

# ACH2002

## Aula 2

## Revisão de C

# Aula Passada

- Equilíbrio das várias áreas da vida => sucesso sustentável
- Funcionamento geral da disciplina
- Programação elegante (leitura e documentação)

# Aula de hoje

- Revisão de C (parte 1)

# Noções básicas de C

(embora esse não seja um curso de C)

# Paradigma e Linguagem de Programação

**Paradigma:** programação estruturada NÃO orientada a objetos

**Linguagem** utilizada:

- EPs e aulas: C
- Aula: mais pseudocódigo

**Ferramentas** de desenvolvimento: o que quiser

- Codeblocks
- Dev-C++
- Editor de texto + **compilador gcc** (disponível em qualquer distribuição linux)
- Emacs
- Visual Studio
- Jupyter
- etc

# Pseudocódigo

# C

% comentário

/\* .....\*/ ou //

← % atribuição

=

se <cond> então

if (<cond>)

Se <cond> então... senão

if (<cond>) ... else

enquanto <cond>... faça

while (<cond>)

repita ... até <cond>

do .... while (<cond>)

para... faça

for (<inic>; <parada>;passo)

pare

exit()

...

# Paradigmas de Programação

- Programação **imperativa** x **declarativa**

- **Imperativa**: programa-se o que/COMO deve ser feito

- *Programação linear*: ex: Assembly (necessidade de goto's)
- *Programação **estruturada ou procedural*** (loops e modularização): ex: **C**, Pascal
- *Programação orientada a objetos* (encapsulamento, herança, polimorfismo): ex: C++, Java, Python, Smalltalk, Ruby

- **Declarativa**: declara-se funções, soluções, predicados, axiomas

- *Programação funcional*: ex: LISP, Haskell, Scala
- *Programação lógica*: ex: Prolog

- Vídeo interessante: <https://youtu.be/EefVmQ2wPIM>

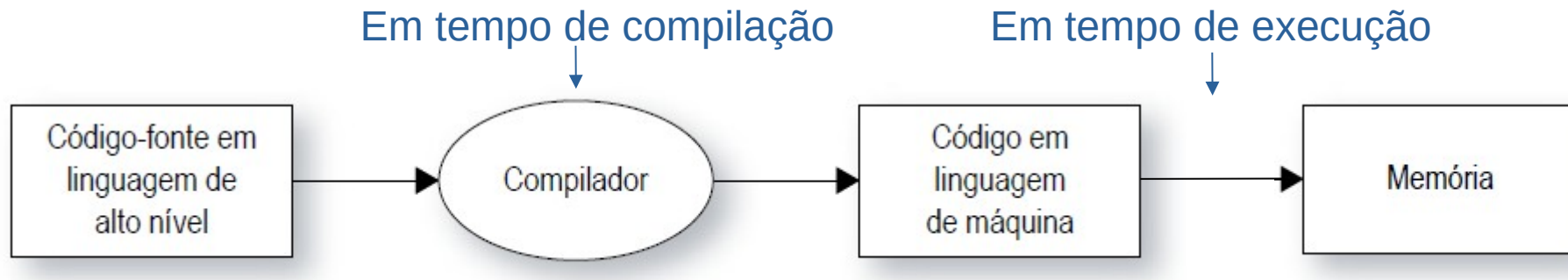
```
10 CLS
20 A = 1
30 PRINT A
40 A = A + 2
50 IF A > 99 THEN END
60 GOTO 30
```

# Origem do C

- Criada em 1972 por Dennis Ritchie, no centro de pesquisas da Bell Laboratories, para reescrever o sistema operacional UNIX (anteriormente escrito em Assembly)
- Isso explica bastante de C:
  - uma evolução e tanto em relação à Assembly (fluxos de controle, operadores, estrutura...)
  - **rápida** (é quase um Assembly ;-)
  - **não tão alto-nível** (bom para entender como as estruturas de dados e os programas funcionam no computador)
  - de propósito geral

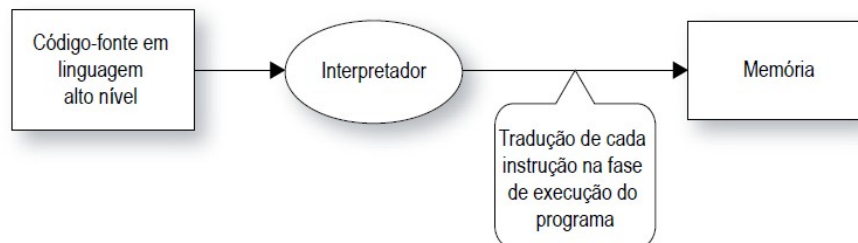


# C é compilável



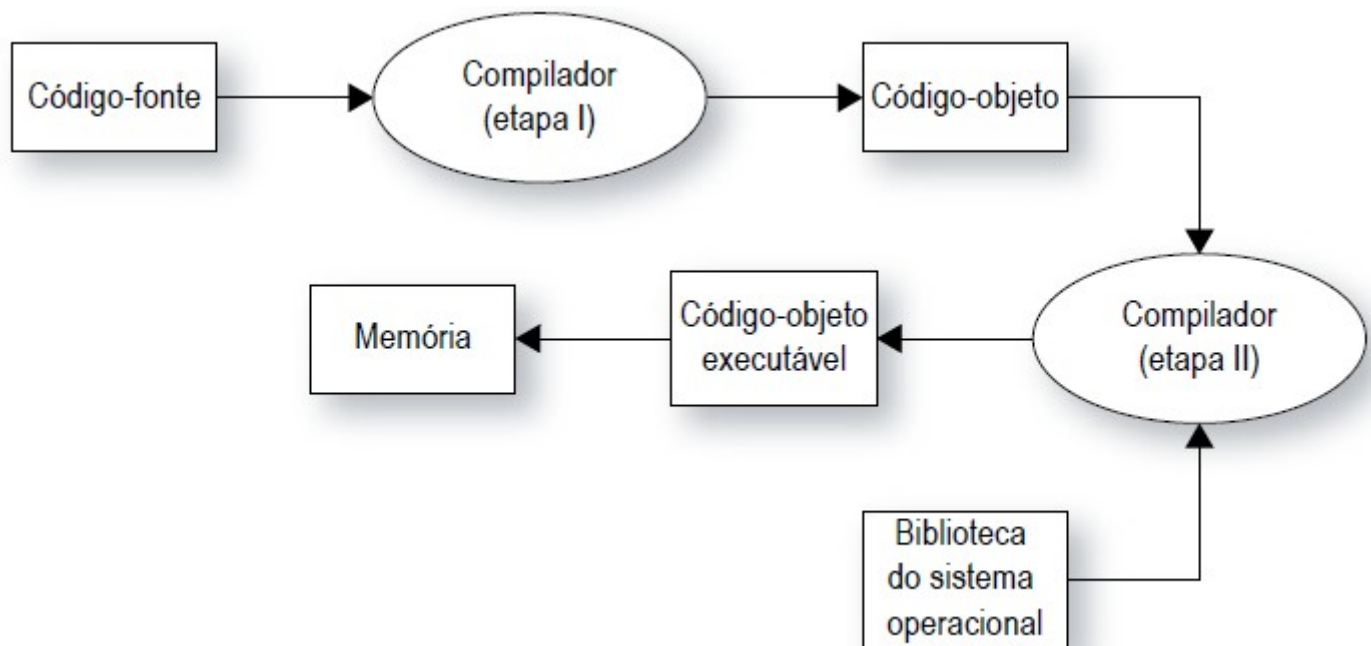
Ex: **gcc** (GNU Compiler Collection)

- Diferente de outras que são interpretáveis (ex: Perl, Python, ...)



# C é compilável

- Código fonte pode ter vários módulos ou fazer chamadas a rotinas do sistema operacional (*system calls*) para não ter que manipular hardware
  - Etapa adicional de ligação (linkagem) dos códigos-objetos em um código executável



# Primeiro programa em C

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

← Necessário para usar o printf

```
int main()
```

← Função principal

```
{
```

```
    printf("Hello World!\n");
```

← \n para pular uma linha (newline)

```
    return 0;
```

← Normalmente associado a  
término sem erros

```
}
```

# Primeiro programa em C

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

← Necessário para usar o printf

```
int main()
```

← **Função principal**

```
{
```

```
    printf("Hello World!\n");
```

← \n para pular uma linha (newline)

```
    return 0;
```

← Normalmente associado a  
término sem erros

```
}
```

OU

```
void main()
```

```
{ ... }
```

OU

```
int main(int argc, const char* argv[] )
```

```
{ ... }
```

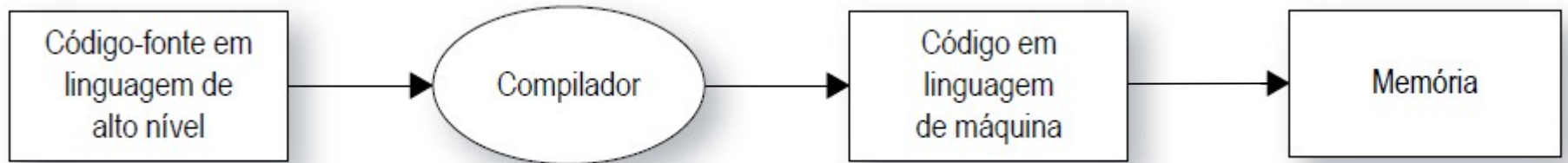
# Compilando esse código, que estaria em um arquivo chamado **hello.c** por exemplo

```
#include <stdio.h>

int main()
{
    printf("Hello World!\n");

    return 0;
}
```

gcc -o hello.exe hello.c



# Compilando com gcc

Para vários módulos:

`gcc -c part1.c` // gera part1.o

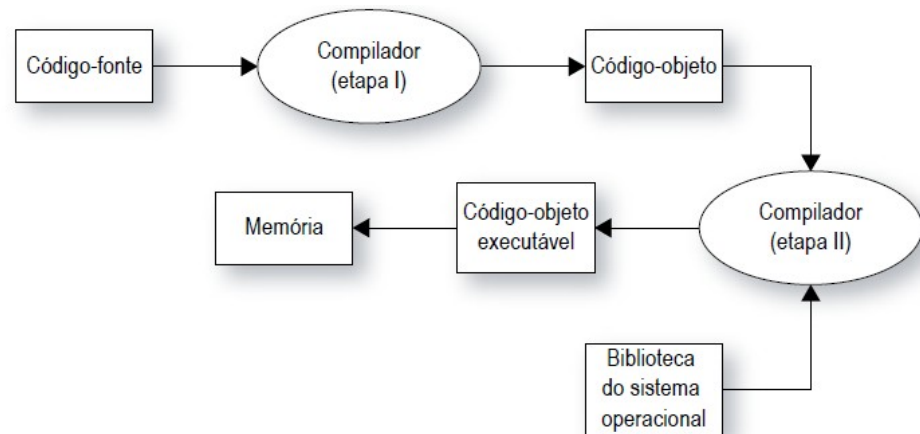
`gcc -c part2.c` // gera part2.o

...

`gcc -c partn.c` // gera partn.o

`gcc -c main.c` // gera main.o

`gcc -o myprogram.exe part1.o part2.o ... partn.o main.o` // faz a linkagem



# Tipos em C

# Tipagem

- C é uma linguagem com *tipagem estática*: o tipo de cada variável é definido no código, no momento de sua declaração  
Ex:  
`int idade;`
- No entanto, é possível alterar essa tipagem (mas ainda no código) via *type casting*:  
Ex:  
`float idadeQuebrada = (float) idade;`
- Outras linguagens são de *tipagem dinâmica*: o tipo depende do conteúdo atribuído em tempo de execução (ex: Python, Smalltalk)

`idade = 18`



# Tipos básicos em C

- Tipos escalares
  - Inteiros: tipicamente `int`
  - Caractere: `char` (mas é um inteiro também)
  - Reais: tipicamente usamos `double`
  - Booleanos
  - Ponteiros
- Tipos não-escalares
  - Estruturas: `struct`
  - Uniões: `union`
  - Arranjos: vetores e matrizes



Tipo void

# Tipos básicos em C

- Tipos escalares
  - **Inteiros**: tipicamente **int**
  - **Caractere**: **char** (mas é um inteiro também)
  - Reais: tipicamente usamos **double**
  - Booleanos
  - Ponteiros
- Tipos não-escalares
  - Estruturas: struct
  - Uniões: union
  - Arranjos: vetores e matrizes

Type	Storage size	Value range
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	8 bytes or (4bytes for 32 bit OS)	-9223372036854775808 to 9223372036854775807
unsigned long	8 bytes	0 to 18446744073709551615

# Tipos básicos em C

Incluído a partir do padrão C99

Para usar:

`#include <stddef.h>`

## ASCII Table

Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char
0	0	0		32	20	40	[space]	64	40	100	@	96	60	140	`
1	1	1		33	21	41	!	65	41	101	A	97	61	141	a
2	2	2		34	22	42	"	66	42	102	B	98	62	142	b
3	3	3		35	23	43	#	67	43	103	C	99	63	143	c
4	4	4		36	24	44	\$	68	44	104	D	100	64	144	d
5	5	5		37	25	45	%	69	45	105	E	101	65	145	e
6	6	6		38	26	46	&	70	46	106	F	102	66	146	f
7	7	7		39	27	47	'	71	47	107	G	103	67	147	g
8	8	10		40	28	50	(	72	48	110	H	104	68	150	h
9	9	11		41	29	51	)	73	49	111	I	105	69	151	i
10	A	12		42	2A	52	*	74	4A	112	J	106	6A	152	j
11	B	13		43	2B	53	+	75	4B	113	K	107	6B	153	k
12	C	14		44	2C	54	,	76	4C	114	L	108	6C	154	l
13	D	15		45	2D	55	-	77	4D	115	M	109	6D	155	m
14	E	16		46	2E	56	.	78	4E	116	N	110	6E	156	n
15	F	17		47	2F	57	/	79	4F	117	O	111	6F	157	o
16	10	20		48	30	60	0	80	50	120	P	112	70	160	p
17	11	21		49	31	61	1	81	51	121	Q	113	71	161	q
18	12	22		50	32	62	2	82	52	122	R	114	72	162	r
19	13	23		51	33	63	3	83	53	123	S	115	73	163	s
20	14	24		52	34	64	4	84	54	124	T	116	74	164	t
21	15	25		53	35	65	5	85	55	125	U	117	75	165	u
22	16	26		54	36	66	6	86	56	126	V	118	76	166	v
23	17	27		55	37	67	7	87	57	127	W	119	77	167	w
24	18	30		56	38	70	8	88	58	130	X	120	78	170	x
25	19	31		57	39	71	9	89	59	131	Y	121	79	171	y
26	1A	32		58	3A	72	:	90	5A	132	Z	122	7A	172	z
27	1B	33		59	3B	73	;	91	5B	133	[	123	7B	173	{
28	1C	34		60	3C	74	<	92	5C	134	\	124	7C	174	
29	1D	35		61	3D	75	=	93	5D	135	]	125	7D	175	}
30	1E	36		62	3E	76	>	94	5E	136	^	126	7E	176	~
31	1F	37		63	3F	77	?	95	5F	137	_	127	7F	177	

## Tipos escalares

- Inteiros: tipicamente `int`
- **Caractere**: `char` (mas é um inteiro também) e `wchar_t` (2 ou 4 bytes)
- Reais: tipicamente usamos `double`
- Booleanos
- Ponteiros

## Tipos não-escalares

- Estruturas: `struct`
- Uniões: `union`
- Arranjos: vetores e matrizes

## Tipo void

# Tipos básicos em C

- Tipos escalares
  - Inteiros: tipicamente `int`
  - Caractere: `char` (mas é um inteiro também)
  - **Reais**: tipicamente usamos `double`
  - Booleanos
  - Ponteiros
- Tipos não-escalares
  - Estruturas: `struct`
  - Uniões: `union`
  - Arranjos: vetores e matrizes

Type	Storage size	Value range	Precision
float	4 byte	1.2E-38 to 3.4E+38	6 decimal places
double	8 byte	2.3E-308 to 1.7E+308	15 decimal places
long double	10 byte	3.4E-4932 to 1.1E+4932	19 decimal places

Tipo void

# Tipos básicos em C

- Tipos escalares
  - Inteiros: tipicamente `int`
  - Caractere: `char` (mas é um inteiro também)
  - Reais: tipicamente usamos `double`
  - **Booleanos**: `bool`
  - Ponteiros
- Tipos não-escalares
  - Estruturas: `struct`
  - Uniões: `union`
  - Arranjos: vetores e matrizes

Incluído a partir do padrão C99

Para usar:

```
#include <stdbool.h>
```

Tipo void

# Tipos básicos em C

```
1  #include<stdio.h>
2  #include<stdbool.h>
3  int main()
4  {
5      bool x = 10 > 18; /* false, que significa 0 se imprimir como inteiro*/
6      printf("%d ou ",x);
7      printf("%s", x ? "true" : "false");
8      bool y = true;
9      printf("\n%d ou ",y);
10     printf("%s", y ? "true" : "false");
11     printf("\nTamanho do bool em bytes é %d\n",sizeof(bool));
12     return 0;
13 }
```

# Tipos básicos em C

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

int main()
{
    bool x = 10 > 18; /* false, que significa 0 se imprimir como inteiro*/
    printf("%d ou ",x);
    printf("%s", x ? "true" : "false");
    bool y = true;
    printf("\n%d ou ",y);
    printf("%s", y ? "true" : "false");
    printf("\nTamanho do bool em bytes é %ld\n",sizeof(bool));
    return 0;
}
```

```
(base) ariane@rainbow:~/ACH2002-2022-2/codigos/introducao$ ./bool
0 ou false
1 ou true
Tamanho do bool em bytes é 1
```

# Tipos básicos em C

- Tipos escalares
  - Inteiros: tipicamente `int`
  - Caractere: `char` (mas é um inteiro também)
  - Reais: tipicamente usamos `double`
  - Booleanos: `bool`
  - **Ponteiros**: endereço em memória
- Tipos não-escalares
  - Estruturas: `struct`
  - Uniões: `union`
  - Arranjos: vetores e matrizes

Existem dois espaços em memória:

- um que tem o inteiro da variável `idade`
- outro que tem o endereço de memória onde está esse `int` (isso ficará mais claro nos slides 56 a 85)

Ex:  
`int* idade;`  
`float* temperatura;`  
`etc...`

 Tipo void



# Tipos básicos em C

- Tipos escalares

- Inteiros: tipicamente **int**
- Caractere: **char** (mas é um inteiro também)
- Reais: tipicamente usamos **double**
- Booleanos: **bool**
- Ponteiros

- Tipos não-escalares

- Estruturas (ou registros): **struct**
- Uniões: **union**
- Arranjos: vetores e matrizes

```
struct {  
  
    <tipo do campo 1> <campo 1>;  
  
    <tipo do campo 2> <campo 2>;  
  
    ...  
  
    <tipo do campo n> <campo n>;  
  
} <identificador da variável>;
```

Uso: <nome da variável desse tipo>.<campo>

# Struct - exemplo

```
#include<stdio.h>
int main()
{
    struct {
        long nrUSP;
        float notaEP1;
        //float notaEP2;
        float mediaProvinhas;
    } aluno;

    aluno.nrUSP = 555555;
    aluno.notaEP1 = 9.9;
    printf("\nNota do aluno %ld é %f\n", aluno.nrUSP, aluno.notaEP1);

    printf("\nTamanho em bytes do long é %ld\n",sizeof(long));
    printf("\nTamanho em bytes do float é %ld\n",sizeof(float));
    printf("\nTamanho em bytes da estrutura é %ld\n",sizeof(aluno));

    return 0;
}
```

```

#include<stdio.h>
int main()
{
    struct {
        long nrUSP;
        float notaEP1;
        //float notaEP2;
        float mediaProvinhas;
    } aluno;

    aluno.nrUSP = 555555;
    aluno.notaEP1 = 9.9;
    printf("\nNota do aluno %ld é %f\n", aluno.nrUSP, aluno.notaEP1);

    printf("\nTamanho em bytes do long é %ld\n", sizeof(long));
    printf("\nTamanho em bytes do float é %ld\n", sizeof(float));
    printf("\nTamanho em bytes da estrutura é %ld\n", sizeof(aluno));

    return 0;
}

```

`/introducao$ ./struct`

Nota do aluno 555555 é 9.900000

Tamanho em bytes do long é 8

Tamanho em bytes do float é 4

Tamanho em bytes da estrutura é 16

```

#include<stdio.h>
int main()
{
    struct {
        long nrUSP;
        float notaEP1;
        //float notaEP2;
        float mediaProvinhas;
    } aluno;

    aluno.nrUSP = 555555;
    aluno.notaEP1 = 9.9;
    printf("\nNota do aluno %ld é %f\n", aluno.nrUSP, aluno.notaEP1);

    printf("\nTamanho em bytes do long é %ld\n", sizeof(long));
    printf("\nTamanho em bytes do float é %ld\n", sizeof(float));
    printf("\nTamanho em bytes da estrutura é %ld\n", sizeof(aluno));

    return 0;
}

```

## Tamanho da struct:

Menor nr de palavras em que caibam todos os campos

```
introducao$ ./struct
```

```
Nota do aluno 555555 é 9.900000
```

```
Tamanho em bytes do long é 8
```

```
Tamanho em bytes do float é 4
```

```
Tamanho em bytes da estrutura é 16
```

# Struct - exemplo

```
#include<stdio.h>
int main()
{
    struct {
        long nrUSP;
        float notaEP1;
        //float notaEP2;
        float mediaProvinhas;
    } aluno1;

    aluno1.nrUSP = 555555;
    aluno1.notaEP1 = 9.9;
    printf("\nNota do aluno %ld é %f\n", aluno1.nrUSP, aluno1.notaEP1);

    printf("\nTamanho em bytes do long é %ld\n",sizeof(long));
    printf("\nTamanho em bytes do float é %ld\n",sizeof(float));
    printf("\nTamanho em bytes da estrutura é %ld\n",sizeof(aluno1));

    return 0;
}
```

# Struct não anônima

```
1  #include<stdio.h>
2  int main()
3  {
4      struct aluno{
5          long nrUSP;
6          float notaEP1;
7          float notaEP2;
8          float mediaProvinhas;
9      };
10
11     struct aluno aluno1, aluno2;
12
13     aluno1.nrUSP = 555555;
14     aluno1.notaEP1 = 9.9;
15     printf("\nNota do aluno %ld é %.1f\n", aluno1.nrUSP, aluno1.notaEP1);
16
17     aluno2.nrUSP = 6666666;
18     aluno2.notaEP1 = 8.88;
19     printf("\nNota do aluno %ld é %.1f\n", aluno2.nrUSP, aluno2.notaEP1);
20
21     printf("\nTamanho em bytes do long é %ld\n",sizeof(long));
22     printf("\nTamanho em bytes do float é %ld\n",sizeof(float));
23     printf("\nTamanho em bytes da estrutura é %ld\n",sizeof(struct aluno));
24
25     return 0;
26 }
```



# Typedef

```
1  #include<stdio.h>
2  int main()
3  {
4      typedef struct {
5          long nrUSP;
6          float notaEP1;
7          float notaEP2;
8          float mediaProvinhas;
9      } Aluno;
10
11     Aluno aluno1, aluno2, aluno3;
12
13     aluno1.nrUSP = 555555;
14     aluno1.notaEP1 = 9.9;
15     printf("\nNota do aluno %ld é %.1f\n", aluno1.nrUSP, aluno1.notaEP1);
16
17     printf("\nTamanho em bytes do long é %ld\n",sizeof(long));
18     printf("\nTamanho em bytes do float é %ld\n",sizeof(float));
19     printf("\nTamanho em bytes da estrutura é %ld\n",sizeof(Aluno));
20
21     return 0;
22 }
```

# Estrutura em C

Tipagem:

```
typedef struct {  
    <tipo do campo 1> <campo 1>;  
    <tipo do campo 2> <campo 2>;  
    ...  
    <tipo do campo n> <campo n>;  
} <identificador do tipo>;
```



# Tipos básicos em C

- Tipos escalares
  - Inteiros: tipicamente `int`
  - Caractere: `char` (mas é um inteiro também)
  - Reais: tipicamente usamos `double`
  - Booleanos: `bool`
  - Ponteiros
- Tipos não-escalares
  - Estruturas: `struct`
  - **Uniões**: `union`
  - Arranjos: vetores e matrizes

Tipo void

# Union

```
1  #include<stdio.h>
2  int main()
3  {
4      union pesoVolume {
5          double volume; /* medido em litros */
6          int peso;      /* medido em gramas */
7      };
8
9      union pesoVolume pv;
10
11     pv.volume = 0.5;
12     pv.peso = 200;
13     printf("\nVolume: %.2f, peso: %d\n", pv.volume, pv.peso);
14
15     pv.peso = 800;
16     pv.volume = 0.2;
17     printf("\nVolume: %.2f, peso: %d\n", pv.volume, pv.peso);
18
19     printf("\nTamanho em bytes do float é %ld\n", sizeof(double));
20     printf("\nTamanho em bytes do int é %ld\n", sizeof(int));
21     printf("\nTamanho em bytes da union é %ld\n", sizeof(union pesoVolume));
22
23     return 0;
24 }
```

- Semelhante à struct, mas somente um campo é o válido de cada vez
- Economiza espaço quando os campos são mutualmente exclusivos
- Útil por conta da tipagem estática de C

# Tipos básicos em C

- Tipos escalares

- Inteiros: tipicamente `int`
- Caractere: `char` (mas é um inteiro também)
- Reais: tipicamente usamos `double`
- Booleanos: `bool`
- Ponteiros

- Tipos não-escalares

- Estruturas: `struct`
- Uniões: `union`
- **Arranjos**: vetores e matrizes

Conjuntos homogêneos – todos os itens são do mesmo tipo

**Tipo** nomeVariavel[nr de itens];

Ex:

```
char nome[50]; /* string */  
float temperaturasDoMes[31];  
Aluno alunosACH2023[66];
```



Tipo void

# Arranjos (vetores e matrizes)

Tipo nomeVariavel[nr de itens];

Ex:

```
char nome[30] = "Ariane Machado Lima";  
printf("Nome: %s\n", nome);
```

float temperatura  
ou  
float temperatura  
temperaturasDaSemana[6] = temp30;  
temperaturasDaSemana[hoje] = 30.1;  
temperaturasDaSemana[7] = 23; ERRO !!!

Em C, strings são vetores de char que terminam com '\0' (invisível).

Há funções de manipulação de strings na biblioteca string.h  
(copiar, comparar, concatenar, ...)

```
Aluno alunosACH2023[66];  
alunosACH2023[0].nrUSP = 555555;
```

...  
Profª. Ariane Machado Lima

# Arranjos (vetores e matrizes)

Tipo nomeVariavel[nr de itens];

Ex:

```
char nome[30] = "Ariane Machado Lima";  
printf("Nome: %s\n", nome);
```

```
float temperaturasDaSemana[7] = {20, 20.4, 20.6, 19, 17.8, 18.9, 24.1};  
ou
```

```
float temperaturasDaSemana[7];  
temperaturasDaSemana[0] = 28.5;  
temperaturasDaSemana[6] = temp30;          /* temp30 sendo uma variável float */  
temperaturasDaSemana[hoje] = 30.1;         /* hoje sendo uma variável inteira  
                                           com valor de 0 a 6) */
```

**temperaturasDaSemana[7] = 23;    ERRO !!!**

# Arranjos (vetores e matrizes)

Matrizes: arranjos multidimensionais

Tipo nomeVariavel[*nr de itens*];

Tipo nomeVariavel[*nr de itens dim 1*][*nr de itens dim 2*];

...

Tipo nomeVariavel[*nr de itens dim 1*][*nr de itens dim 2*] ... [*nr de itens dim n*];

Ex:

float notasTrabalhosACH2002[66][3];

NotasTrabalhosACH2002[12][0] = 9.4;

notasTrabalhosACH2002[i][j] = nota;      /\* i, j variáveis int, nota variável float \*/

# Tipos básicos em C

- Tipos escalares
  - Inteiros: tipicamente `int`
  - Caractere: `char` (mas é um inteiro também)
  - Reais: tipicamente usamos `double`
  - Booleanos: `bool`
  - Ponteiros
- Tipos não-escalares
  - Estruturas: `struct`
  - Uniões: `union`
  - Arranjos: vetores e matrizes

**Nada ou tipo desconhecido**  
**Veremos mais adiante**

Tipo **void**



# Enum

Constantes simbólicas para inteiros

```
typedef enum {DOMINGO, SEGUNDA, TERCA, QUARTA, QUINTA, SEXTA,  
SABADO} DIAS_DA_SEMANA;
```

```
typedef enum {DOMINGO=1, SEGUNDA, TERCA, QUARTA, QUINTA, SEXTA,  
SABADO} DIAS_DA_SEMANA;
```

```
typedef enum {DOMINGO=7, SEGUNDA=1, TERCA=3, QUARTA=55, QUINTA=2,  
SEXTA=100, SABADO=80} DIAS_DA_SEMANA;
```



# Funções

- Forma mais comum de modularização
- Em algumas linguagens, como Pascal:
  - **Procedimento**: executa tarefas e não retorna um valor
  - **Função**: retorna um valor
- Comumente tratados como sinônimos
- Ambos podem possuir parâmetros

# Função em C

O retorno é feito assim que é executado o comando **return**, e o valor retornado é o da variável argumento desse comando

# Função – Ex: funcoes.c

```
int calculaIdadeAproximada(Pessoa p)
{
    int idade, year;

    time_t now; // `time_t` é um tipo para tempo (data e hora)
    time(&now); // Obtem tempo corrente

    // localtime converte um `time_t` para um `tm` de tempo de calendario
    struct tm *local = localtime(&now);
    year = local->tm_year + 1900;    // pega ano desde 1900

    // calcula a idade em anos
    idade = year - p.dataNascimento.ano;
    return (idade);
}
```

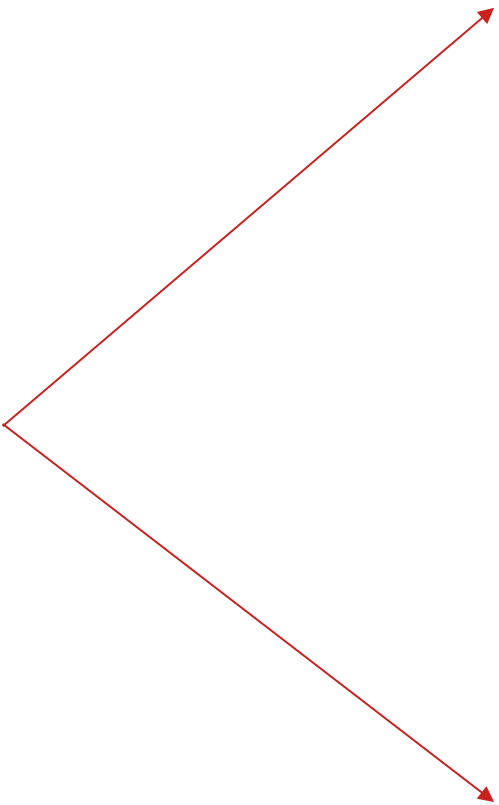
# Ordem de definição das funções

Depende da linguagem de programação (característica do compilador)

Na maioria (inclusive C), primeiro as funções que não fazem chamadas a funções ainda não definidas, por último o módulo principal

Alternativa de C: definir antes o **protótipo** da função (sua interface: tipo de retorno, nome, lista de parâmetros)

# Ex: funcoesComPrototipo.c



```
int calculaIdadeAproximada(Pessoa p);

int main()
{

    Pessoa p1;
    p1.dataNascimento.dia = 27;
    p1.dataNascimento.mes = 9;
    p1.dataNascimento.ano = 2000;

    int idadeP1 = calculaIdadeAproximada(p1);

    printf("Idade: %d\n", idadeP1);

    return 0;
}

int calculaIdadeAproximada(Pessoa p)
{
    int idade, anoAtual;
```

# Escopo de variáveis parte 1 (variáveis locais) - Passagem de parâmetro por valor x referência

# O que esse código imprime?(passagemParametros.c)

```
void alteraAnoNascimento(Pessoa p, int novoAno)
{
    p.dataNascimento.ano = novoAno;
    return;
}

int main()
{
    Pessoa p1;
    p1.dataNascimento.dia = 27;
    p1.dataNascimento.mes = 9;
    p1.dataNascimento.ano = 2000;

    int idadeP1 = calculaIdadeAproximada(p1);
    printf("Idade: %d\n", idadeP1);

    alteraAnoNascimento(p1, 1990);
    idadeP1 = calculaIdadeAproximada(p1);
    printf("Idade: %d\n", idadeP1);

    return 0;
}
```

```
typedef struct{
    int dia;
    int mes;
    int ano;
} Data;

typedef struct{
    char nome[50];
    Data dataNascimento;
    // .... outros campos
} Pessoa;
```

```
int calculaIdadeAproximada(Pessoa p)
{
    int idade;
    Data hoje = dataDeHoje();
    idade = hoje.ano - p.dataNascimento.ano;
    return (idade);
}
```

# O que esse código imprime?(passagemParametros.c)

```
void alteraAnoNascimento(Pessoa p, int novoAno)
{
    p.dataNascimento.ano = novoAno;
    return;
}

int main()
{
    Pessoa p1;
    p1.dataNascimento.dia = 27;
    p1.dataNascimento.mes = 9;
    p1.dataNascimento.ano = 2000;

    int idadeP1 = calculaIdadeAproximada(p1);
    printf("Idade: %d\n", idadeP1);

    alteraAnoNascimento(p1, 1990);
    idadeP1 = calculaIdadeAproximada(p1);
    printf("Idade: %d\n", idadeP1);

    return 0;
}
```

```
typedef struct{
    int dia;
    int mes;
    int ano;
} Data;

typedef struct{
    char nome[50];
    Data dataNascimento;
    // .... outros campos
} Pessoa;
```

```
int calculaIdadeAproximada(Pessoa p)
{
    int idade;
    Data hoje = dataDeHoje();
    idade = hoje.ano - p.dataNascimento.ano;
    return (idade);
}
```

```
ariane@ariane-053525:~/ACH2002-2023-2/codigos/introducao$ ./passagemParametros
Idade: 23
Idade: 23
```



# Escopo de variáveis

**Escopo de uma variável:** parte do programa dentro da qual a variável pode ser usada (a partir do momento em que é definida até o final de seu escopo)

## Variáveis locais

Declaradas dentro da função a que pertencem (normalmente no início),  
**incluindo parâmetros**

Somente podem acessadas dentro da função em que foram declaradas  
("tempo de vida" limitado à execução do módulo)

## Variáveis globais

Declaradas antes de todas as funções, inclusive da principal

Acessíveis (compartilhadas) por todas as funções A PARTIR DO PONTO  
EM QUE É DECLARADA ATÉ O FINAL DO ARQUIVO

# O que aconteceu aqui?????????

```
void alteraAnoNascimento(Pessoa p, int novoAno)
{
    p.dataNascimento.ano = novoAno;
    return;
}

int main()
{
    Pessoa p1;
    p1.dataNascimento.dia = 27;
    p1.dataNascimento.mes = 9;
    p1.dataNascimento.ano = 2000;

    int idadeP1 = calculaIdadeAproximada(p1);
    printf("Idade: %d\n", idadeP1);

    alteraAnoNascimento(p1, 1990);
    idadeP1 = calculaIdadeAproximada(p1);
    printf("Idade: %d\n", idadeP1);

    return 0;
}
```

```
typedef struct{
    int dia;
    int mes;
    int ano;
} Data;

typedef struct{
    char nome[50];
    Data dataNascimento;
    // .... outros campos
} Pessoa;
```

```
int calculaIdadeAproximada(Pessoa p)
{
    int idade;
    Data hoje = dataDeHoje();
    idade = hoje.ano - p.dataNascimento.ano;
    return (idade);
}
```

```
ariane@ariane-053525:~/ACH2002-2023-2/codigos/introducao$ ./passagemParametros
Idade: 23
Idade: 23
```

# Abre parêntesis...

## Passagem por valor x passagem por referência

# Variáveis em C

## ► Importante:

- Dizemos que **a** e **b** armazenam diretamente o seu conteúdo

`int a;`

0000	ocupado anteriormente
0001	ocupado anteriormente
0002	
0003	
0004	
0005	
0006	ocupado anteriormente
0007	
0008	
0009	
0010	
0011	
...	
F....FF	

Espaço reservado para  
variável a (4 bytes)

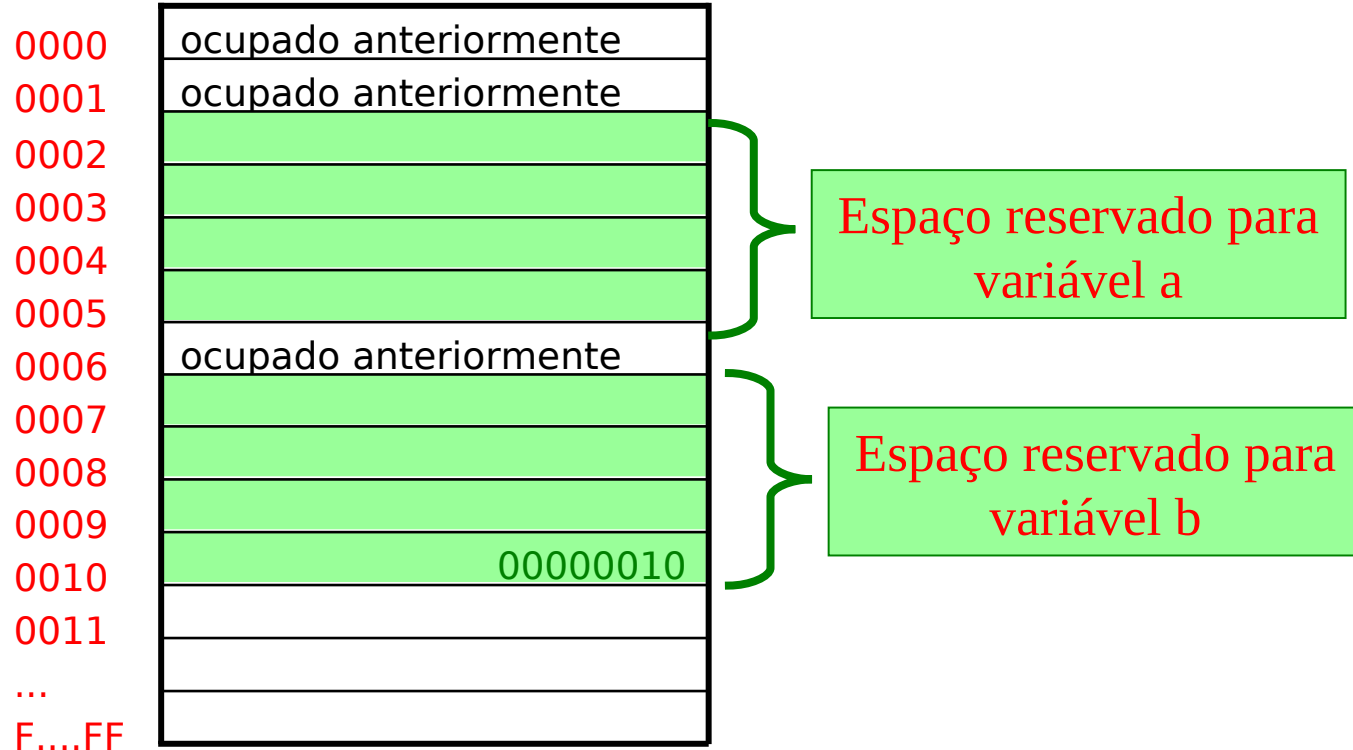
# Variáveis em C

## ► Importante:

- Dizemos que **a** e **b** armazenam diretamente o seu conteúdo

```
int a;
```

```
int b = 2;
```



# Variáveis em C

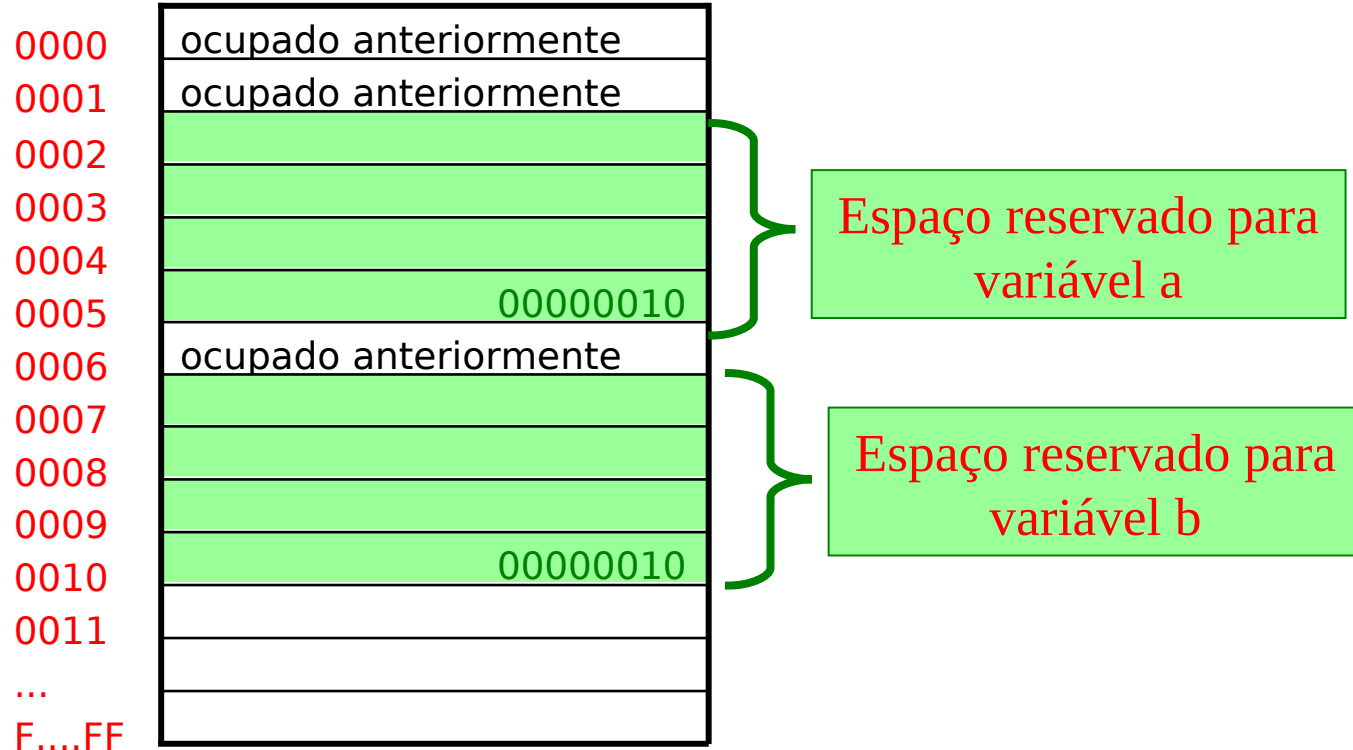
## ► Importante:

- Dizemos que **a** e **b** armazenam diretamente o seu conteúdo

```
int a;
```

```
int b = 2;
```

```
a = b;
```



# Variáveis em C

```
void fazAlgo (int x)
{
    (...)
    x++;
    (...)
}
```

```
int y = 2;
```

```
fazAlgo(y);
```

0000	ocupado anteriormente
0001	ocupado anteriormente
0002	
0003	
0004	
0005	00000010
0006	ocupado anteriormente
0007	
0008	
0009	
0010	
0011	
...	
F....FF	

Espaço reservado para  
variável y

# Variáveis em C

```
void fazAlgo (int x)
{
    (...)
    x++;
    (...)
}
```

```
int y = 2;
```

```
fazAlgo(y);
```

0000	ocupado anteriormente
0001	ocupado anteriormente
0002	
0003	
0004	
0005	00000010
0006	ocupado anteriormente
0007	
0008	
0009	
0010	00000010
0011	
...	
F....FF	

Espaço reservado para  
variável y

Espaço reservado para  
variável x



# Variáveis em C

```
void fazAlgo (int x)
{
    (...)
    x++;
    (...)
}
```

```
int y = 2;
```

```
fazAlgo(y);
```

0000	ocupado anteriormente
0001	ocupado anteriormente
0002	
0003	
0004	
0005	00000010
0006	ocupado anteriormente
0007	
0008	
0009	
0010	00000011
0011	
...	
F....FF	

Espaço reservado para  
variável y

Espaço reservado para  
variável x

# Variáveis em C

```
void fazAlgo (int x)
{
    (...)
    x++;
    (...)
}
```

```
int y = 2;
```

```
fazAlgo(y);
```

0000	ocupado anteriormente
0001	ocupado anteriormente
0002	
0003	
0004	
0005	00000010
0006	ocupado anteriormente
0007	
0008	
0009	
0010	
0011	
...	
F....FF	

Espaço reservado para  
variável y

# Variáveis em C

- **Por isso:** dizemos que a mudança em um parâmetro dentro de um método não se reflete fora dele: *variáveis locais*

```
void fazAlgo (int x)
{
    (...)
    x++;
    (...)
}
```

```
int y = 2;
```

```
fazAlgo(y);
```

0000	ocupado anteriormente
0001	ocupado anteriormente
0002	
0003	
0004	
0005	00000010
0006	ocupado anteriormente
0007	
0008	
0009	
0010	
0011	
...	
F....FF	

Espaço reservado para  
variável y

# Variáveis em C

- **Por isso:** dizemos que a mudança em um parâmetro dentro de um método não se reflete fora dele: *variáveis locais*

```
void fazAlgo (int x)
{
    (...)
    x++;
    (...)
}
```

```
int y = 2;
```

```
fazAlgo(y);
```

0000	ocupado anteriormente
0001	ocupado anteriormente
0002	
0003	
0004	
0005	00000010
0006	ocupado anteriormente
0007	
0008	
0009	
0010	
0011	
...	
F....FF	

Espaço reservado para  
variável y

Nada mudaria se o  
parâmetro também  
fosse y ...

# Variáveis em C

- ▶ **Passagem de parâmetro por VALOR:** o argumento é avaliado e COPIADO para o parâmetro

```
void fazAlgo (int x)
{
    (...)
    x++;
    (...)
}
```

```
int y = 2;
```

```
fazAlgo(y);
```

0000	ocupado anteriormente
0001	ocupado anteriormente
0002	
0003	
0004	
0005	00000010
0006	ocupado anteriormente
0007	
0008	
0009	
0010	
0011	
...	
F....FF	

Espaço reservado  
para variável y

# Variáveis em C

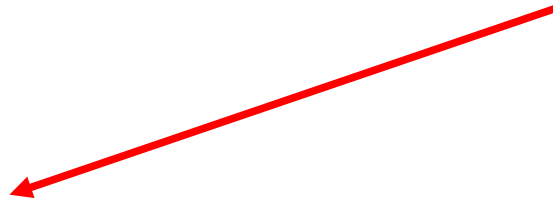
- ▶ **Passagem de parâmetro por VALOR:** o argumento é avaliado e COPIADO para o parâmetro

```
void fazAlgo (int x)
{
    (...)
    x++;
    (...)
}
```

```
int y = 2;
```

```
fazAlgo((y * 4) % 3);
```

Avaliado!!!!



# NÃO em C, nem em Java!!!! - Ex: C++ SIM

- ▶ **Passagem de parâmetro por REFERÊNCIA:** o parâmetro é um apelido (“atalho”) para o argumento (os dois apontam para o mesmo endereço em memória!!!)

# NÃO em C, nem em Java!!!! - Ex: C++ SIM

- **Passagem de parâmetro por REFERÊNCIA:** o parâmetro é um apelido (“atalho”) para o argumento (os dois apontam para o mesmo endereço em memória!!!)

```
void fazAlgo (int &x)
{
    (...)
    x++;
    (...)
}
```

```
int y = 2;
```

```
fazAlgo(y);
```

0000	ocupado anteriormente
0001	ocupado anteriormente
0002	
0003	
0004	
0005	00000010
0006	ocupado anteriormente
0007	
0008	
0009	
0010	
0011	
...	
F....FF	

Espaço reservado  
para variável y



# NÃO em C, nem em Java!!!!!! - Ex: C++ SIM

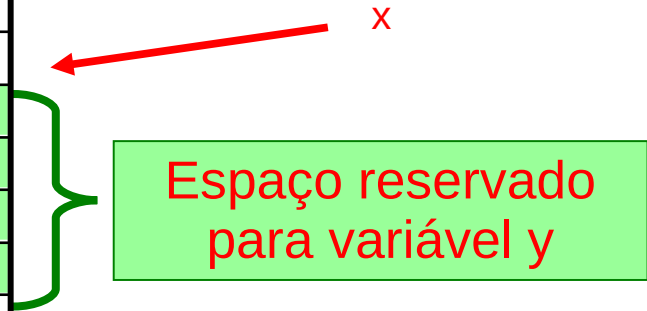
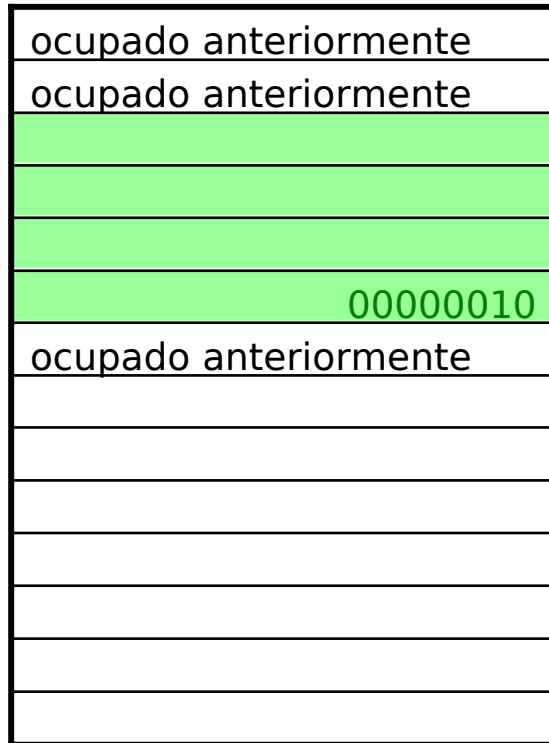
- **Passagem de parâmetro por REFERÊNCIA:** o parâmetro é um apelido (“atalho”) para o argumento (os dois apontam para o mesmo endereço em memória!!!)

```
void fazAlgo (int &x)
{
    (...)
    x++;
    (...)
}
```

```
int y = 2;
```

```
fazAlgo(y);
```

0000  
0001  
0002  
0003  
0004  
0005  
0006  
0007  
0008  
0009  
0010  
0011  
...  
F....FF



# NÃO em C, nem em Java!!!!!! - Ex: C++ SIM

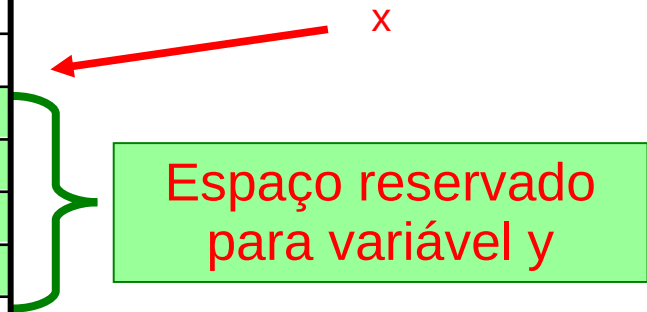
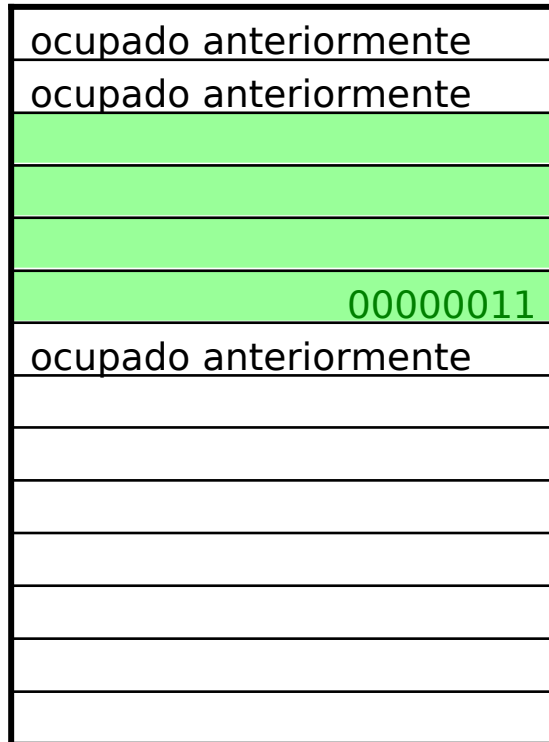
- **Passagem de parâmetro por REFERÊNCIA:** o parâmetro é um apelido (“atalho”) para o argumento (os dois apontam para o mesmo endereço em memória!!!)

```
void fazAlgo (int &x)
{
    (...)
    x++;
    (...)
}
```

```
int y = 2;
```

```
fazAlgo(y);
```

0000  
0001  
0002  
0003  
0004  
0005  
0006  
0007  
0008  
0009  
0010  
0011  
...  
F....FF



# NÃO em C, nem em Java!!!! - Ex: C++ SIM

- **Passagem de parâmetro por REFERÊNCIA:** o parâmetro é um apelido (“atalho”) para o argumento (os dois apontam para o mesmo endereço em memória!!!)

```
void fazAlgo (int &x)
{
    (...)
    x++;
    (...)
}
```

```
int y = 2;
```

```
fazAlgo(y);
```

0000  
0001  
0002  
0003  
0004  
0005  
0006  
0007  
0008  
0009  
0010  
0011  
...  
F....FF

ocupado anteriormente

ocupado anteriormente

00000011

ocupado anteriormente

**Espaço reservado  
para variável y**

# NÃO em C, nem em Java!!!!!! - Ex: C++ SIM

- **Passagem de parâmetro por REFERÊNCIA:** o parâmetro é um apelido (“atalho”) para o argumento (os dois apontam para o mesmo endereço em memória!!!)

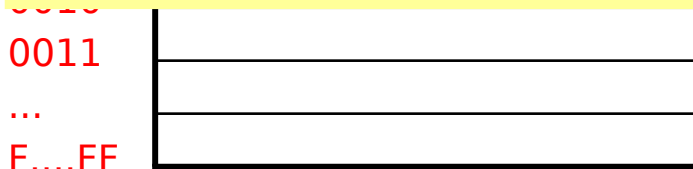
```
void fazAlgo (int &x)
{
    (...)
    x++;
    (...)
}
```

```
int y = 2;
```

```
fazAlgo(y);
```

## ISSO É C++, NÃO É C !!!

Em C, quando nós passamos ponteiro como parâmetro, nós obtemos o efeito de alterar o CONTEÚDO de uma variável dentro de uma função, mas se eu alterar o valor o parâmetro (ponteiro) eu acesso outro endereço de memória!!!



# Em resumo

**Passagem de parâmetros por valor:** o parâmetro é uma cópia do argumento (o conteúdo é o mesmo, mas são dois espaços em memória distintos)

**Passagem de parâmetros por referência:** o parâmetro e o argumento correspondem ao mesmo espaço na memória

# Como simular isso em C?

- ▶ Usando **ponteiros**

# Tipos básicos em C

- Tipos escalares

- Inteiros: tipicamente int
- Caractere: char (mas é um inteiro também)
- Reais: tipicamente usamos double
- Booleanos: bool
- **Ponteiros**: endereço em memória

Ex:

```
int* idade;  
float* temperatura;  
etc...
```

- Tipos não-escalares

- Estruturas: struct
- Uniões: union
- Arranjos: vetores e matrizes

**Endereço** de memória que armazena um conteúdo de determinado **TIPO**

# Como simular isso em C?

- **Passando o endereço (ponteiro) da variável a ser alterada:** o endereço é COPIADO (passagem por valor, afinal), mas aponta para o mesmo espaço em memória que contém a variável que você quer alterar!!!)

```
void fazAlgo (int* x)
{
    (...)
    (*x)++;
    (...)
}
```

```
int y = 2;
```

```
fazAlgo(&y);
```

0000	ocupado anteriormente
0001	ocupado anteriormente
0002	
0003	
0004	
0005	00000010
0006	ocupado anteriormente
0007	
0008	
0009	
0010	
0011	
...	
F....FF	

**Espaço reservado para  
variável y (int)**



# Como simular isso em C?

- ▶ **Passando o endereço (ponteiro) da variável a ser alterada:** o endereço é COPIADO (passagem por valor, afinal), mas aponta para o mesmo espaço em memória que contém a variável que você quer alterar!!!)

```
void fazAlgo (int* x)
{
    (...)
    (*x)++;
    (...)
}
```

```
int y = 2;

fazAlgo(&y);
```

0000	ocupado anteriormente
0001	ocupado anteriormente
0002	
0003	
0004	
0005	00000010
0006	ocupado anteriormente
0007	
0008	
0009	
0010	00000002(hexadecimal)
0011	
...	
F....FF	

Espaço reservado para  
variável y (int)

Espaço reservado para  
variável x (um ponteiro,  
um endereço...)

# Como simular isso em C?

- ▶ **Passando o endereço (ponteiro) da variável a ser alterada:** o endereço é COPIADO (passagem por valor, afinal), mas aponta para o mesmo espaço em memória que contém a variável que você quer alterar!!!)

```
void fazAlgo (int* x)
{
    (...)
    (*x)++;
    (...)
}
```

```
int y = 2;

fazAlgo(&y);
```

0000	ocupado anteriormente
0001	ocupado anteriormente
0002	
0003	
0004	
0005	00000011
0006	ocupado anteriormente
0007	
0008	
0009	
0010	00000002(hexadecimal)
0011	
...	
F....FF	

Espaço reservado para  
variável y (int)

Espaço reservado para  
variável x (um ponteiro,  
um endereço...)

# Como simular isso em C?

- ▶ **Passando o endereço (ponteiro) da variável a ser alterada:** o endereço é COPIADO (passagem por valor, afinal), mas aponta para o mesmo espaço em memória que contém a variável que você quer alterar!!!)

```
void fazAlgo (int* x)
{
    (...)
    (*x)++;
    (...)
}
```

```
int y = 2;

fazAlgo(&y);
```

0000	ocupado anteriormente
0001	ocupado anteriormente
0002	
0003	
0004	
0005	00000011
0006	ocupado anteriormente
0007	
0008	
0009	
0010	
0011	
...	
F....FF	

Espaço reservado para  
variável y (int)

# Outro exemplo clássico: troca

```
void troca(int* x, int* y)
{
    int temp = *x;
    *x = *y;
    *y = temp;
}
```

//Chamada:

```
int x, y;
...
troca(&x,&y);
```

# Como escrever uma função “troca” que funcione em C?

Uso de ponteiros!

Variáveis que armazenam o endereço de uma variável

O OPERADOR unário “&” fornece o endereço de uma variável

O OPERADOR unário “\*” acessa o objeto que um apontador (endereço) aponta

# Voltando ao nosso exemplo... como consertar?

```
void alteraAnoNascimento(Pessoa p, int novoAno)
{
    p.dataNascimento.ano = novoAno;
    return;
}

int main()
{
    Pessoa p1;
    p1.dataNascimento.dia = 27;
    p1.dataNascimento.mes = 9;
    p1.dataNascimento.ano = 2000;

    int idadeP1 = calculaIdadeAproximada(p1);
    printf("Idade: %d\n", idadeP1);

    alteraAnoNascimento(p1, 1990);
    idadeP1 = calculaIdadeAproximada(p1);
    printf("Idade: %d\n", idadeP1);

    return 0;
}
```

# Passando como parâmetro o endereço de Pessoa...

```
void alteraAnoNascimento(Pessoa p, int novoAno)
{
    p.dataNascimento.ano = novoAno;
    return;
}
```

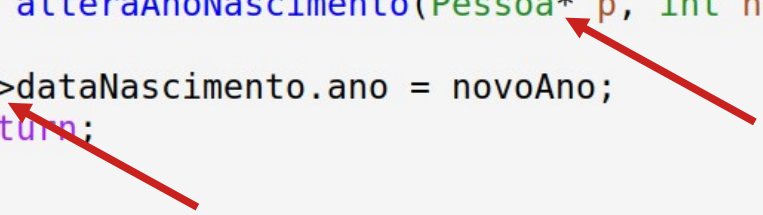
```
int main()
{
    Pessoa p1;
    p1.dataNascimento.dia = 27;
    p1.dataNascimento.mes = 9;
    p1.dataNascimento.ano = 2000;

    int idadeP1 = calculaIdadeAproximada(p1);
    printf("Idade: %d\n", idadeP1);

    alteraAnoNascimento(p1, 1990);
    idadeP1 = calculaIdadeAproximada(p1);
    printf("Idade: %d\n", idadeP1);

    return 0;
}
```

```
void alteraAnoNascimento(Pessoa* p, int novoAno)
{
    p->dataNascimento.ano = novoAno;
    return;
}
```

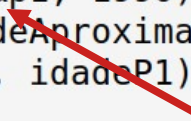


```
int main()
{
    Pessoa p1;
    p1.dataNascimento.dia = 27;
    p1.dataNascimento.mes = 9;
    p1.dataNascimento.ano = 2000;

    int idadeP1 = calculaIdadeAproximada(p1);
    printf("Idade: %d\n", idadeP1);

    alteraAnoNascimento(&p1, 1990);
    idadeP1 = calculaIdadeAproximada(p1);
    printf("Idade: %d\n", idadeP1);

    return 0;
}
```



# Passando como parâmetro o endereço de Pessoa...

Quando a struct é muito grande também usamos essa estratégia para não gastar tempo com a cópia da estrutura durante a passagem de parâmetros

```
void alteraAnoNascimento(Pessoa* p, int novoAno)
{
    p->dataNascimento.ano = novoAno;
    return;
}

int main()
{
    Pessoa p1;
    p1.dataNascimento.dia = 27;
    p1.dataNascimento.mes = 9;
    p1.dataNascimento.ano = 2000;

    int idadeP1 = calculaIdadeAproximada(p1);
    printf("Idade: %d\n", idadeP1);

    alteraAnoNascimento(&p1, 1990);
    idadeP1 = calculaIdadeAproximada(p1);
    printf("Idade: %d\n", idadeP1);

    return 0;
}
```



# Passagem de parâmetros

Quem quiser rever essas explicações:

<https://eaulas.usp.br/portal/video?idItem=30404>

# Escopo de variáveis

- Dependendo da linguagem, as variáveis declaradas no módulo principal são as globais (NÃO é o caso de C)
- Variáveis locais e globais podem ter o mesmo nome
  - Quando o módulo da variável local está executando, vale a variável local
  - O que vocês acham de variáveis locais e globais com o mesmo nome?

# Escopo de variáveis

- Dependendo da linguagem, as variáveis declaradas no módulo principal são as globais (NÃO é o caso de C)
- Variáveis locais e globais podem ter o mesmo nome
  - Quando o módulo da variável local está executando, vale a variável local
  - O que vocês acham de variáveis locais e globais com o mesmo nome?

Péssimo! Pode confundir qual variável você realmente gostaria de acessar...

# Const

Torna a variável constante após definida  
(declaração junto com definição)

Ex:

```
const double PI = 3.1416;
```

```
PI += 1; // ERRO
```

# #define

Também pode definir constantes (mas não é uma variável) e macros  
É uma diretiva do compilador para tocar um TEXTO pelo OUTRO

Ex:

```
#define PI 3.1416
```

#define como macro:

- ver/testar macros\_exNaoOK.c: entender o que aconteceu... como consertar?
- Só depois ver/testar macrosOK.c

# Alocação estática x dinâmica

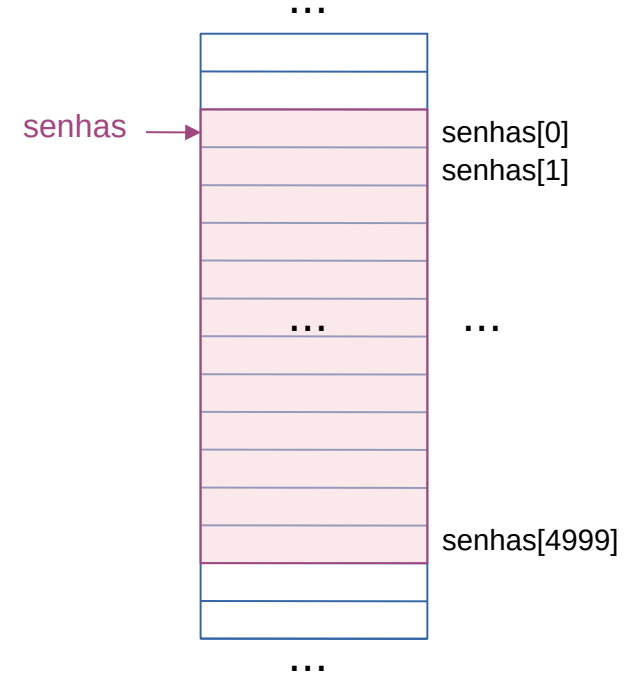
# Ponteiros

Veremos que ponteiros não servem apenas para simular passagem por referência

# Alocação estática x dinâmica

Alocação **estática**: em tempo de compilação

```
#define MAX_SENHAS 5000
int senhas[MAX_SENHAS];
int nrSenhas = 0;
void cadastraSenha(void){
    if (nrSenhas < 5000){
        senhas[nrSenhas] = leSenhaDoTeclado();
    }
}
```



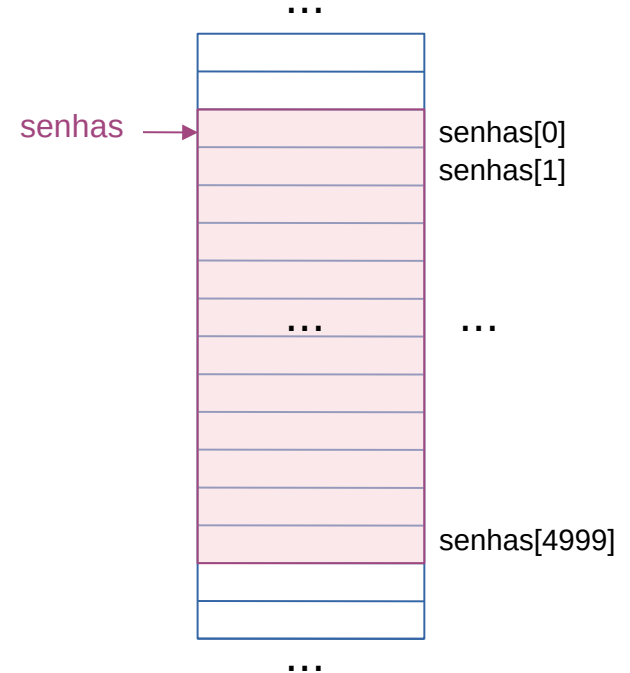


# Alocação estática x dinâmica

Alocação **estática**: em tempo de compilação

```
#define MAX_SENHAS 5000
int senhas[MAX_SENHAS];
int nrSenhas = 0;
void cadastraSenha(void){
    if (nrSenhas < 5000){
        senhas[nrSenhas] = leSenhaDoTeclado();
    }
}
```

Prós e contras?



# Alocação estática x dinâmica

Alocação **estática**: em tempo de compilação

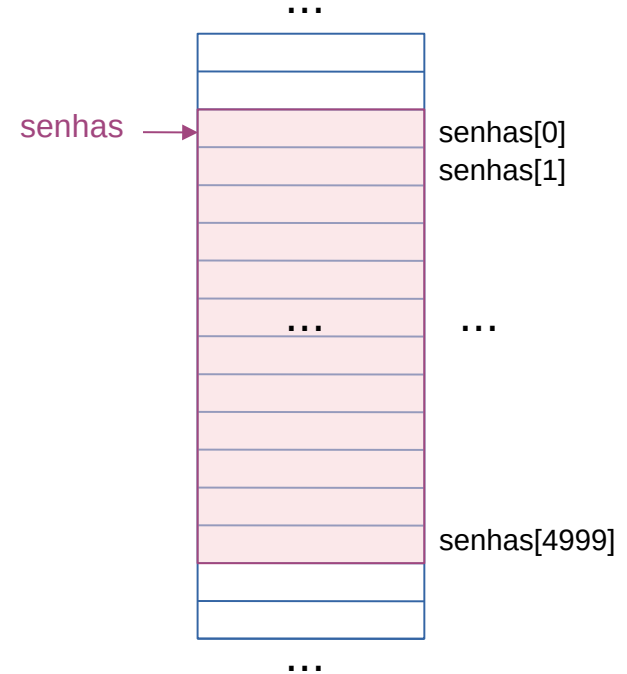
```
#define MAX_SENHAS 5000
int senhas[MAX_SENHAS];
int nrSenhas = 0;
void cadastraSenha(void){
    if (nrSenhas < 5000){
        senhas[nrSenhas] = leSenhaDoTeclado();
    }
}
```

Prós:

- fácil de gerenciar
- memória liberada quando finaliza o trecho no qual ela foi declarada (ex: dentro de uma função)

Contra:

- tamanho constante (e se o tamanho adequado eu só conhecesse em tempo de execução?)



# Alocação estática x dinâmica (em C)

Alocação **dinâmica**: em tempo de **execução**

```
#include <stdlib.h>
```

```
int* senhas;
```

```
int nrSenhas = leNrSenhas() //do teclado, de um arquivo...
```

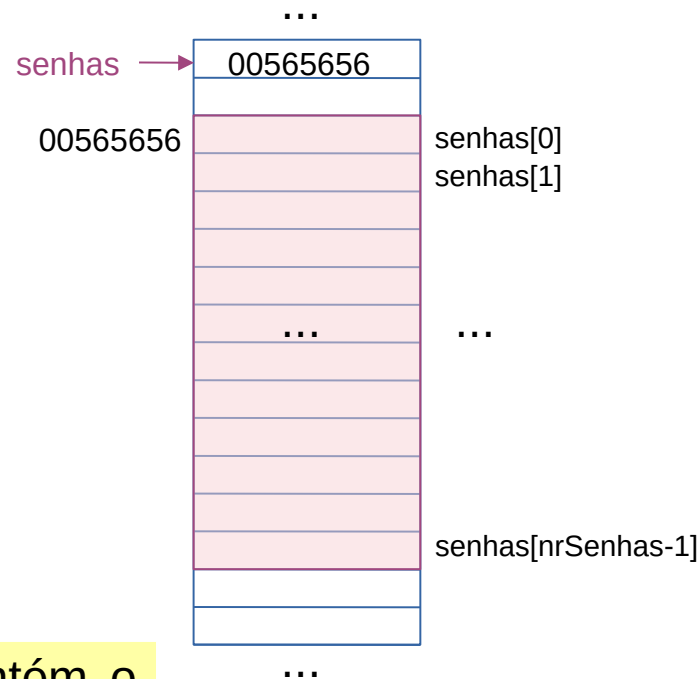
```
senhas = (int*) malloc(sizeof(int) * nrSenhas);
```

```
for (int i = 0; i < nrSenhas; i++)
```

```
    senhas[i] = leProximaSenha();
```

```
...
```

```
free(senhas); // libera quando não for mais usar
```



- Ou seja, **senhas** é uma variável que contém o endereço do int que está na posição 0 do vetor.
- Como todos os elementos do vetor ficam contíguos na memória, **senhas** contém o endereço de onde esse vetor começa!

# Exemplo: o que essa função (em C) faz?

```
#include <stdio.h>

void funcaoMisteriosa(int n, int* array){
    for (int i = 0; i < n; i++){
        (*array)++;
        array++;
    }
}

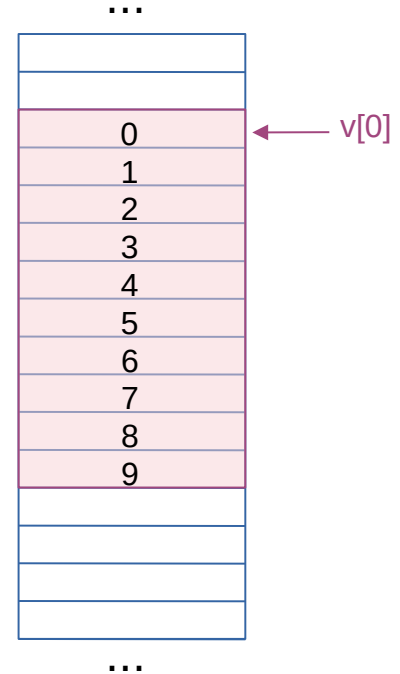
int main(void){
    int v[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    funcaoMisteriosa(10, v);
    for (int i = 0; i < 10; i++)
        printf("%d ", v[i]);
    printf("\n");
}
```

# Exemplo: o que essa função (em C) faz?

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

```
void funcaoMisteriosa(int n, int* array){  
    for (int i = 0; i < n; i++){  
        (*array)++;  
        array++;  
    }  
}
```

```
int main(void){  
    int v[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};  
    funcaoMisteriosa(10, v);  
    for (int i = 0; i < 10; i++)  
        printf("%d ", v[i]);  
    printf("\n");  
}
```



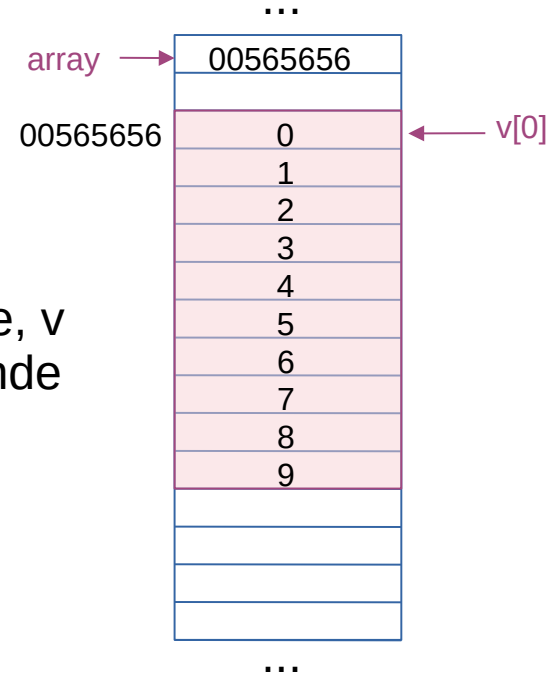
# Exemplo: o que essa função (em C) faz?

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

```
void funcaoMisteriosa(int n, int* array){ ←  
    for (int i = 0; i < n; i++){  
        (*array)++;  
        array++;  
    }  
}
```

Ou seja, mesmo que alocado estaticamente, v também contém o endereço de memória onde começa o vetor

```
int main(void){  
    int v[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};  
    funcaoMisteriosa(10, v); ←  
    for (int i = 0; i < 10; i++)  
        printf("%d ", v[i]);  
    printf("\n");  
}
```

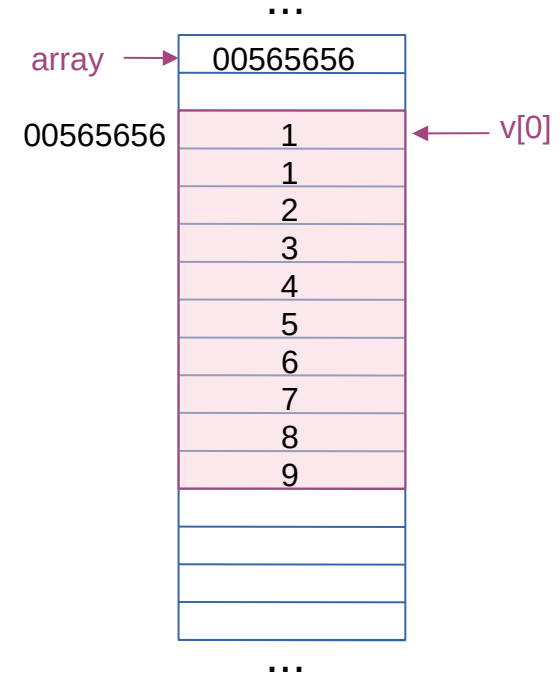


# Exemplo: o que essa função (em C) faz?

```
#include <stdio.h>

void funcaoMisteriosa(int n, int* array){
    for (int i = 0; i < n; i++){
        (*array)++;
        array++;
    }
}

int main(void){
    int v[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    funcaoMisteriosa(10, v);
    for (int i = 0; i < 10; i++)
        printf("%d ", v[i]);
    printf("\n");
}
```

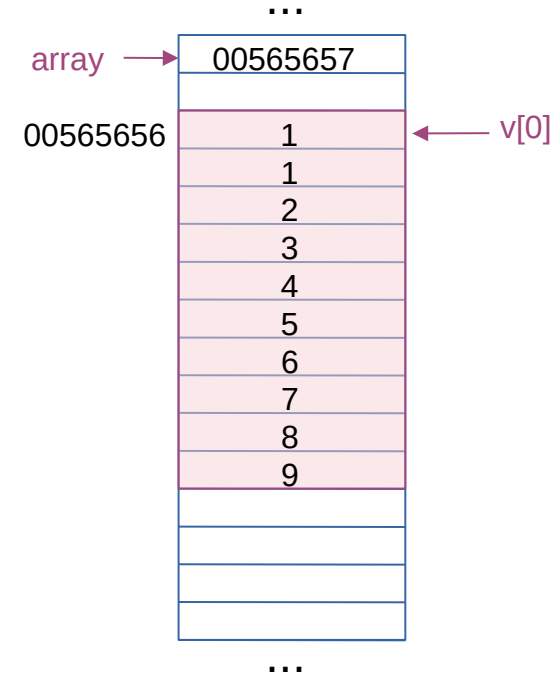


# Exemplo: o que essa função (em C) faz?

```
#include <stdio.h>

void funcaoMisteriosa(int n, int* array){
    for (int i = 0; i < n; i++){
        (*array)++;
        array++;
    }
}

int main(void){
    int v[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    funcaoMisteriosa(10, v);
    for (int i = 0; i < 10; i++)
        printf("%d ", v[i]);
    printf("\n");
}
```



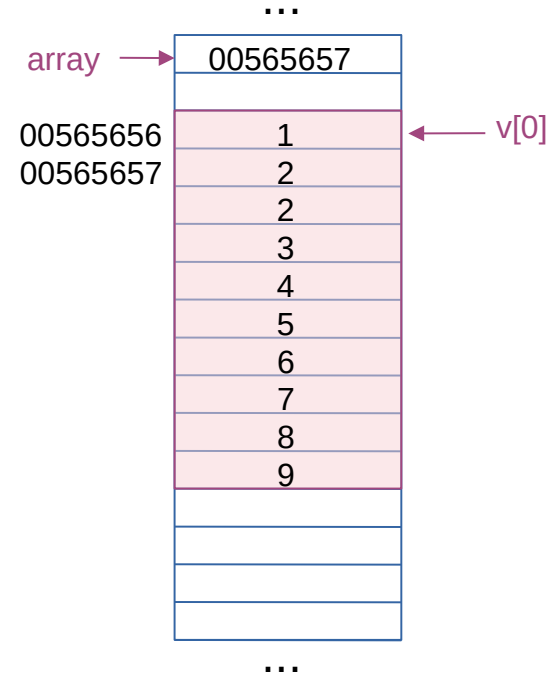


# Exemplo: o que essa função (em C) faz?

```
#include <stdio.h>

void funcaoMisteriosa(int n, int* array){
    for (int i = 0; i < n; i++){
        (*array)++;
        array++;
    }
}

int main(void){
    int v[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    funcaoMisteriosa(10, v);
    for (int i = 0; i < 10; i++)
        printf("%d ", v[i]);
    printf("\n");
}
```

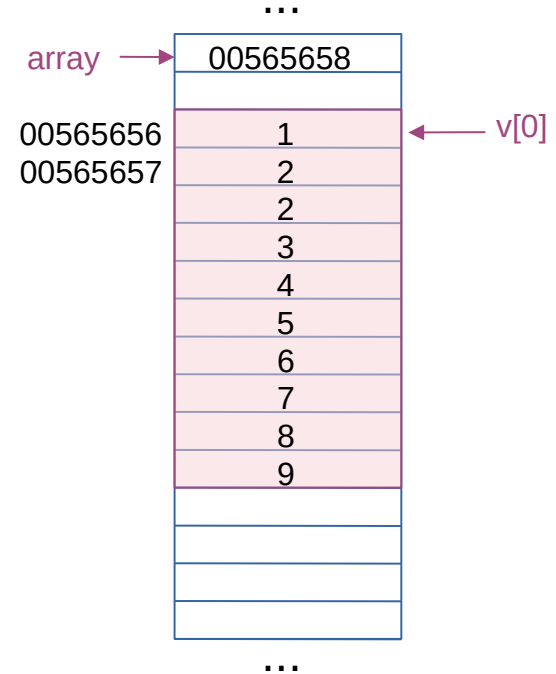


# Exemplo: o que essa função (em C) faz?

```
#include <stdio.h>

void funcaoMisteriosa(int n, int* array){
    for (int i = 0; i < n; i++){
        (*array)++;
        array++;
    }
}

int main(void){
    int v[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    funcaoMisteriosa(10, v);
    for (int i = 0; i < 10; i++)
        printf("%d ", v[i]);
    printf("\n");
}
```



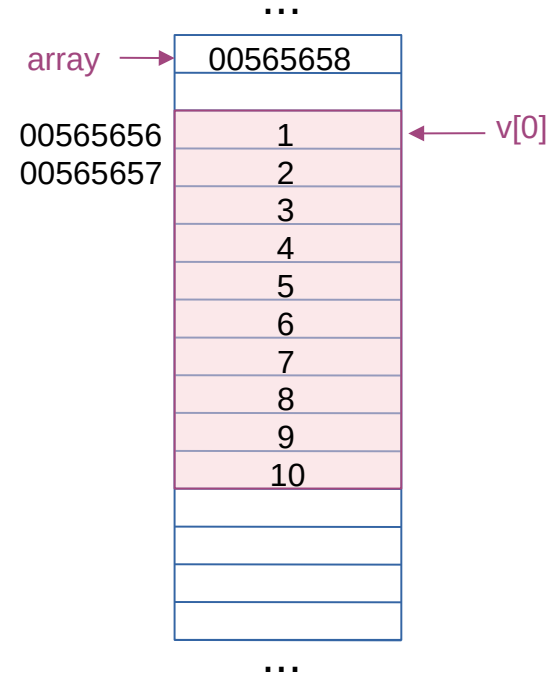
# Exemplo: o que essa função (em C) faz?

```
#include <stdio.h>

void funcaoMisteriosa(int n, int* array){
    for (int i = 0; i < n; i++){
        (*array)++;
        array++;
    }
}

int main(void){
    int v[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    funcaoMisteriosa(10, v);
    for (int i = 0; i < 10; i++)
        printf("%d ", v[i]);
    printf("\n");
}
```

n vezes...

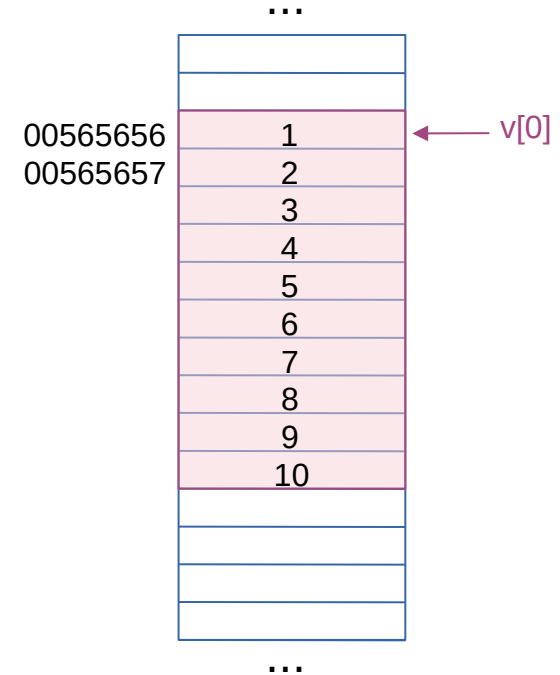


# Exemplo: o que essa função (em C) faz?

```
#include <stdio.h>

void funcaoMisteriosa(int n, int* array){
    for (int i = 0; i < n; i++){
        (*array)++;
        array++;
    }
}

int main(void){
    int v[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    funcaoMisteriosa(10, v);
    for (int i = 0; i < 10; i++) ←
        printf("%d ", v[i]);
    printf("\n");
}
```



# Obs: inicialização

Se precisar inicializar o vetor, use `calloc` (ele já inicializa com 0 de forma eficiente)

```
senhas = (int*) malloc(sizeof(int) * nrSenhas);  
for (int i = 0; i < nrSenhas; i++)  
    senhas[i] = 0;
```

é equivalente a

```
senhas = (int*) calloc(nrSenhas, sizeof(int));
```

# Alocação estática x dinâmica

Posso também fazer alocação dinâmica de qualquer outro tipo

```
Data* dataNasc;    // struct  
...  
dataNasc = (Data*) malloc (sizeof(Data));  
...  
dataNasc → ano = 2013;  
...  
free(dataNasc);
```

# Gerenciamento de memória em C

Cuidado com os casamentos entre malloc e free!

# Gerenciamento de memória em C

Dica: substitua o `free(p)` por:

```
if (p != NULL)
{ free(p); p = NULL; }
```

(evita dar um `free` duas vezes no mesmo ponteiro)



# Gerenciamento de memória em C

Dica: substitua o malloc:

```
if ((senhas = (int*) malloc(sizeof(inteiro) *  
    nr_senhas)) == NULL)  
{ printf “Erro na alocação de senhas na função  
    tal.\n”; }
```

# Exercícios

Brinquem com criar e manipular estatica e dinamicamente:

- struct contendo vetores (arrays)
- vetor de structs (ex: Aluno\* turma)