de dados usando C. São Paulo, SP: Pearson Makron Books, 1995.
- ZIVIANI, N. Projeto de algoritmos com implementação em Java e C++. São Paulo, SP: Thomson Learning, 2007.