

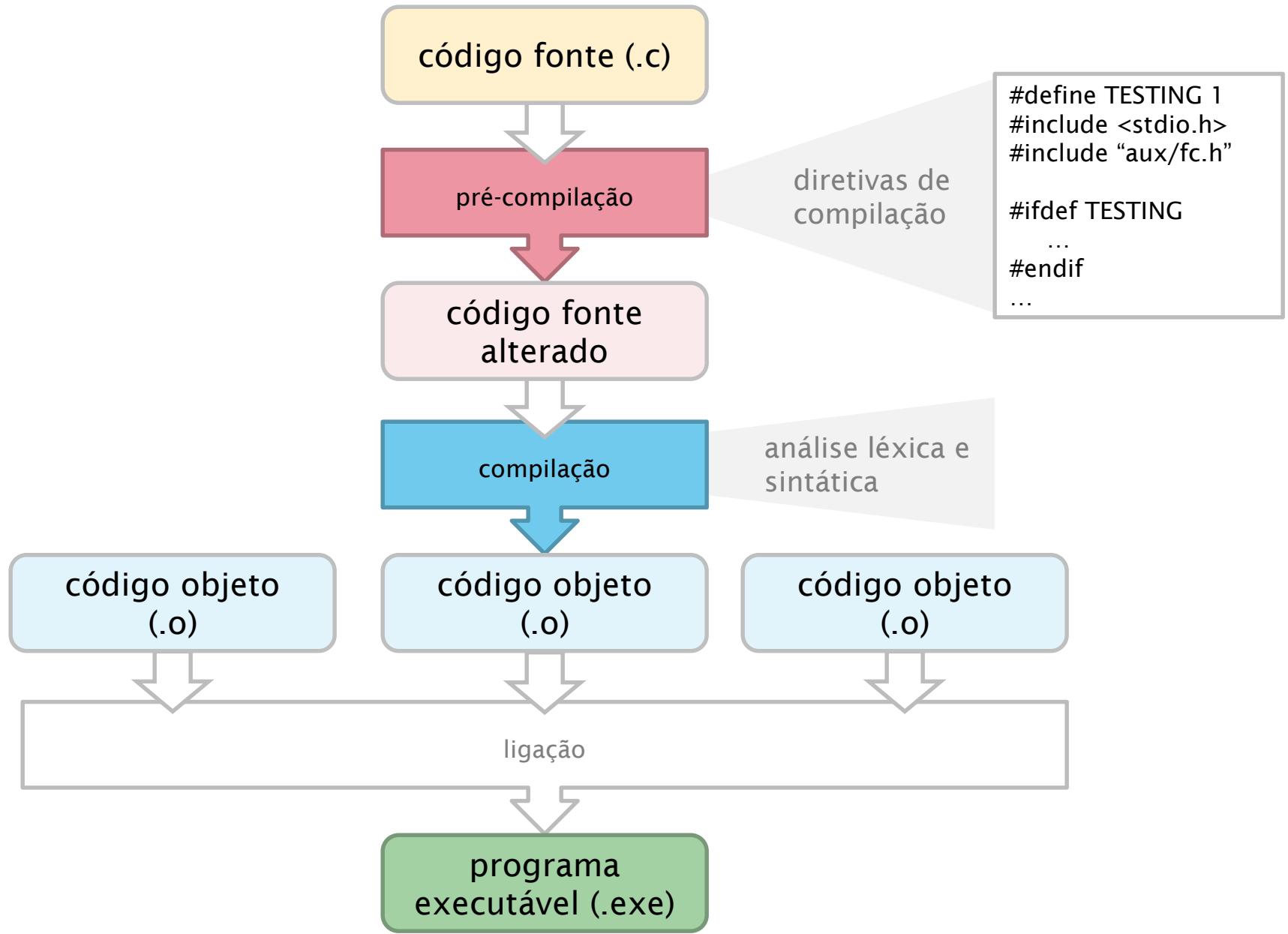


INF 1010

Estruturas de Dados Avançadas

Revisão de C

etapas de compilação revisão



Diretiva #include “arquivo” ou #include <arquivo>

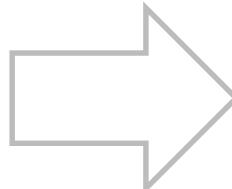
o pré-processador substitui o “include” pelo corpo do arquivo especificado

arquivo fc.h no subdiretório aux

```
void testa_fatorial(int n);  
void testa_etc();
```

arquivo prog.c

```
#include "aux/fc.h"  
  
int main(void)  
{  
    testa_fatorial(1);  
    return 0;  
}
```



```
void testa_fatorial(int n);  
void testa_etc();  
  
int main(void)  
{  
    testa_fatorial(1);  
    return 0;  
}
```

Diretiva #define constante valor

o pré-processador substitui toda ocorrência da constante pelo valor especificado

```
#define PI 3.14159F
```

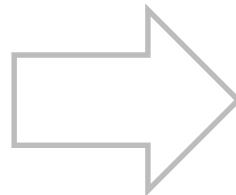
```
float area (float r)
```

```
{
```

```
    float a = PI * r * r;
```

```
    return a;
```

```
}
```



```
float area (float r)
```

```
{
```

```
    float a = 3.14159F * r * r;
```

```
    return a;
```

```
}
```

Diretiva #define *nome macro*

definição com parâmetros

```
#define MAX(a,b) ((a) > (b) ? (a) : (b))
```

...

```
v = 4.5;
```

```
c = MAX(v, 3.0);
```

...

```
v = 4.5;
```

```
c = ((v) > (3.0) ? (v) : (3.0));
```

...

Diretiva #define: cuidados

macro e seus parâmetros entre parênteses

exemplo

#define MULT(a,b) a*b

int x = MULT (3+2, 4) /* $3+2*4 = 3+8 = 11$, **ERRADO** */

#define MULT(a,b) ((a)*(b))

int x = MULT (3+2, 4) /* $((3+2)*(4)) = (5*4) = 20$, **CERTO** */

Evitando Dupla Inclusão

Dependendo da ordem dos #include, um mesmo arquivo pode ser incluído duas ou mais vezes criando problemas de compilação, para isso é recomendado usar macros guarda (#include guard)

```
#ifndef _FULANO
#define _FULANO
struct fulano
{
    int valor;
};
#endif
```

tipos de dados revisão

Bits vs. Algarismos Decimais

bit

0

1

algarismo
decimal

0

1

2

3

4

5

6

7

8

9

Sistemas numéricos

decimal

9	5	0	7
---	---	---	---

$$9*10^3 + 5*10^2 + 0*10^1 + 7*10^0 = \\ 9000 + 500 + 0 + 7$$

binário

1	0	1	1
---	---	---	---

$$1*2^3 + 0*2^2 + 1*2^1 + 1*2^0 = \\ 8 + 0 + 2 + 1 = 11$$

Bits & Bytes

bit

0

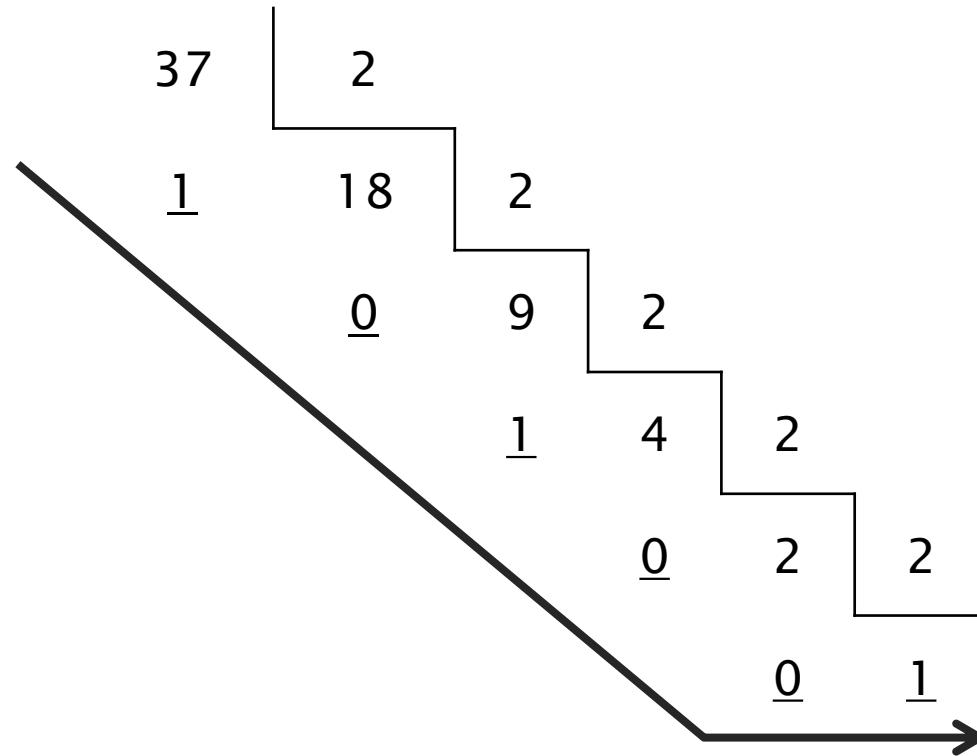
1

byte

0 0 0 0 1 0 1 1

8 bits

Como obter a representação binária de um número decimal?



$$37 = 2^5 + 2^2 + 2^0 = 32 + 4 + 1 = \underline{\underline{00100101}}$$

Quanto cabe num byte?

2^8 valores → 256 valores

sem sinal: 0 a 255

1	1	1	1	1	1	1	1
---	---	---	---	---	---	---	---

(unsigned char)

$$= 255 \quad (0xFF)$$

inteiros: -128 a 127

1	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---

(signed char)

1	0	0	0	0	0	0	1
---	---	---	---	---	---	---	---

$$= -128 \quad (0x80)$$

1	0	0	0	0	0	1	0
---	---	---	---	---	---	---	---

$$= -127 \quad (0x81)$$

1	0	0	0	0	0	1	0
---	---	---	---	---	---	---	---

$$= -126 \quad (0x82)$$

1	1	1	1	1	1	1	1
---	---	---	---	---	---	---	---

$$= -1 \quad (0xFF)$$

0	1	1	1	1	1	1	1
---	---	---	---	---	---	---	---

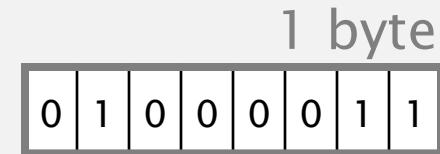
$$= 127 \quad (0x7F)$$

Tipos de dados inteiros "tradicionalmente"

char

unsigned char: 0 a 255

char -128 a 127

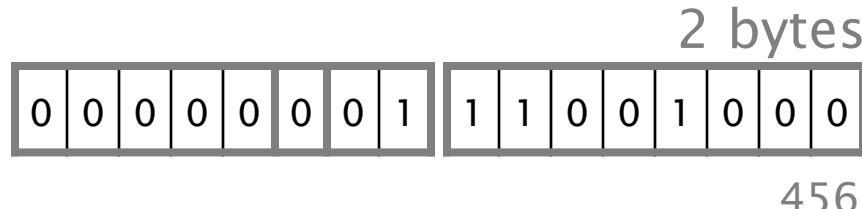


letra 'C' = 67

short int

short int: -32 768 a 32 767

unsigned short int: 0 a 65 535



456

long int

long int: -2 147 483 648 a 2 147 483 647

unsigned long int: 0 a 4 294 967 295

4 bytes



1.077.150.273

Tabela ASCII

000	(nul)	016 ► (dle)	032 sp	048 ò	064 @	080 P	096 `	112 p
001	☺ (soh)	017 ◀ (dc1)	033 !	049 1	065 A	081 Q	097 a	113 q
002	☻ (stx)	018 ↴ (dc2)	034 "	050 2	066 B	082 R	098 b	114 r
003	♥ (etx)	019 !! (dc3)	035 #	051 3	067 C	083 S	099 c	115 s
004	♦ (eot)	020 ¶ (dc4)	036 \$	052 4	068 D	084 T	100 d	116 t
005	♣ (enq)	021 § (nak)	037 %	053 5	069 E	085 U	101 e	117 u
006	♠ (ack)	022 – (syn)	038 &	054 6	070 F	086 V	102 f	118 v
007	• (bel)	023 ↵ (etb)	039 '	055 7	071 G	087 W	103 g	119 w
008	▣ (bs)	024 ↑ (can)	040 (056 8	072 H	088 X	104 h	120 x
009	(tab)	025 ↓ (em)	041)	057 9	073 I	089 Y	105 i	121 y
010	(lf)	026 (eof)	042 *	058 :	074 J	090 Z	106 j	122 z
011	♂ (vt)	027 ← (esc)	043 +	059 ;	075 K	091 [107 k	123 {
012	♀ (np)	028 ↳ (fs)	044 ,	060 <	076 L	092 \	108 l	124
013	(cr)	029 ↔ (gs)	045 -	061 =	077 M	093]	109 m	125 }
014	¤ (so)	030 ▲ (rs)	046 .	062 >	078 N	094 ^	110 n	126 ~
015	⊗ (si)	031 ▼ (us)	047 /	063 ?	079 O	095 _	111 o	127 □

Tabela ASCII estendida

128	Ç	143	Å	158	⌐	172	¼	186		200	└─	214	━	228	Σ	242	≥
129	ü	144	É	159	f	173	;`	187	»	201	└─	215	━	229	σ	243	≤
130	é	145	æ	160	á	174	«	188	„	202	└─	216	━	230	µ	244	∫
131	â	146	Æ	161	í	175	»	189	„	203	┐	217	┘	231	τ	245]
132	ä	147	ö	162	ó	176	„	190	„	204	┐	218	┌	232	Φ	246	÷
133	à	148	ö	163	ú	177	„	191	„	205	=	219	█	233	Θ	247	≈
134	å	149	ò	164	ñ	178	„	192	„	206	┐	220	█	234	Ω	248	°
135	ç	150	û	165	Ñ	179	„	193	„	207	└─	221	█	235	δ	249	·
136	ê	151	ù	166	ª	180	„	194	„	208	┐	222	█	236	∞	250	·
137	ë	152	ÿ	167	º	181	„	195	„	209	┐	223	█	237	φ	251	√
138	è	153	Ö	168	¿	182	„	196	—	210	└─	224	α	238	ε	252	¤
139	ï	154	Ü	169	¬	183	„	197	+	211	└─	225	ß	239	∩	253	²
140	î	155	¢	170	¬	184	„	198	+	212	└	226	Γ	240	≡	254	■
141	ì	156	£	171	½	185	„	199	+	213	━	227	π	241	±	255	
142	Ä	157	¥														

Tipos de dados inteiros (cont.)

	tipo	bytes	valores
	char	1	-128 a 127
	short int	2	-32.768 a 32.767
números inteiros	long int	4	-2.147.483.648 a 2.147.483.647
	long long	8	-9.223.372.036.854.775.808 a 9.223.372.036.854.775.807
	int	(em geral = long int)	

Tipo de dados Reais

Forma de representação criada por Konrad Zuse para os seus computadores Z1 e Z3.

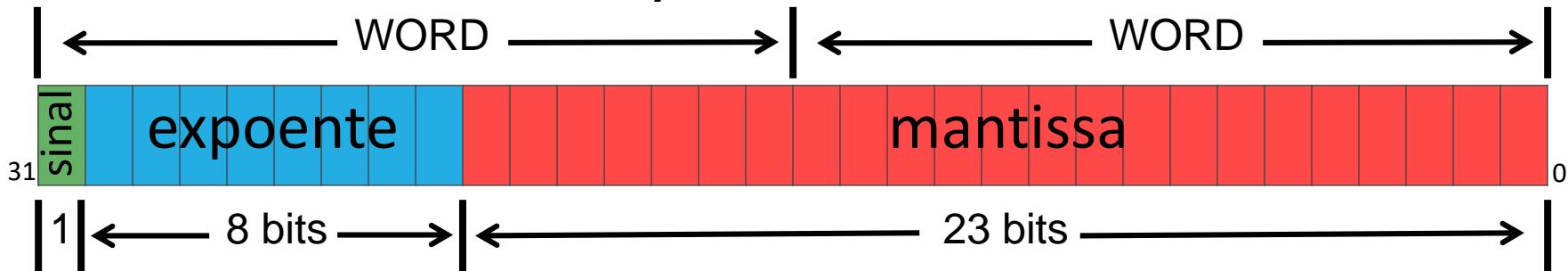
Diversas representações de ponto flutuante existem, porém a mais utilizada é a norma IEEE 754-2008:

Precisão	bits	tipo	faixa	casas
simples	32	float	$\sim 10^{-38}$ a 10^{38}	~ 7
dupla	64	double	$\sim 10^{-308}$ a 10^{308}	~ 15

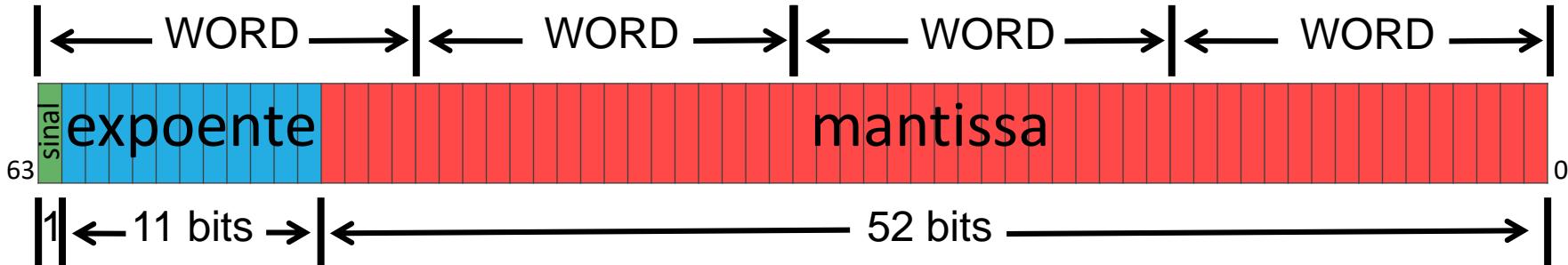
Ponto Flutuante

Representação no IEEE 754-2008:
sinal (S), mantissa (M) e expoente (E) => $S \cdot (1+M) \cdot 2^{E-C}$

- *IEEE 754-2008 Precisão Simples*



- *IEEE 754-2008 Precisão Dupla*

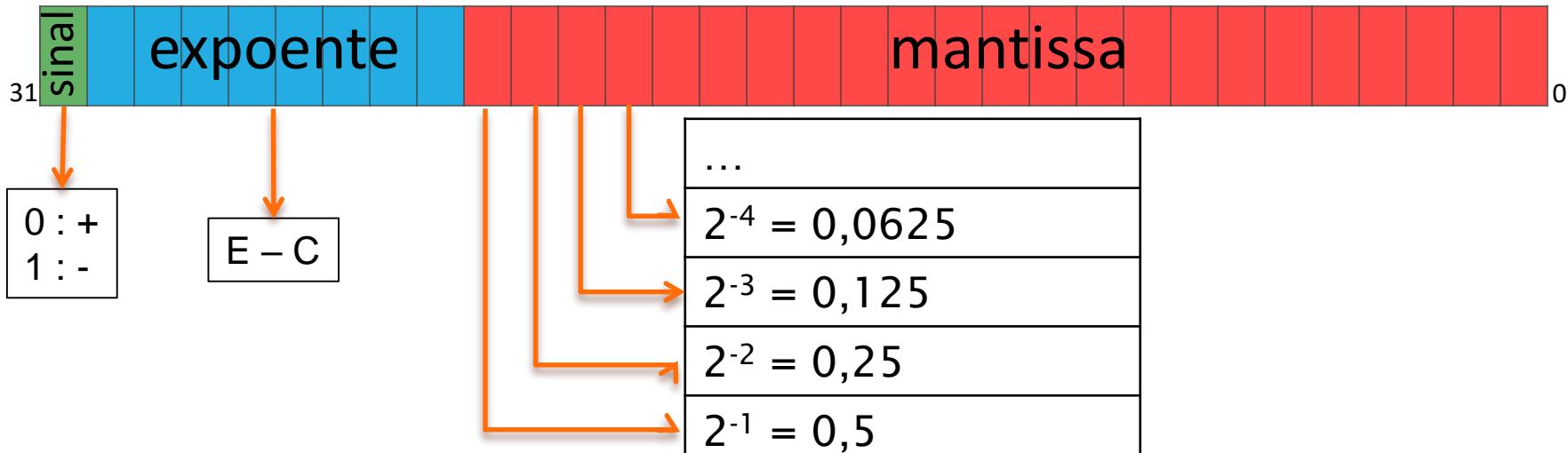


Ponto Flutuante

O sinal é + (positivo) para 0 (zero) e - (negativo) para 1(um).

O expoente se calcula subtraindo 127(precisão simples) ou 1023(precisão dupla) do valor armazenado.

A mantissa é sempre normalizada entre 1 e 2, assim sua parte inteira é sempre 1 (um), que não é necessária representar.



Exemplo de Ponto Flutuante

bits	s	mant.	exp.	cálculo	valor
0 0111111 0000000000000000000000000000	0	0	127	$+(1+0) \cdot 2^{(127-127)}$	1
1 0111110 0000000000000000000000000000	1	0	126	$-(1+0) \cdot 2^{(126-127)}$	-0,5
0 00000111 1000000000000000000000000000	0	0,75	7	$+(1+0,75) \cdot 2^{(7-127)}$	$\sim 1,31 \cdot 10^{-36}$
1 10000001 0110000000000000000000000000	1	0,375	129	$-(1+0,375) \cdot 2^{(129-127)}$	-5,5

Valores Especiais de Ponto Flutuante

Os seguintes valores são especiais

IEEE 754 - Single Precision			Valor
s	e	m	
0	0000 0000	000 0000 0000 0000 0000 0000	+0
1	0000 0000	000 0000 0000 0000 0000 0000	-0
0	1111 1111	000 0000 0000 0000 0000 0000	+Inf
1	1111 1111	000 0000 0000 0000 0000 0000	-Inf
0	1111 1111	010 0000 0000 0000 0000 0000	+NaN
1	1111 1111	010 0000 0000 0000 0000 0000	-NaN

Constantes

valores constantes

```
int a = 5;
```

```
int b;
```

```
b = a + 12;
```

exemplos

inteiros

13 -4

double

12.45

1245e-2

float

12.45F

char

'a' 'A'

Operadores

aritméticos

+ - * / % (módulo)

atribuição

=

incremento e
decremento

++ --

relacionais e lógicos

> < <= >= == !=

&& (and)

|| (or)

! (not)

Operadores aritméticos

+ - * / % (módulo)

$7 / 2 \rightarrow 3$ (a parte fracionária é **descartada**)

$7 / 2.0 \rightarrow 7.0 / 2.0 \rightarrow 3.5$

$7.0 / 2 \rightarrow 7.0 / 2.0 \rightarrow 3.5$

(converte operandos para a **maior precisão**)

Operadores aritméticos – Módulo

Resto da divisão inteira
(operандos devem ser inteiros)

$0 \% 3$ resulta em 0

$1 \% 3$ resulta em 1

$2 \% 3$ resulta em 2

$3 \% 3$ resulta em 0

$4 \% 3$ resulta em 1

$5 \% 3$ resulta em 2

Útil para identificar números pares ou ímpares

se $x \% 2$ é 0, o número x é par

se $x \% 2$ é 1, o número x é ímpar

Em que outras
situações o $\%$ é útil?

Operadores de atribuição

=

a = 5;

y = x = 5;

operadores de atribuição compostos

i = i + 2 equivale a i += 2

i = i * 2 equivale a i *= 2

i = i / 2 equivale a i /= 2

...

var = var *op* (expr); equivale a var *op=* expr

x *= y+1 equivale ao quê? x = x * (y+1)

Operadores de incremento e decremento

```
int n,x;  
  
n = 5;                                /* n ← 5 */  
  
x = n++;                                /* x ← 5; n incrementa para 6 */  
  
x = ++n;                                /* n incrementa para 7; x ← 7 */  
  
x = ++n * 2;                            /* n incrementa para 8; x ← 16 */  
  
x = n++ * 2;                            /* x ← 16; n incrementa para 9 */
```

Operadores relacionais

> < <= >= == !=

usados numa expressão, resultam em

0, caso a expressão seja falsa, ou

1, caso a expressão seja verdadeira

Exemplo:

int a=10, b=10, c=5;

a > b resulta em 0

b > c resulta em 1

a == b resulta em 1

b != c resulta em 1

Operadores lógicos

&& (and) || (or) ! (not)

Exemplo: int a = 2, b = 4, c = 5, d = 7;

`(a < b) && (c < d)` /* 1 && 1 -> resulta em 1;
ambas expressões são avaliadas */

`(a > b) && (c < d)` /* (a>b) 0 -> resulta em 0;
somente (a>b) é avaliada */

`(a < b) || (d > c) /* (a<b) 1 -> resulta em 1;
somente (a<b) é avaliada */`

`(a > b) || (c > d)` /* 0 && 0 -> resulta em 0;
ambas expressões são avaliadas */

Operador *sizeof*

número de bytes ocupados por um tipo

```
int a = sizeof(long);           /* 4 */  
a = sizeof(char);              /* 1 */  
a = sizeof(short);             /* 2 */  
a = sizeof(float);             /* 4 */  
a = sizeof(double);            /* 8 */
```

Número de bytes ocupados por um ponteiro

máquina de 32 bits

`sizeof(void*)`: 4

```
C:\Windows\system32\cmd.exe
=====
Numero de bytes por tipo:
sizeof(char):      1
sizeof(short):     2
sizeof(int):       4
sizeof(long):      4
sizeof(long long): 8
sizeof(void*):      4
sizeof(float):     4
sizeof(double):    8
```

máquina de 64 bits

`sizeof(void*)`: 8

```
C:\Windows\system32\cmd.exe
=====
Numero de bytes por tipo:
sizeof(char):      1
sizeof(short):     2
sizeof(int):       4
sizeof(long):      4
sizeof(long long): 8
sizeof(void*):      8
sizeof(float):     4
sizeof(double):    8
```

variáveis revisão

Tipos de Variáveis

global

```
int a;           /* declarada no módulo, fora de qualquer função */
```

local ou automática

```
int a;           /* declarada dentro de um bloco, como uma função */
```

estática

```
static int a;  
/* declarada numa função mas alocada em área fixa de memória;  
valor é mantido durante toda a execução do programa */
```

externa

```
extern int a;           /* global de um outro módulo */
```

Como declarar uma variável?

declaração

```
float tempC;  
float tempF;
```

atribuição de valor
(após a declaração)

```
int a, b;  
a = 5;  
b = 10;
```

declaração de variáveis
de um mesmo tipo

```
float tempC, tempF;  
int a, b;
```

declaração com
inicialização

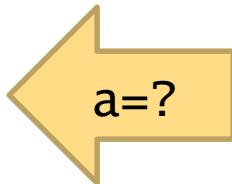
```
int a = 5, b = 10;
```

Qual é o valor de *a* em cada exemplo?

Exemplo 1

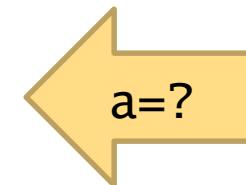
```
int a;
```

```
a = 4;
```



Exemplo 3

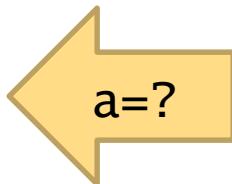
```
int a, b = 5;
```



Exemplo 2

```
int a;
```

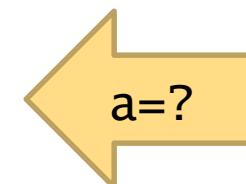
```
a = 4.9;
```



Exemplo 4

```
int a, b, c=3;
```

```
a = b+c;
```



Conversão de tipo

implicitamente
(automática, na avaliação de uma
expressão)

```
float a = 3; /* conversão para 3.0F */
```

explicitamente, através de *cast*

```
int a;
```

```
float b = 3.5;
```

```
a = (int) b;
```

Exercício

Suponha que:

a = 3;

b = a / 2;

c = b + 3.1;

Como as variáveis a, b e c devem ser declaradas para obter cada um dos seguintes resultados?

c = 4.6 float a, b, c;

c = 4.1 int a, b; float c;

c = 4 int a, b, c;

Variáveis de um bloco

declaração da variável i

```
if ( n > 0 )
{
    int i; /* só pode declarar no início do bloco */
    ...
    i = f(n);
    ...
} /* a variável i já não existe neste ponto */
```

escopo da variável i

Variável global

variável declarada fora das funções

não faz parte da pilha de execução

é visível por todas as funções

existe enquanto o programa estiver sendo executado

causa interdependência entre funções

pode tornar código difícil de entender e reutilizar

```
#include <stdio.h>

int s, p; /* declaração de variáveis globais */

void somaprod (int a, int b) {
    ...
}

int main (void) {
    ...
}
```

Exemplo de programa

```
#include <stdio.h>

int s, p;      /* declaração de variáveis globais */

void somaprod (int a, int b)  {
    s = a + b;    /* atribuição a variáveis globais */
    p = a * b;
}

int main (void) {
    int x, y;
    printf("Digite dois numeros: ");
    scanf("%d %d", &x, &y);
    somaprod(x, y);
    printf("Soma: %d, produto: %d \n", s, p);
    return 0;
}
```

```
Digite dois numeros: 2 3
Soma: 5, produto: 6
Press any key to continue . . .
```

Variável estática

`static int i;`

declarada no corpo de uma função:

- visível apenas naquela função

não é armazenada na pilha de execução:

- área de memória estática

- existe enquanto o programa estiver sendo executado

utilização de variável estática:

- quando for necessário recuperar o valor de uma variável atribuída na última vez que a função foi executada

Exemplo – Variável estática

```
#include <stdio.h>

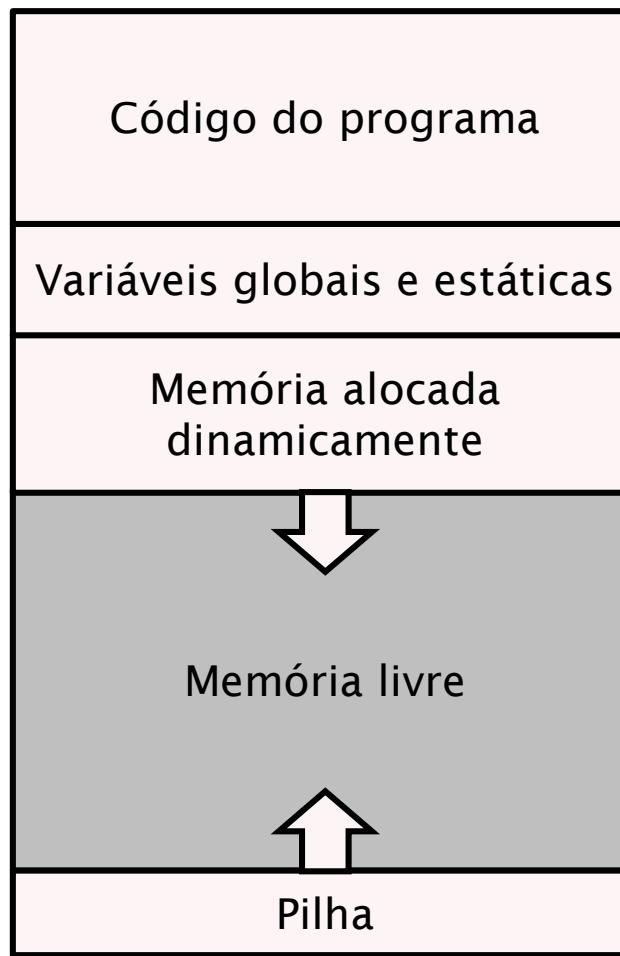
void imprime(void)
{
    static int i = 1;
    printf("%d\n", i++);
}

int main(void)
{
    printf("Primeiro numero: ");
    imprime();
    printf("Proximo numero: ");
    imprime();
    imprime();
}
```

```
Primeiro numero: 1
Proximo numero: 2
3
```

uso da memória revisão

Uso esquemático da memória



```
/* exemplo do uso da memória */
```

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

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

```
char delim = ',';
```

```
char *concat(char a, char b)
```

```
{
```

```
    char *sigla = (char *)malloc(4);
    sigla[0] = a;
    sigla[1] = delim;
    sigla[2] = b;
    sigla[3] = '\0';
    return sigla;
}
```

```
int main(void)
```

```
{
```

```
    char a='A', b='Z';
    char *s;
    printf("Digite o delimitador: ");
    scanf(" %c", &delim);
    s = concat(a,b);
    printf("%s\n", s);
    free(s);
    return 0;
}
```

variável
global

```
/* exemplo do uso da memória */
#include <stdio.h>
#include <string.h>
char delim = ',';
char *concat(char a, char b){
    char *sigla = (char *)malloc(4);
    sigla[0] = a;
    sigla[1] = delim;
    sigla[2] = b;
    sigla[3] = '\0';
}

int main(void){
    char a='A', b='Z';
    printf("Digite o delimitador: ");
    scanf(" %c", &delim);
    s = concat(a,b);
    printf("%s\n", s);
    free(s);
    return 0;
}
```

delim

,

Memória livre

não há
espaço
alocado
dinamic.

ainda não
há nada na
pilha

```
/* exemplo do uso da memória */
```

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

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

```
char delim = ',';
```

```
char *concat(char a, char b)
```

```
{
```

```
    char *sigla = (char *)malloc(4);  
    sigla[0] = a;  
    sigla[1] = delim;  
    sigla[2] = b;  
    sigla[3] = '\0';  
    return sigla;
```

```
}
```

```
int main(void)
```

```
{
```

```
    char a='A', b='Z';
```

```
    char *s;
```

```
    printf("Digite o delimitador: ");
```

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

```
    s = concat(a,b);
```

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

```
    free(s);
```

```
    return 0;
```

variável
global

delim

,

variáveis
lokais à
função
main

Memória livre

s ?
b 'Z'
a 'A'

não há
espaço
alocado
dinamic.

Pilha

```
/* exemplo do uso da memória */
```

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

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

```
char delim = ',';
```

```
char *concat(char a, char b)
```

```
{
```

```
    char *sigla = (char *)malloc(4);  
    sigla[0] = a;  
    sigla[1] = delim;  
    sigla[2] = b;  
    sigla[3] = '\0';  
    return sigla;
```

```
}
```

```
int main(void)
```

```
{
```

```
    char a='A', b='Z';
```

```
    char *s;
```

```
    printf("Digite o delimitador: ");
```

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

```
s = concat(a,b);
```

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

```
free(s);
```

```
return 0;
```

variável
global

delim

‘,’

Memória livre

variáveis
lokais à
função
main

s

?

b

‘Z’

a

‘A’

não há
espaço
alocado
dinamic.

Pilha

```
/* exemplo do uso da memória */
```

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

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

```
char delim = ',';
```

```
char *concat (char x, char y)
```

```
{
```

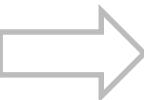
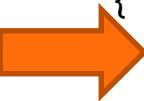
```
    char *sigla = (char *)malloc(4);  
    sigla[0] = x;  
    sigla[1] = delim;  
    sigla[2] = y;  
    sigla[3] = '\0';  
    return sigla;
```

```
}
```

```
int main(void)
```

```
{
```

```
    char a='A', b='Z';  
    char *s;  
    printf("Digite o delimitador: ");  
    scanf(" %c", &delim);  
    s = concat(a,b);  
    printf("%s\n", s);  
    free(s);  
    return 0;
```



variável global

delim

'.'

não há espaço alocado dinamic.

variáveis locais à função concat

sigla

?

y

'Z'

x

'A'

variáveis locais à função main

s

?

b

'Z'

a

'A'

Pilha

```
/* exemplo do uso da memória */
```

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

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

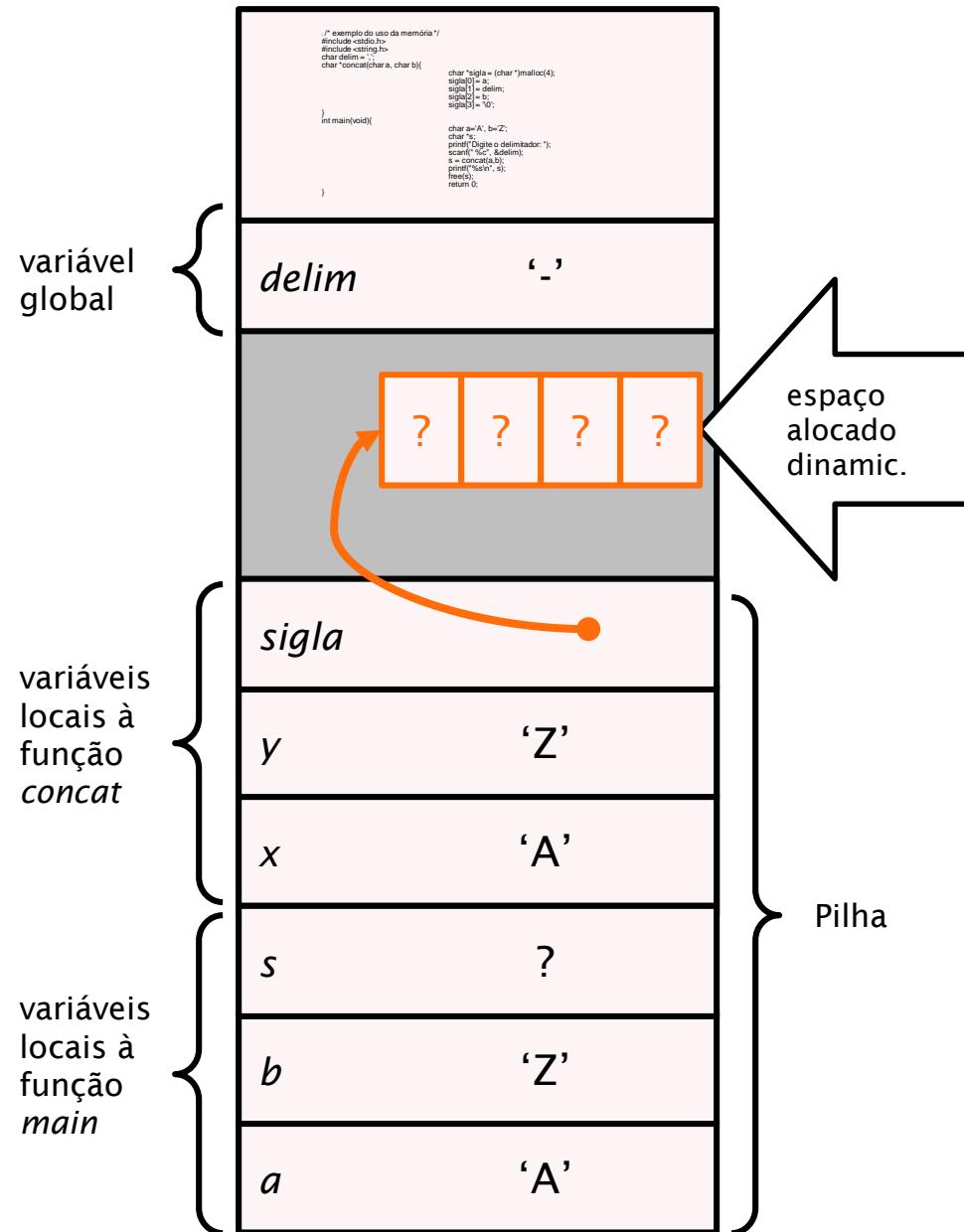
```
char delim = ',';
```

```
char *concat (char x, char y)
```

```
{  
    char *sigla = (char *)malloc(4);  
    sigla[0] = x;  
    sigla[1] = delim;  
    sigla[2] = y;  
    sigla[3] = '\0';  
    return sigla;  
}
```

```
int main(void)
```

```
{  
    char a='A', b='Z';  
    char *s;  
    printf("Digite o delimitador: ");  
    scanf(" %c", &delim);  
    s = concat(a,b);  
    printf("%s\n", s);  
    free(s);  
    return 0;  
}
```



```
/* exemplo do uso da memória */
```

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

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

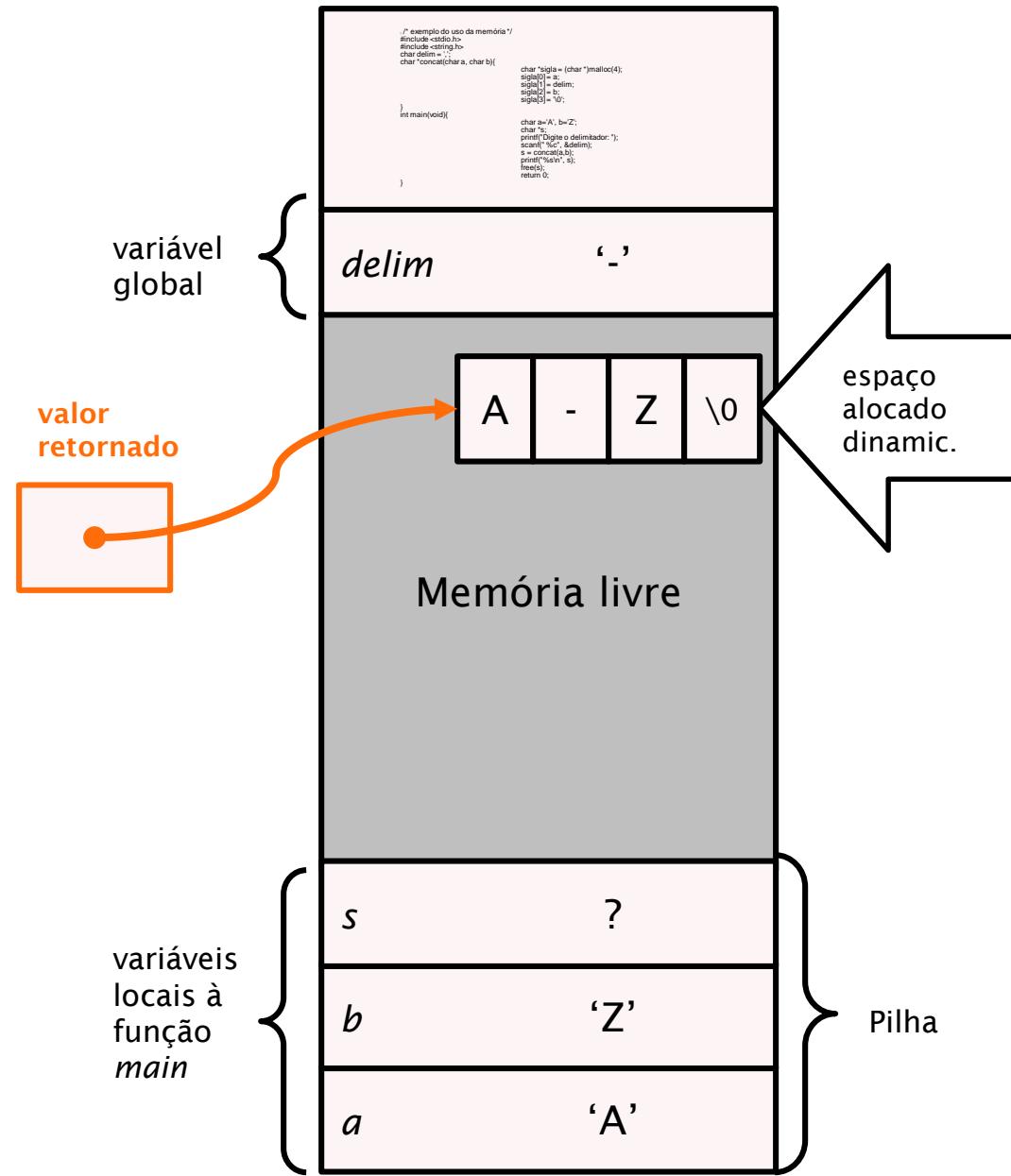
```
char delim = ',';
```

```
char *concat(char x, char y)
```

```
{  
    char *sigla = (char *)malloc(4);  
    sigla[0] = x;  
    sigla[1] = delim;  
    sigla[2] = y;  
    sigla[3] = '\0';  
    return sigla;  
}
```

```
int main(void)
```

```
{  
    char a='A', b='Z';  
    char *s;  
    printf("Digite o delimitador: ");  
    scanf(" %c", &delim);  
    s = concat(a,b);  
    printf("%s\n", s);  
    free(s);  
    return 0;  
}
```



```
/* exemplo do uso da memória */
```

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

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

```
char delim = ',';
```

```
char *concat(char x, char y)
```

```
{
```

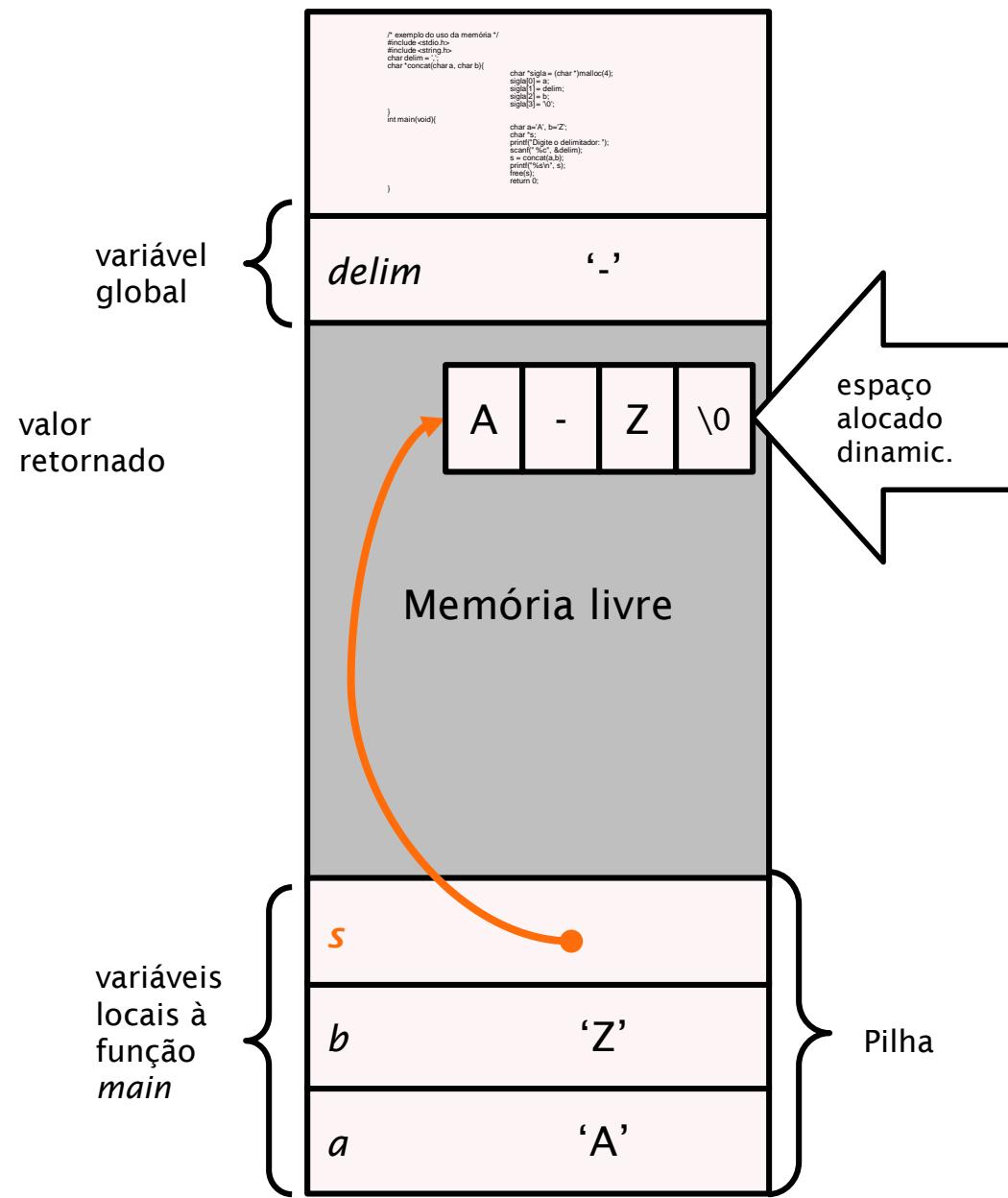
```
    char *sigla = (char *)malloc(4);  
    sigla[0] = x;  
    sigla[1] = delim;  
    sigla[2] = y;  
    sigla[3] = '\0';  
    return sigla;
```

```
}
```

```
int main(void)
```

```
{
```

```
    char a='A', b='Z';  
    char *s;  
    printf("Digite o delimitador: ");  
    scanf(" %c", &delim);  
    s = concat(a,b);  
    printf("%s\n", s);  
    free(s);  
    return 0;
```



```
/* exemplo do uso da memória */
```

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

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

```
char delim = ',';
```

```
char *concat(char x, char y)
```

```
{
```

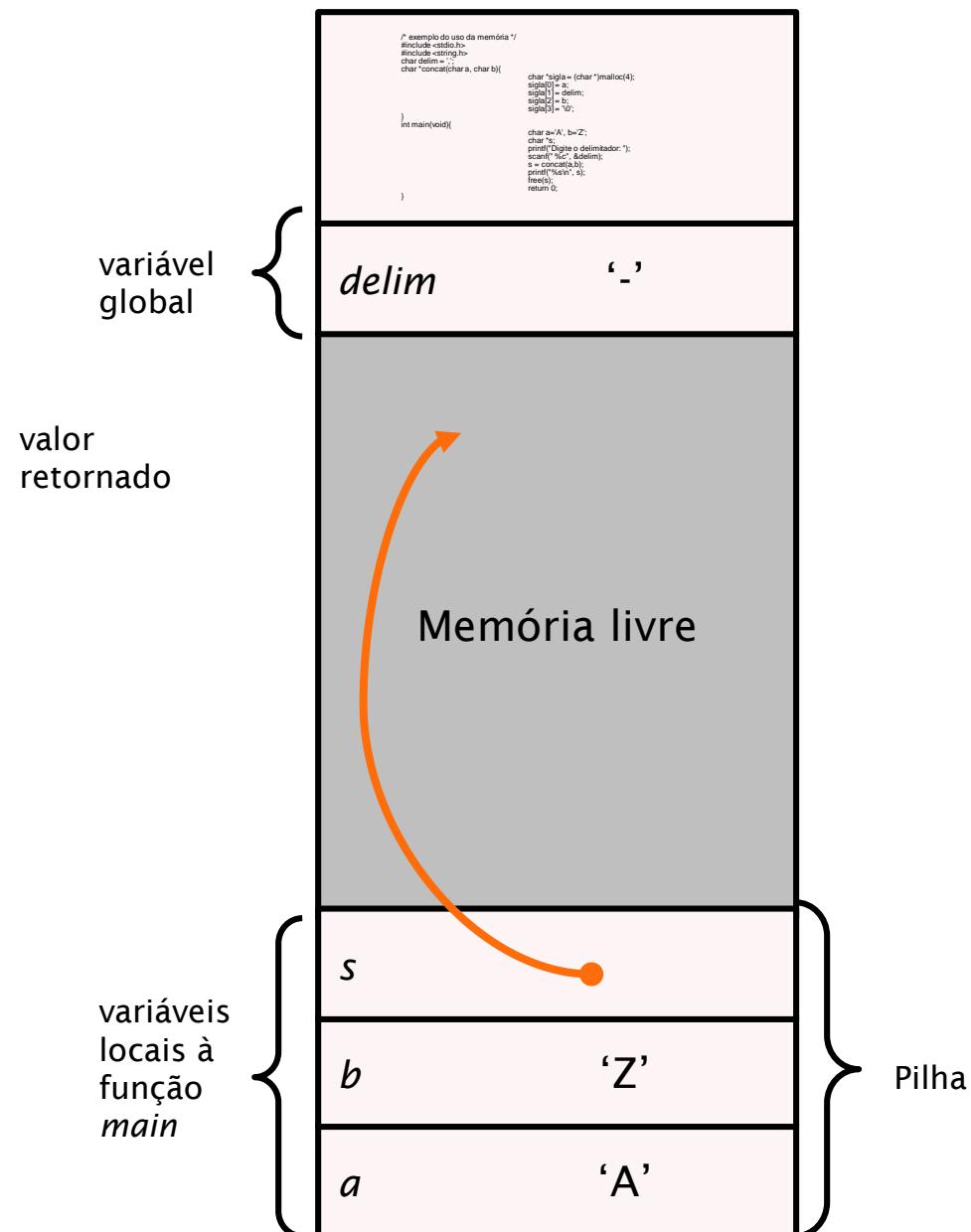
```
    char *sigla = (char *)malloc(4);  
    sigla[0] = x;  
    sigla[1] = delim;  
    sigla[2] = y;  
    sigla[3] = '\0';  
    return sigla;
```

```
}
```

```
int main(void)
```

```
{
```

```
    char a='A', b='Z';  
    char *s;  
    printf("Digite o delimitador: ");  
    scanf(" %c", &delim);  
    s = concat(a,b);  
    printf("%s\n", s);  
    free(s);  
    return 0;
```



```
/* exemplo do uso da memória */
```

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

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

```
char delim = ',';
```

```
char *concat(char x, char y)
```

```
{
```

```
    char *sigla = (char *)malloc(4);
```

```
    sigla[0] = x;
```

```
    sigla[1] = delim;
```

```
    sigla[2] = y;
```

```
    sigla[3] = '\0';
```

```
    return sigla;
```

```
}
```

```
int main(void)
```

```
{
```

```
    char a='A', b='Z';
```

```
    char *s;
```

```
    printf("Digite o delimitador: ");
```

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

```
    s = concat(a,b);
```

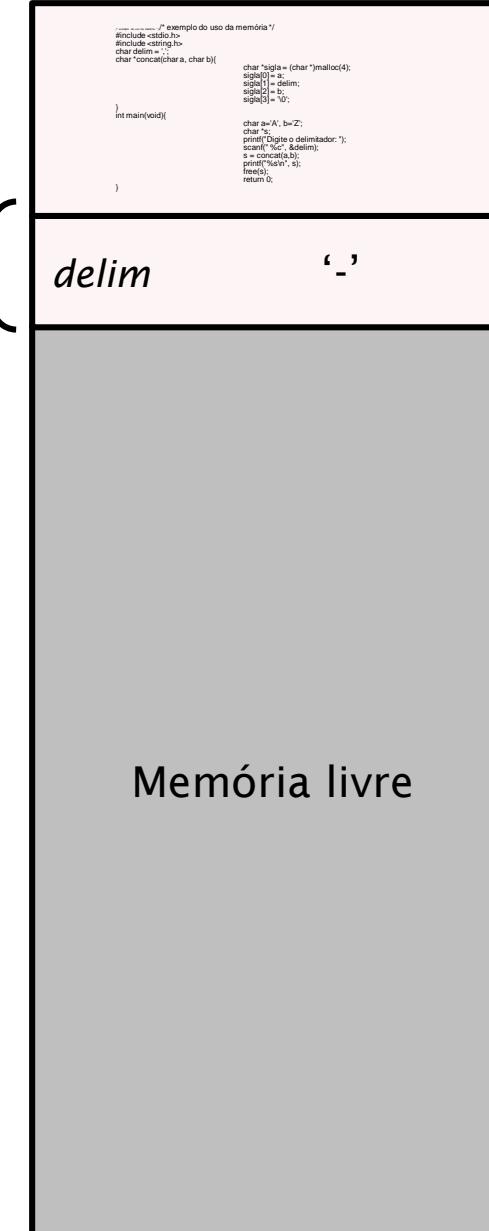
```
    printf("%s\n", s);
```

```
    free(s);
```

```
    return 0;
```

variável
global

valor
retornado



controle de fluxo revisão

Condicionais if e ?

```
if ( a > b ) —> condicao é avaliada  
    maximo = a; —> comando executado se condição for verdadeira  
else  
    maximo = b; —> comando executado se condição for falsa
```

se condicao for verdadeira, expressao1 é avaliada

```
condicao ? expressao1 : expressao2;
```

condicao é avaliada

caso contrário, expressao2 é avaliada

```
maximo = a > b ? a : b ;
```

Construções com laços

```
while ( expressao )
{
    bloco de comandos
}
```

- 1) enquanto expressao for verdadeira, executa bloco de comandos

```
do
{
    bloco de comandos
} while (expressao);
```

- 1) executa bloco de comandos
- 2) se expressao for verdadeira, repete

```
for (expr_inicial; expr_teste_laco; expr_alteracao)
{
    bloco de comandos
}
```

- 1) avalia expr_inicial
- 2) enquanto expr_teste_laco for verdadeira:
 - 2.1) executa bloco de comandos
 - 2.2) executa expr_alteracao

```
expr_inicial;
while (expr_teste_laco)
{
    bloco de comandos
    ...
    expr_alteração;
}
```

Laços – interrupção com break

```
#include <stdio.h>
int main (void)
{
    int i;
    for (i = 0; i < 10; i++) {
        if (i == 5)
            break;
        printf("%d ", i);
    }
    printf("fim\n");
    return 0;
}
```

Qual é a saída do programa?

0 1 2 3 4 fim

Laços – interrupção com continue

```
#include <stdio.h>
int main (void)
{
    int i;
    for (i = 0; i < 10; i++) {
        if (i == 5)
            continue;
        printf("%d ", i);
    }
    printf("fim\n");
    return 0;
}
```

Qual é a saída do programa?

0 1 2 3 4 6 7 8 9 fim

“pula” o 5

Laços – Cuidado com loops eternos!

```
#include <stdio.h>
int main (void)
{
    int i = 0;
    while (i < 10) {
        printf("%d ", i);
        if (i == 5)
            continue;
        i++;
    }
    printf("fim\n");
    return 0;
}
```

Qual é a saída do programa?

0 1 2 3 4 5 5 5 5 5 5 5 5 5 ...

nunca
termina

Seleção – comando switch

seleciona um dentre vários casos
(op_k deve ser um inteiro ou caractere)

```
switch ( expr )
{
    case op1: bloco de comandos 1; break;
    case op2: bloco de comandos 2; break;
    ...
    default:   bloco de comandos default; break;
}
```

```
/* calculadora de quatro operações */
#include <stdio.h>

int main (void)
{
    float num1, num2;
    char op;
    printf("Digite uma expressao: numero operador numero\n");
    scanf ("%f %c %f", &num1, &op, &num2);
    switch (op)
    {
        case '+':      printf(" = %f\n", num1+num2); break;
        case '-':      printf(" = %f\n", num1-num2); break;
        case '*':      printf(" = %f\n", num1*num2); break;
        case '/':      printf(" = %f\n", num1/num2); break;
        default:       printf("Operador invalido!\n"); break;
    }
    return 0;
}
```

funções revisão

Forma geral para definir funções

tipo_retornado nome_da_função (lista de parâmetros...)

{

corpo_da_função

}

Exemplo:

```
float converte (float c)
```

{

```
    float f;  
    f = 1.8*c + 32;  
    return f;
```

}

tipo retornado

nome da função

parâmetro

corpo da função

retorno do valor

Protótipo

Funções declaradas **antes** de serem invocadas: não precisam de protótipo

```
float converte (float c)
{
    ...
}

int main (void)
{
    ...
    t2 = converte(t1);
    ...
}
```

funções declaradas **após** serem invocadas: precisam de protótipo

```
/* COM ponto-e-vírgula no final*/
float converte (float c);

int main (void)
{
    ... t2 =
        converte(t1);
    }

/* SEM ponto-e-vírgula no final*/
float converte (float c)
{
    ...
}
```

Exemplo de programa – Cálculo de Fatorial

```
#include <stdio.h>

long fat (int n); /* protótipo da função */
int main (void)      /* função principal */
{
    int n; long r;
    printf("Digite um número não negativo:");
    scanf("%d", &n);
    r = fat(n); /* chamada da função */
    printf("Fatorial = %d\n", r);
    return 0;     /* retorno da main: 0 = execução OK */
}

/* função para calcular o valor do fatorial */
long fat (int n)      /* declaração da função */
{
    int i;
    long f = 1;
    for (i = 1; i <= n; i++)
        f *= i;
    return f;
}
```

Pilha de Execução

comunicação entre funções

funções são independentes entre si

transferência de dados entre funções é feita através de

parâmetros (passagem por **valor**)

valor de retorno da função chamada

variáveis **locais** a uma função:

definidas dentro do corpo da função (incluindo os parâmetros)

não existem fora da função

são criadas cada vez que a função é executada

deixam de existir quando a execução da função terminar

Pilha de execução – Fatorial

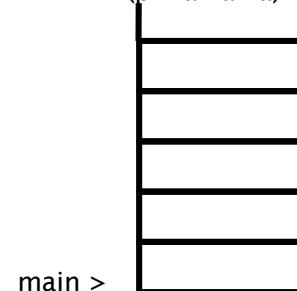
```
#include <stdio.h>

long fat (int n); /* protótipo da função */

int main (void) /* função principal */
{
    int n; long r;
    printf("Digite um número não negativo:");
    scanf("%d", &n);
    r = fat(n); /* chamada da função */
    printf("Fatorial = %d\n", r);
    return 0; /* retorno da main: 0 = execução OK */
}

/* função para calcular o valor do fatorial */
long fat (int n) /* declaração da função */
{
    int i;
    long f = 1;
    for (i = 1; i <= n; i++)
        f *= i;
    return f;
}
```

1. início do programa
(pilha vazia)



main >

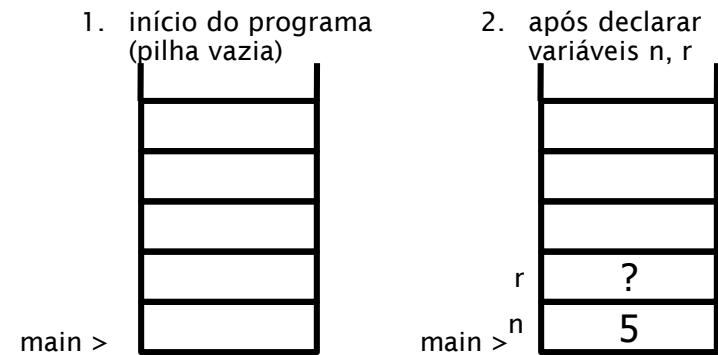
Pilha de execução – Fatorial

```
#include <stdio.h>

long fat (int n); /* protótipo da função */

int main (void) /* função principal */
{
    int n; long r;
    printf("Digite um número não negativo:");
    scanf("%d", &n);
    r = fat(n); /* chamada da função */
    printf("Fatorial = %d\n", r);
    return 0; /* retorno da main: 0 = execução OK */
}

/* função para calcular o valor do fatorial */
long fat (int n) /* declaração da função */
{
    int i;
    long f = 1;
    for (i = 1; i <= n; i++)
        f *= i;
    return f;
}
```



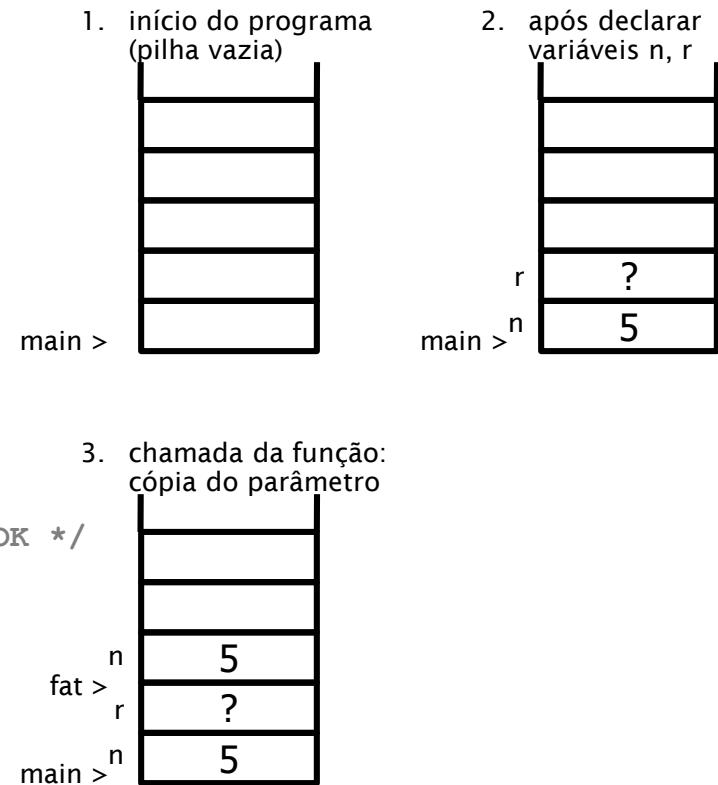
Pilha de execução – Fatorial

```
#include <stdio.h>

long fat (int n); /* protótipo da função */

int main (void) /* função principal */
{
    int n; long r;
    printf("Digite um número não negativo:");
    scanf("%d", &n);
    r = fat(n); /* chamada da função */
    printf("Fatorial = %d\n", r);
    return 0; /* retorno da main: 0 = execução OK */
}

/* função para calcular o valor do fatorial */
long fat (int n) /* declaração da função */
{
    int i;
    long f = 1;
    for (i = 1; i <= n; i++)
        f *= i;
    return f;
}
```



Pilha de execução – Fatorial

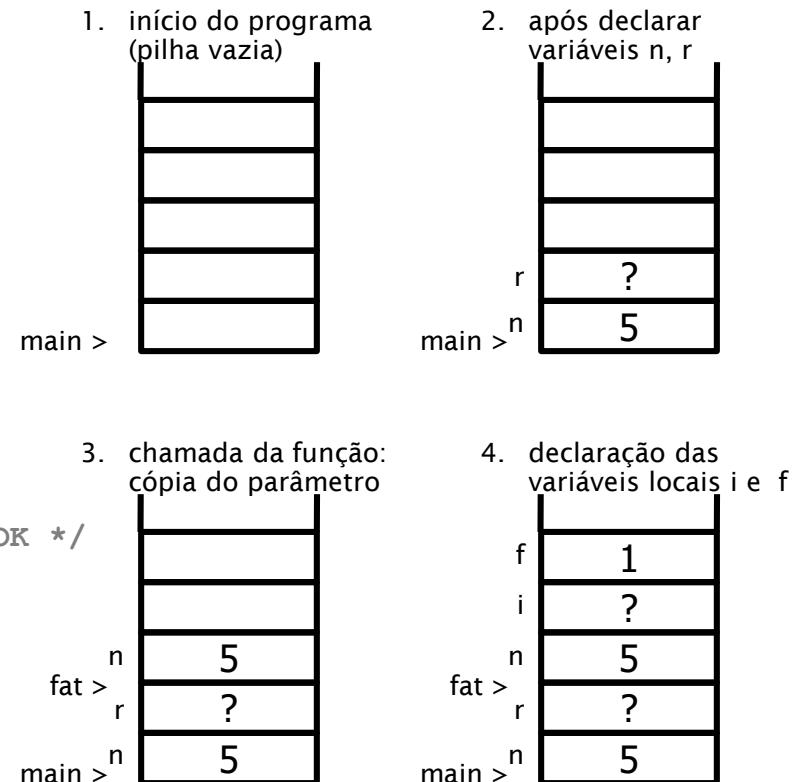
```
#include <stdio.h>

long fat (int n); /* protótipo da função */

int main (void) /* função principal */
{
    int n; long r;
    printf("Digite um número não negativo:");
    scanf("%d", &n);
    r = fat(n); /* chamada da função */
    printf("Fatorial = %d\n", r);
    return 0; /* retorno da main: 0 = execução OK */
}

/* função para calcular o valor do fatorial */
long fat (int n) /* declaração da função */
{
    int i;
    long f = 1;
    for (i = 1; i <= n; i++)
        f *= i;
    return f;
}
```

Diagrama de setas amarelo e laranja apontando para a chamada de função `r = fat(n);` e para a definição da função `long fat (int n)`.



Pilha de execução – Fatorial

```
#include <stdio.h>

long fat (int n); /* protótipo da função */

int main (void) /* função principal */
{
    int n; long r;
    printf("Digite um número não negativo:");
    scanf("%d", &n);
    r = fat(n); /* chamada da função */
    printf("Fatorial = %d\n", r);
    return 0; /* retorno da main: 0 = execução OK */
}

/* função para calcular o valor do fatorial */
long fat (int n) /* declaração da função */
{
    int i;
    long f = 1;
    for (i = 1; i <= n; i++)
        f *= i;
    return f;
}
```

The diagram illustrates the state of the execution stack across five stages:

1. **início do programa (pilha vazia)**: The stack is empty.
2. **após declarar variáveis n, r**: The stack shows local variables *n* and *r*. *n* contains 5, and *r* contains a question mark (?) because it has not been initialized.
3. **chamada da função: cópia do parâmetro**: The stack shows the call frame for *fat*. It contains local variable *n* with value 5, and parameter *n* with value 5. The previous frame for *main* is still present.
4. **declaração das variáveis locais i e f**: The stack shows the *fat* frame with local variable *i* containing a question mark (?) and local variable *f* containing 1. The *main* frame is still present.
5. **final do laço**: The stack shows the *fat* frame after the loop has completed. Local variable *f* contains 120, local variable *i* contains 6, and local variable *n* contains 0. The *main* frame is still present.

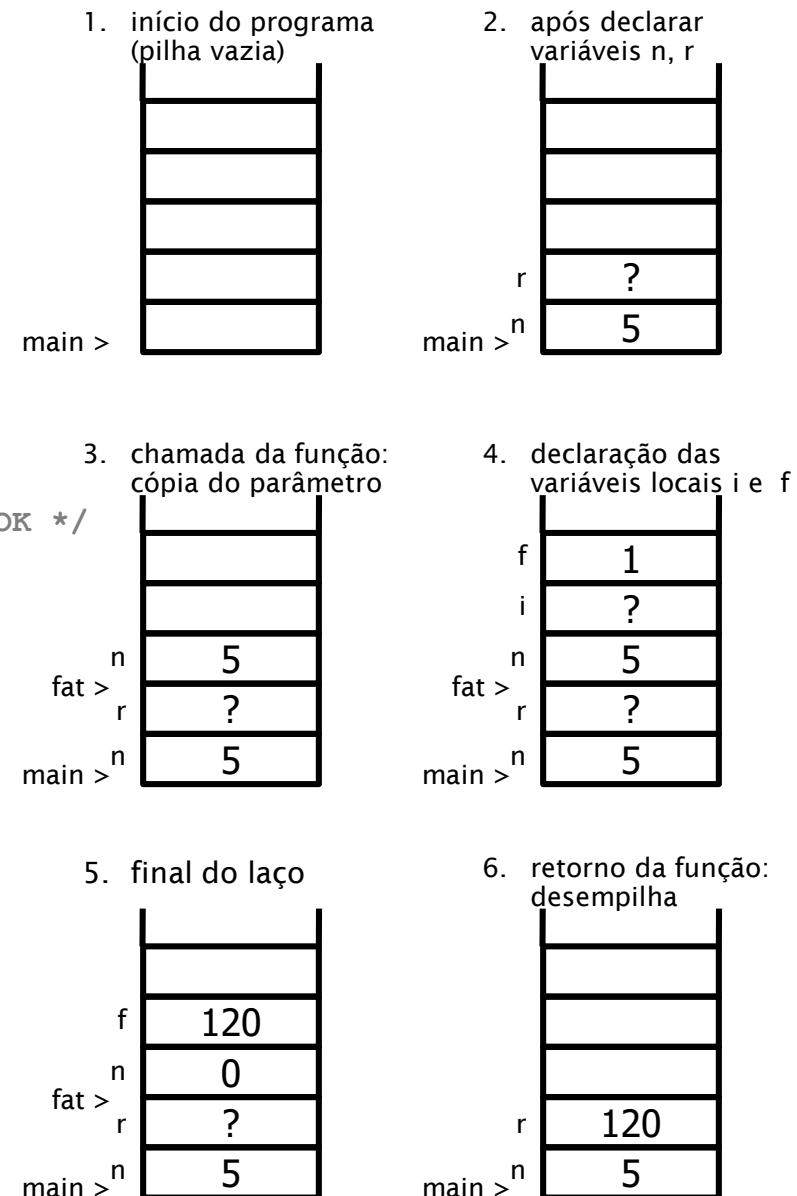
The stack is represented as a vertical column of horizontal bars. Each bar represents a memory location. The stack grows downwards. The diagram shows the state of the stack at five different points in the program's execution. At each point, the stack contains the current call frame (a stack frame) and the previous call frame (the caller's stack frame). The stack frames are labeled with their respective function names (*main*, *fat*) and parameters/variables (*n*, *r*, *i*, *f*). The values of the variables are indicated by the position of the stack pointer or explicitly labeled within the stack frame. A question mark (?) indicates an undefined or uninitialized value.

Pilha de execução – Fatorial

```
#include <stdio.h>

long fat (int n); /* protótipo da função */
int main (void) /* função principal */
{
    int n; long r;
    printf("Digite um número não negativo:");
    scanf("%d", &n);
    r = fat(n); /* chamada da função */
    printf("Fatorial = %d\n", r);
    return 0; /* retorno da main: 0 = execução OK */
}

/* função para calcular o valor do fatorial */
long fat (int n) /* declaração da função */
{
    int i;
    long f = 1;
    for (i = 1; i <= n; i++)
        f *= i;
    return f;
}
```



Funções Extern e Static

Por padrão as funções são sempre declaradas como extern, sendo visíveis por todo o código.

Se declaradas como static, as funções só são visíveis no arquivo onde foram definidas.

Exemplo de Função Estática

main.c

```
#include "header.h"

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

function.c

```
#include "header.h"

void hello()
{
    printf("HELLO WORLD\n");
}
```

header.h

```
#include <stdio.h>

static void hello();
```

```
$gcc main.c function.c
header.h:4: warning: "hello" used but never defined
/tmp/ccahx5Ic.o: In function `main':
main.c:(.text+0x12): undefined reference to `hello'
collect2: ld returned 1 exit status
```

dúvidas?

ponteiros

revisão

Ponteiro

tipo que armazena um endereço de memória

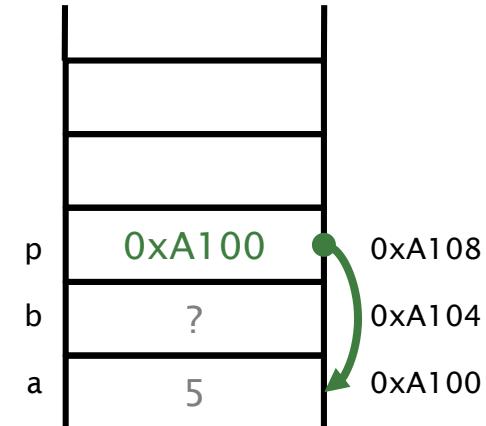
```
/* variáveis do tipo inteiro */
```

```
int a=5, b;
```

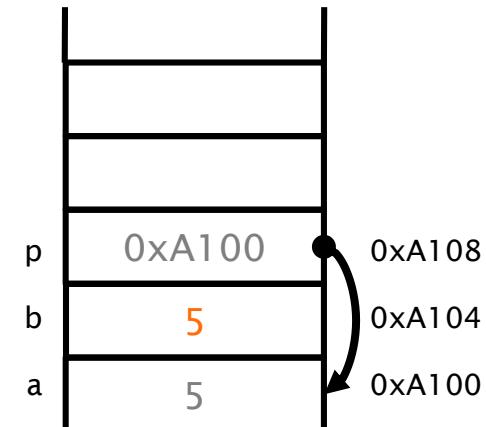
```
/* variável do tipo ponteiro para inteiro */
```

```
int *p;
```

1 → *p = &a; /* atribui a p o endereço de a */*

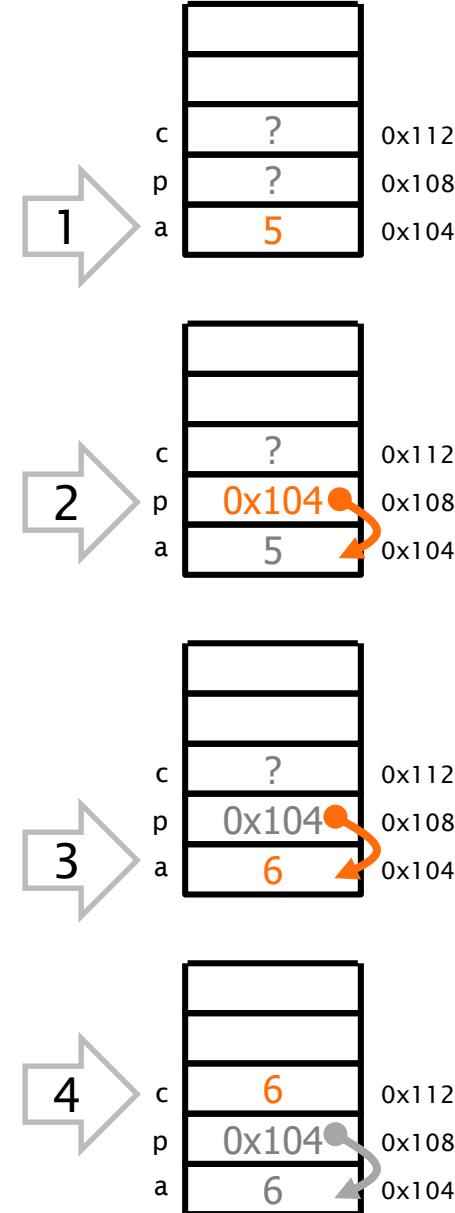


2 → *b = *p; /* atribui a b o conteúdo de p */*



Ponteiro (cont.)

```
int a;  
int *p;  
int c;  
/* a recebe o valor 5 */  
1 a = 5;  
/* p recebe o endereço de a ou seja, p aponta para a */  
2 p = &a;  
/* posição de memória apontada por p recebe 6 */  
3 *p = 6;  
/* c recebe o valor armazenado na posição de memória  
apontada por p */  
4 c = *p;
```



Exemplo

```
int main ( void )
{
    int a=5;
    int *p;
    p = &a;
    *p = 2;
    (*p)++;
    printf(" %d ", a);
    return 0;
}
```

Qual é a saída do programa?

3

Cuidado!

```
int main ( void )  
{  
    int a;  
    int *p;  
    *p = 2; // p não foi inicializado; onde 2 é armazenado?  
    printf(" %d ", a); // a não foi inicializado; qual é o seu valor?  
    return 0;  
}
```

Qual é a saída do programa?

Uma função em C pode retornar dois valores?

NÃO

Mas pode receber ponteiros para valores a serem modificados

Função para calcular soma e produto

A

```
/* passagem de parâmetros por valor */
void somaprod(int a, int b, int soma, int prod)
{ soma = a+b; prod=a*b }

/* o que acontece? */
int main(void) {
    int s, p;
    somaprod (3,5,s,p);
    printf("soma: %d, produto: %d\n", s, p);
}
```

soma: -3879, produto: -29047
(lixo) (lixo)

Função para calcular soma e produto

A

```
/* passagem de parâmetros por valor */  
void somaprod(int a, int b, int soma, int prod)  
{ soma = a+b; prod=a*b }
```

```
/* o que acontece? */  
int main(void) {  
    int s, p;  
    somaprod (3,5,s,p);  
    printf("soma: %d, produto: %d\n", s, p);  
}
```

soma: -3879, produto: -29047
(lixo) (lixo)

B

```
/* passagem de parâmetros por referência */  
void somaprod(int a, int b, int *psoma, int *pprod)  
{ *psoma = a+b; *pprod=a*b; }
```

```
/* o que acontece? */  
int main(void) {  
    int s, p;  
    somaprod (3,5,&s,&p);  
    printf("soma: %d, produto: %d\n", s, p);  
}
```

soma: 8, produto: 15

Exemplo de programa

```
#include <stdio.h> /* declaração de ponteiros */

void somaprod (int a, int b, int *ps, int *pp)
{
    *ps = a + b;      /* altera o conteúdo do ponteiro */
    *pp = a * b;
}

int main (void)
{
    int x, y, s, p;
    printf("Digite dois numeros:");
    scanf("%d%d", &x, &y);
    somaprod(x, y, &s, &p); /* endereço das variáveis */
    printf("Soma: %d, produto: %d \n", s, p);
    return 0;
}
```

Ponteiros – Revisão

Considerando

```
int x;  
int *pX = &x;
```

Se eu quiser ler o valor de x do teclado, como posso fazer?

```
scanf("%d", x);  
scanf("%d", &x);  
scanf("%d", pX);  
scanf("%d", &pX);
```

Ponteiros – Revisão

Considerando

```
int x;  
int *pX = &x;
```

Se eu quiser ler o valor de x do teclado, como posso fazer?

~~scanf("%d", x);~~
~~scanf("%d", &x);~~
scanf("%d", pX);
~~scanf("%d", &pX);~~

Por quê?

dúvidas?

ponteiros para funções

Problema: funções que dependem do tipo do dado

```
/* busca binária limitada a vetor de inteiros */
int compara (int a, int b)
{
    if (a<b) return -1;
    if (a==b) return 0;
    return 1;
}

int busca_binaria (int *v, int n, int elem)
{
    int ini = 0, fim = n-1;
    while (ini <= fim)
    {
        int meio = (ini + fim) / 2;
        int comparacao = compara(elem, v[meio]);
        switch(comparacao) {
            case 0: return meio; break;
            case -1: fim = meio - 1; break;
            case 1: ini = meio + 1; break;
        }
    }
    return -1;
}
```

Como fazer uma implementação de busca binária que sirva para qualquer tipo de dado?

Mudanças necessárias

```
/* busca binária */  
int compara(  
{  
    if (a < b) return -1;  
    if (a == b) return 0;  
    return 1;  
}
```

```
int busca_binaria(void *v, int n,  
                  void* elem, int tam,  
                  int (*compara)(const void*, const void*))  
{  
    int ini = 0, fim = n - 1;  
    while (ini <= fim)  
    {  
        int meio = (ini + fim) / 2;  
        int comparacao = compara(elem, acessa(v, meio, tam));  
        switch(comparacao) {
```

1 Em vez de um vetor com tipo declarado explicitamente, um vetor de ponteiros genéricos: **void *v**. Isso significa que a função **busca_binaria** não saberá qual é o tipo do dado que está buscando.

2

Em vez de um elemento com tipo declarado explicitamente, um ponteiro genérico para o elemento: **void *elem**.

4

Se a **busca_binaria** não conhece o tipo de dado, não sabe como comparar dois valores. Precisa receber como parâmetro um **ponteiro para a função compara**. Essa função conhece o tipo do dado e sabe como comparar dois valores desse tipo. O modificador de tipo **const** é usado para garantir que a função não modificará os valores dos elementos (devem ser tratados como constantes)

3 Se a função não conhece o tipo do dado, não sabe como acessar cada elemento do vetor. Precisa de uma função de acesso ao elemento na posição desejada: **acessa(v, meio, tam)**. A função **busca_binaria** tem que passar para essa função **acessa** qual é o tamanho do dado, e portanto precisa receber esse tamanho também como parâmetro: **int tam**.

Mudanças necessárias na função busca_binaria

```
/* busca binária genérica */

int busca_binaria ( void *v,
                    int n,
                    void *elem,
                    int tam,
                    int (*compara)(const void*, const void*) )
{
    int ini = 0, fim = n-1;
    while (ini <= fim)
    {
        int meio = (ini + fim) / 2;
        int comparacao = compara(elem, acessa(v, meio, tam));
        switch (comparacao) {
            case 0: return meio; break;
            case -1: fim = meio - 1; break;
            case 1: ini = meio + 1; break;
        }
    }
    return -1;
}
```

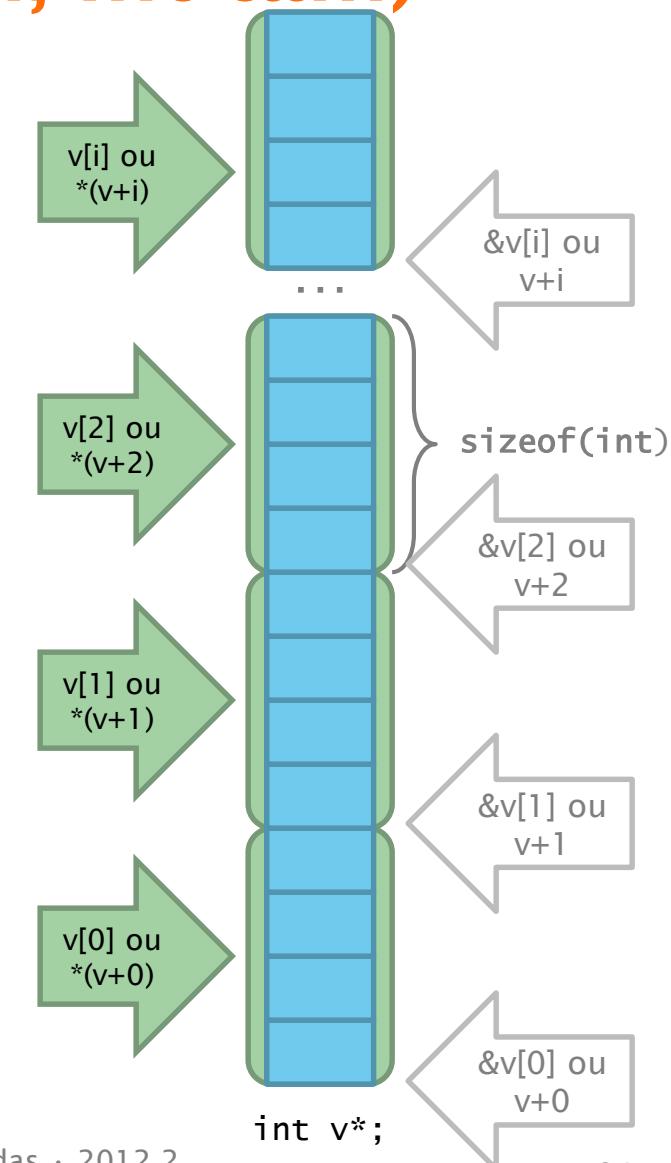
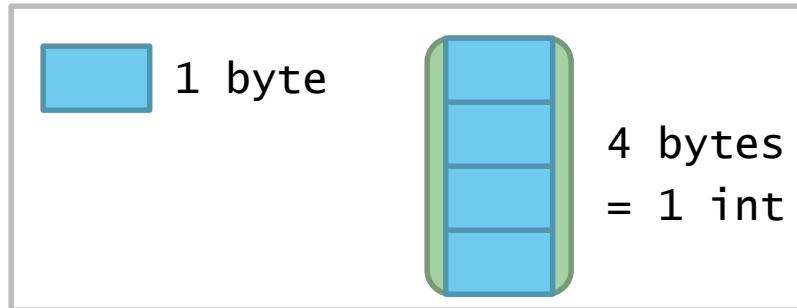
Função

void* acessa (void* v, int i, int tam)

lembmando:

Se declaro `int *v`, posso usar `v[i]`,
porque o compilador sabe que
deve percorrer o vetor de 4 em
4 bytes (`sizeof(int)` é 4).

Nesse caso, `v+i` é o endereço
do i-ésimo elemento de v

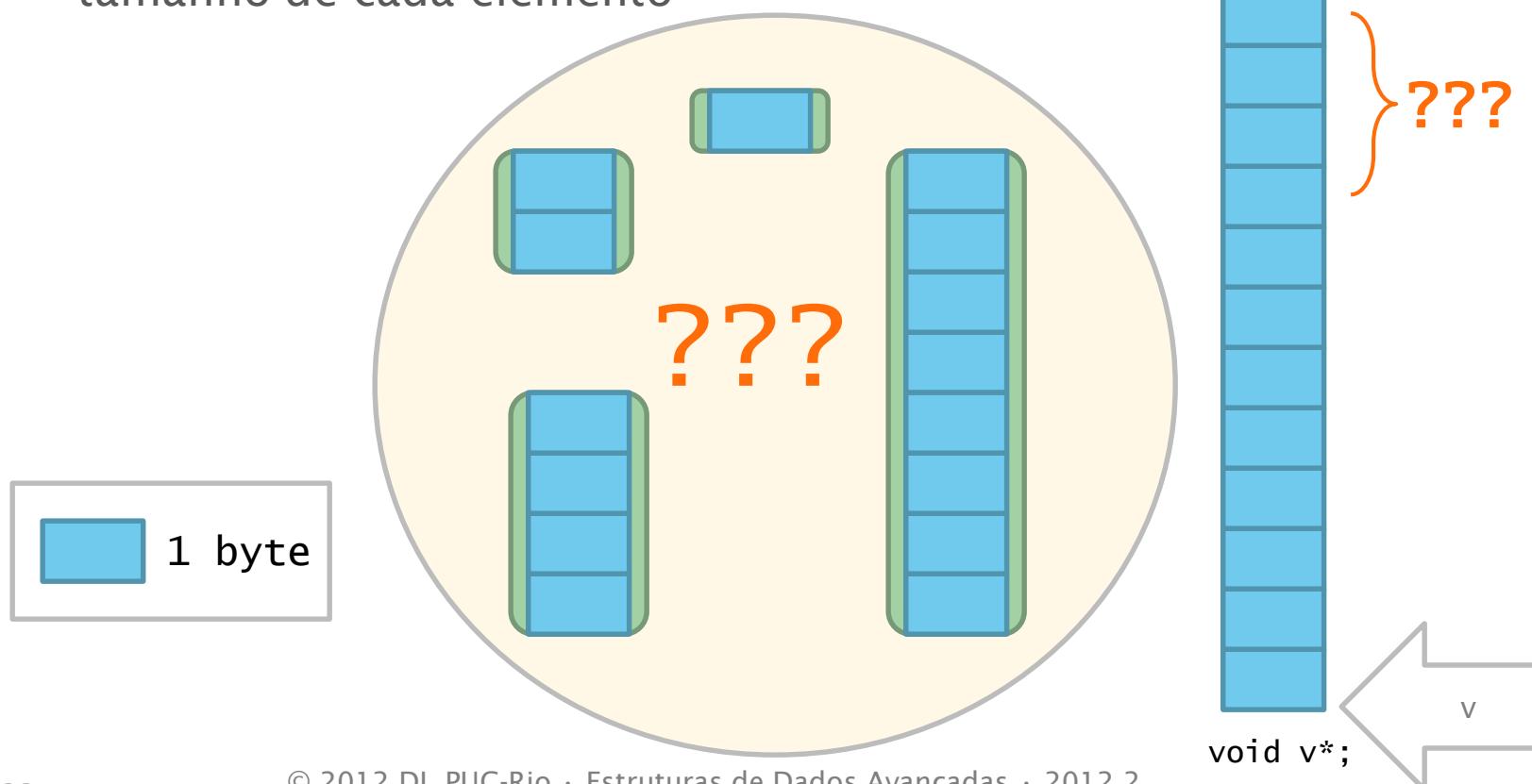


Função

void* acessa (void* v, int i, int tam)

lembmando:

Se declaro `void* v`, não posso
usar `v[i]`, pois não sei qual é o
tamanho de cada elemento



Função

void* acessa (void* v, int i, int tam)

Sabendo o tamanho **tam** de cada elemento do vetor...

o i -ésimo elemento encontra-se no endereço **$v+i*tam$**

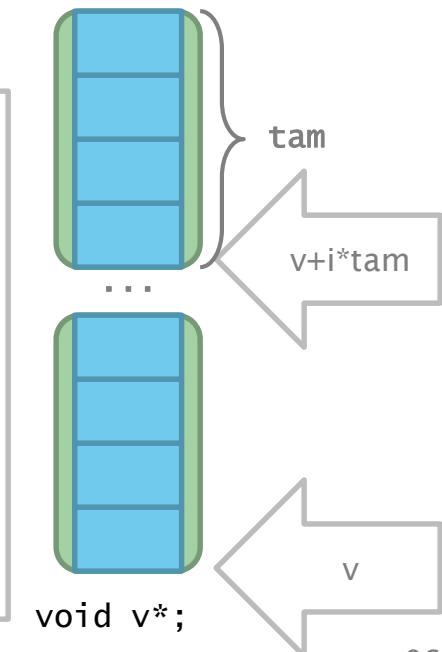
Problema:

Não posso incrementar o endereço de um ponteiro genérico **void***

Como andar de byte em byte?

Usando **char*** (lembrando: `sizeof(char)=1`)

```
void* acessa (void* v, int i, int tam)
{
    /* trata como vetor de char */
    char* t = (char*)v;
    /* obtém o endereço do elemento desejado */
    t += i*tam;
    /* retorna o endereço como ponteiro genérico */
    return (void*)t;
}
```



Função

int compara_X(const void *pa, const void *pb)

Deve retornar:

- { -1 se o elemento em *pa* precede o elemento em *pb*
- 0 se os elementos em *pa* e *pb* são iguais
- 1 se o elemento em *pa* sucede o elemento em *pb*

Se *pa* é genérico, não sei obter seu conteúdo (~~*pa~~)

1. cast para um ponteiro de um tipo conhecido (*int**)*pa*
2. obtém o conteúdo do endereço apontado: **((int*)pa)*
3. efetua a comparação: **((int*)pa) < *((int*)pb)*

```
int compara_int(const void* pa, const void* pb) {  
    int *pia = (int *)pa;  
    int *pib = (int *)pb;  
    int a = *pia;  
    int b = *pib;  
    if (a < b) return -1;  
    if (a > b) return 1;  
    return 0;  
}
```

Algumas versões alternativas da função **int compara_int(const void *pa, const void *pb)**

```
int compara_int(const void* pa, const void* pb) {  
    int *pia = (int *)pa;  
    int *pib = (int *)pb;  
    if (*pia < *pib) return -1;  
    if (*pia > *pib) return 1;  
    return 0;  
}
```

```
int compara_int(const void* pa, const void* pb) {  
    int *pia = (int *)pa;  
    int *pib = (int *)pb;  
    int a = *pia;  
    int b = *pib;  
    return (a < b) ? -1 : (a == b) ? 0 : 1;  
}
```

```
int compara_int(const void* pa, const void* pb) {  
    return (*((int*)pa)<*((int*)pb))?-1:(*((int*)pa)>*((int*)pb))?1:0;  
}
```

Para buscar o inteiro *elem* num vetor *v* de *n* inteiros:

```
res = busca_binaria(v, n, &elem, sizeof(int), compara_int);
```

Outras funções de comparação

```
static int compara_float(const void* pa, const void* pb) {  
    return (*((float *)pa) < *((float *)pb)) ? -1 :  
           (*((float *)pa) > *((float *)pb)) ? 1 : 0;  
}
```

```
static int compara_cadeia(const void* pa, const void* pb) {  
    char **psa = (char **)pa;  
    char **psb = (char **)pb;  
    return strcmp(*psa, *psb);  
}
```

Outras funções de comparação: ponteiro para estrutura

```
/* considerando a seguinte estrutura */

typedef struct aluno {
    int matricula;
    char nome[50];
} Aluno;

static int compara_aluno_nome (const void* pa, const void* pb) {
    Aluno** ppaa = (Aluno**) pa;
    Aluno** ppab = (Aluno**) pb;
    return strcmp((*ppaa)->nome, (*ppab)->nome);
}

static int compara_aluno_matricula (const void* pa, const void *pb) {
    Aluno** ppaa = (Aluno**) pa;
    Aluno** ppab = (Aluno**) pb;
    int mata = (*ppaa)->matricula;
    int matb = (*ppab)->matricula;
    return mata < matb? -1 : mata > matb ? 1 : 0;
}
```

função `qsort` da `stdlib`

Ordenação rápida: função qsort da biblioteca padrão

```
void qsort (void *v,  
           int n,  
           int tam,  
           int (*cmp)(const void*, const void*));
```

v: ponteiro (de tipo genérico, `void*`) para o primeiro elemento do vetor

n: número de elementos do vetor

tam: tamanho, em bytes, de cada elemento do vetor

cmp: ponteiro para a função de comparação, que deve retornar **-1**, **0**, ou **1**, se o primeiro elemento for **menor**, **igual**, ou **maior** que o segundo, respectivamente, de acordo com o critério de ordenação adotado

```
qsort(v, 8, sizeof(float), comp_reais);
```

Ordenação rápida – vetor de números reais

```
/* Ilustra uso do algoritmo qsort para vetor de float */
#include <stdio.h>
#include <stdlib.h>

/* função de comparação de reais */
static int comp_reais (const void* p1, const void* p2)
{
    /* converte para ponteiros de float */
    float *f1 = (float*)p1;
    float *f2 = (float*)p2;
    /* dados os ponteiros de float, faz a comparação */
    if (*f1 < *f2) return -1;
    else if (*f1 > *f2) return 1;
    else return 0;
}

/* ordenação de um vetor de float */
int main (void)
{
    int i;
    float v[8] = {25.6,48.3,37.7,12.1,57.4,86.6,33.3,92.8};
    qsort(v, 8, sizeof(float), comp_reais);
    printf("vetor ordenado: ");
    for (i=0; i<8; i++)
        printf("%g ",v[i]);
    printf("\n");
    return 0;
}
```

Ordenação rápida – vetor de ponteiros para Aluno (que devem ser ordenados pelo nome)

```
#include <string.h>
/* estrutura representando um aluno*/
typedef struct aluno {
    char nome[81]; /* chave de ordenação */
    int matricula;
} Aluno;
...
int main( )
{
    Aluno* v[100]; /* vetor de ponteiros para Aluno */
    Aluno a[5]={ {"Pedro", 1234}, {"Joao", 2345},
                 {"Maria", 3254}, {"Jose", 8102},
                 {"Felipe", 7253};
    v[0]=&a[0]; v[1]=&a[1]; v[2]=&a[2];
    v[3]=&a[3]; v[4]=&a[4];
    show_vet(5,v);
    qsort(v,5,sizeof(Aluno*),comp_alunos_nome);
    show_vet(5,v);
    return 0;
}
```

```
/* Função de comparação de Aluno por nome */
static int comp_alunos_nome
(const void* p1, const void* p2)
{
    Aluno **ppa1 = (Aluno**)p1;
    Aluno **ppa2 = (Aluno**)p2;
    return strcmp((*ppa1)->nome, (*ppa2)->nome);
}
```

Ordenação rápida – vetor de ponteiros para Aluno (que devem ser ordenados pelo nome)

```
#include <string.h>
/* estrutura representando um aluno*/
typedef struct aluno {
    char nome[81]; /* chave de ordenação */
    int matricula;
} Aluno;
...
int main( )
{
    Aluno* v[100]; /* vetor de ponteiros para Aluno */
    Aluno a[5]={ {"Pedro", 1234}, {"Joao", 2345},
                 {"Maria", 3254}, {"Jose", 8102},
                 {"Felipe", 7253};

    v[0]=&a[0]; v[1]=&a[1]; v[2]=&a[2];
    v[3]=&a[3]; v[4]=&a[4];
    show_vet(5,v);
    qsort(v,5,sizeof(Aluno*),comp_alunos_nome);
    show_vet(5,v);
    return 0;
}
```

Alunos:
Pedro, 1234
Joao, 2345
Maria, 3254
Jose, 8102
Felipe, 7253

Alunos:
Felipe, 7253
Joao, 2345
Jose, 8102
Maria, 3254
Pedro, 1234

Recursão

Definição recursiva

regra base

(definição de “objetos simples”)

$$1! = 1$$

passo indutivo

(definição de “objetos maiores” em termos de “objetos menores”)

$$n! = n \times (n-1)!$$

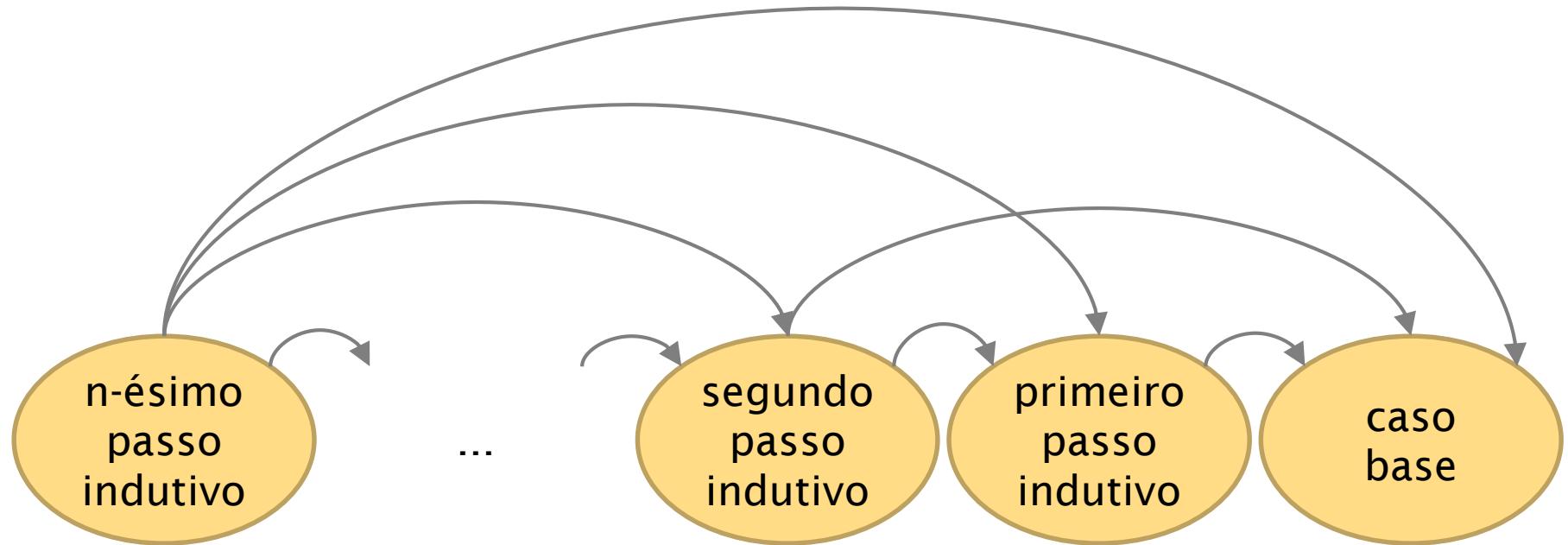
$$2! = 2 \times 1! = 2 \times 1 = 2$$

$$3! = 3 \times 2! = 3 \times 2 = 6$$

$$4! = 4 \times 3! = 4 \times 6 = 24$$

...

Definição recursiva



Exemplo de recursão: fatorial

$$n! = \begin{cases} 1, & \text{se } n \leq 1 \\ n \times (n - 1)!, & \text{caso contrário} \end{cases}$$

```
int fat (int n)
{
    if (n <= 1)
        return 1;
    else
        return n * fat(n-1);
}
```

Função recursiva – Fatorial

```
/* implementação não-recursiva (iterativa) */
unsigned long long factorial(unsigned int x)
{
    unsigned long long res = 1;
    while (x > 0)
        res *= x--;
    return res;
}
```

```
/* implementação recursiva */
unsigned long long factorial (unsigned int x)
{
    if (x <= 1)
        return 1;
    return x * factorial(x-1);
}
```

Fatorial

fat (4)

chama fat (3)

fat(3)

chama fat (2)

fat (2)

chama fat (1)

fat (1)

retorna 1

retorna $2 \times \text{fat}(1) = 2 \times 1 = 2$

retorna $3 \times \text{fat}(2) = 3 \times 2 = 6$

retorna $4 \times \text{fat}(3) = 4 \times 6 = 24$

```
double fat (int x)
{
    if (x <= 1)
        return 1;
    return x * fat (x-1);
}
```

Maior elemento de um vetor

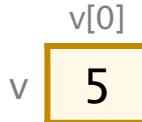
```
/* implementação não-recursiva  
(iterativa) */  
int maior_i (int v[], int n)  
{  
    int maior = v[0];  
    for (n--; n>0; n--)  
        if (v[n] > maior)  
            maior = v[n];  
    return maior;  
}
```

Maior elemento de um vetor

regra base

se um vetor tem tamanho 1, o maior elemento é o seu único elemento

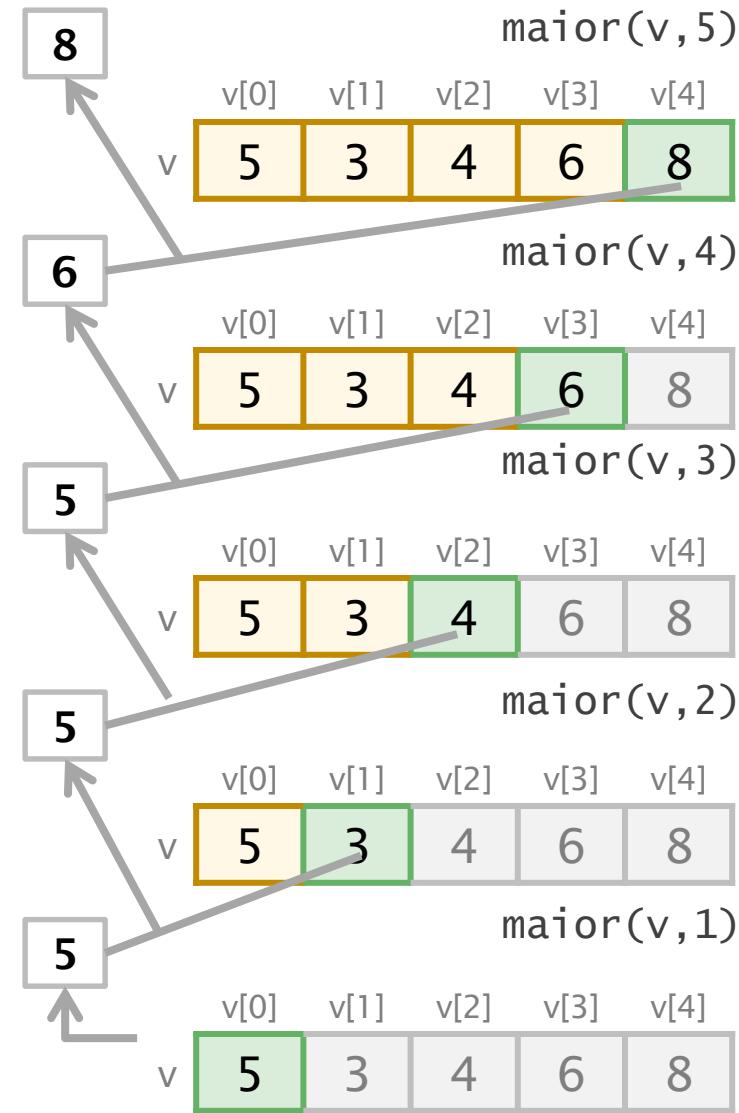
$$\text{maior}(v, 1) = v[0]$$



passo indutivo

o maior elemento do vetor é o maior entre o último elemento e o subvetor de comprimento $n-1$

$$\text{maior}(v, n) = \max(v[n-1], \text{maior}(v, n-1))$$



Maior elemento de um vetor

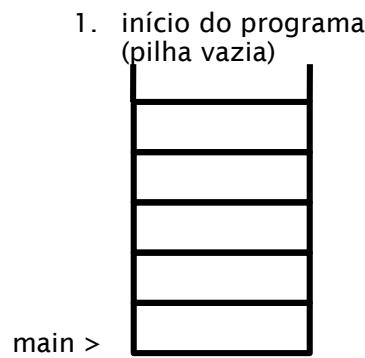
```
/* implementação não-recursiva  
(iterativa) */  
int maior_i (int v[], int n)  
{  
    int maior = v[0];  
    for (n--; n>0; n--)  
        if (v[n] > maior)  
            maior = v[n];  
    return maior;  
}
```

```
/* implementação recursiva */  
int maior_r (int v[], int n)  
{  
    int maior;  
    if (n == 1)  
        return v[0];  
    maior = maior_r(v, n-1);  
    if (v[n-1] > maior)  
        return v[n-1];  
    return maior;  
}
```

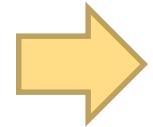
Pilha de execução – Fatorial (recursiva)

```
#include <stdio.h>

long fat (unsigned int x)
{
    if (x <= 1)
        return 1;
    return x * fat (x-1);
}
```



int main (void)
{
 int n = 5;
 long r;
 r = fat(n);
 printf("fat (%d) = %d\n", n, r);
 return 0;
}

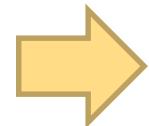
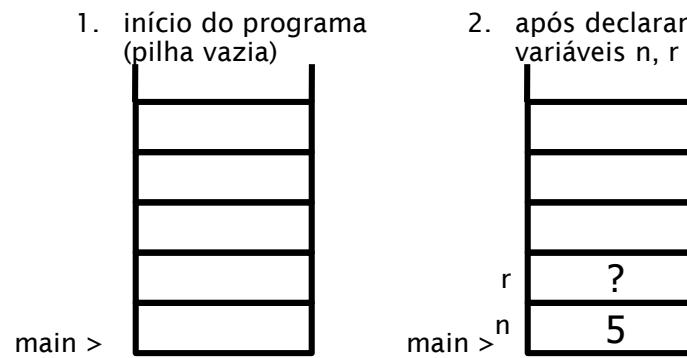


Pilha de execução – Fatorial (recursiva)

```
#include <stdio.h>

long fat (unsigned int x)
{
    if (x <= 1)
        return 1;
    return x * fat (x-1);
}
```

```
int main (void)
{
    int n = 5;
    long r;
    r = fat(n);
    printf("fat (%d) = %d\n", n, r);
    return 0;
}
```



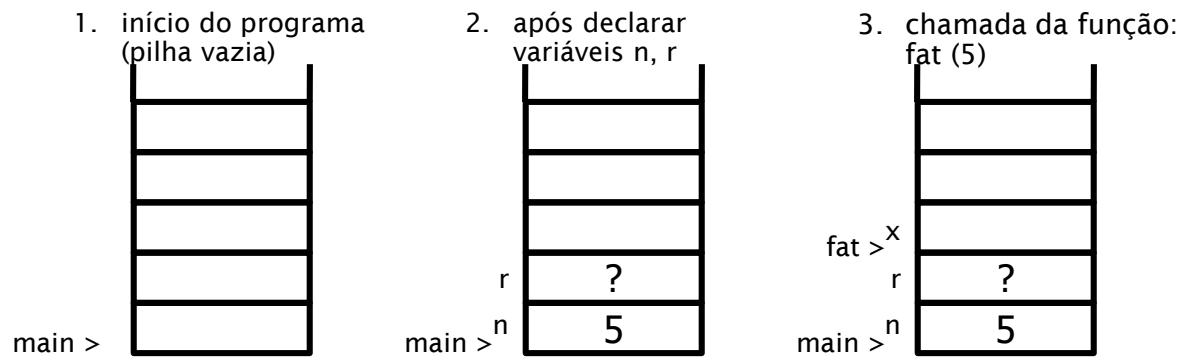
Pilha de execução – Fatorial (recursiva)

```
#include <stdio.h>

long fat (unsigned int x)
{
    if (x <= 1)
        return 1;
    return x * fat (x-1);
}
```

```
int main (void)
{
    int n = 5;
    long r;
    r = fat(n);
    printf("fat (%d) = %d\n", n, r);
    return 0;
}
```

fat(5)

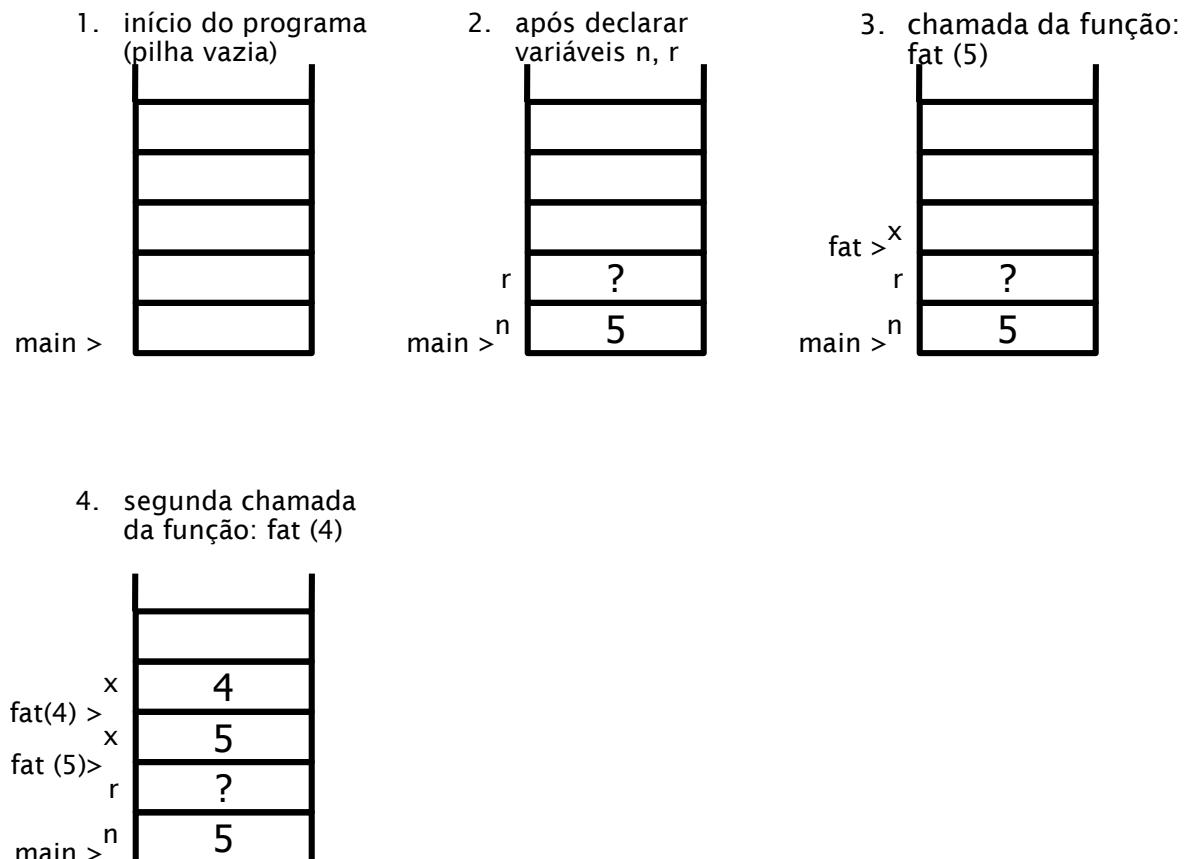


Pilha de execução – Fatorial (recursiva)

```
#include <stdio.h>

long fat (unsigned int x)
{
    if (x <= 1)
        return 1;
    return x * fat (x-1);
}
```

```
int main (void)
{
    int n = 5;
    long r;
    r = fat(n);
    printf("fat (%d) = %d\n", n, r);
    return 0;
}
```

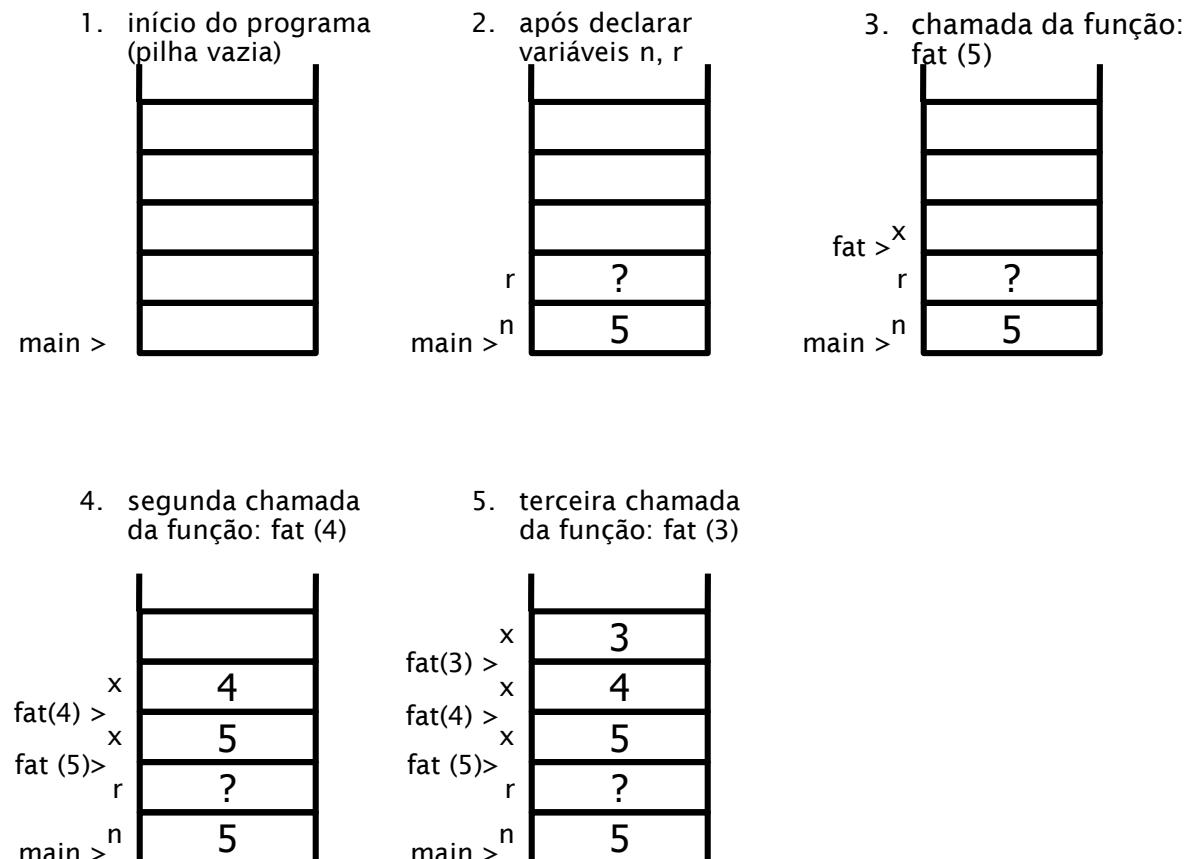


Pilha de execução – Fatorial (recursiva)

```
#include <stdio.h>

long fat (unsigned int x)
{
    if (x <= 1)
        return 1;
    return x * fat (x-1);
}
```

```
int main (void)
{
    int n = 5;
    long r;
    r = fat(n);
    printf("fat (%d) = %d\n", n, r);
    return 0;
}
```



Pilha de execução – Fatorial (recursiva)

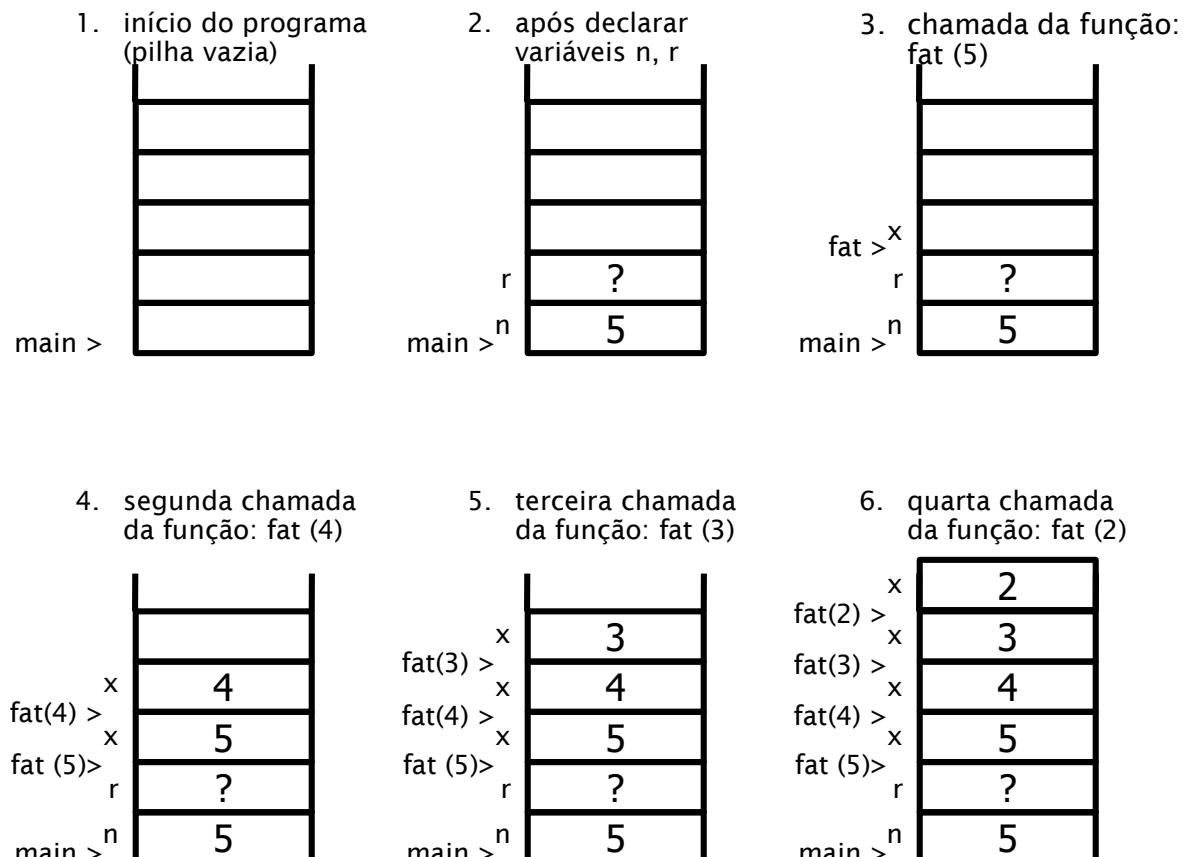
```
#include <stdio.h>

long fat (unsigned int x)

    if (x <= 1)
        return 1;
    return x * fat (x-1);

}

int main (void)
{
    int n = 5;
    long r;
    r = fat(n);
    printf("fat (%d) = %d\n", n, r);
    return 0;
}
```



Pilha de execução – Fatorial (recursiva)

```
#include <stdio.h>

long fat (unsigned int x)

    if (x <= 1)
        return 1;
    return x * fat (x-1);
}
```

```
int main (void)
{
    int n = 5;
    long r;
    r = fat(n);
    printf("fat (%d) = %d\n", n, r);
    return 0;
}
```

fat(1)

7. quinta chamada
da função: fat (1)

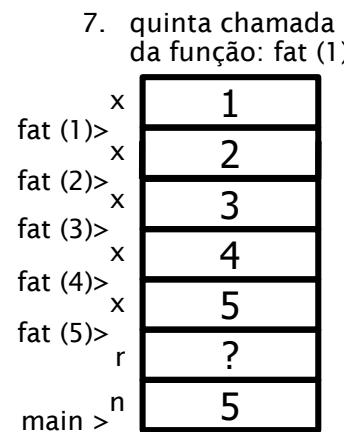
x	1
fat (1)>	2
fat (2)>	3
fat (3)>	4
fat (4)>	5
fat (5)>	?
main >	5

Pilha de execução – Fatorial (recursiva)

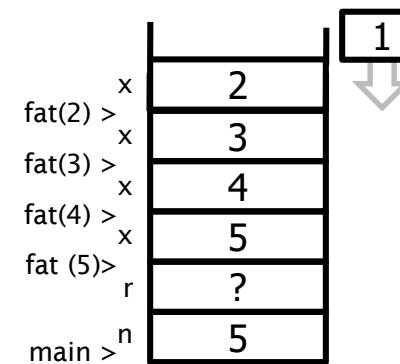
```
#include <stdio.h>

long fat (unsigned int x)
{
    if (x <= 1)
        return 1;
    return x * fat (x-1);
}

int main (void)
{
    int n = 5;
    long r;
    r = fat(n);
    printf("fat (%d) = %d\n", n, r);
    return 0;
}
```



8. fat(1) retorna 1

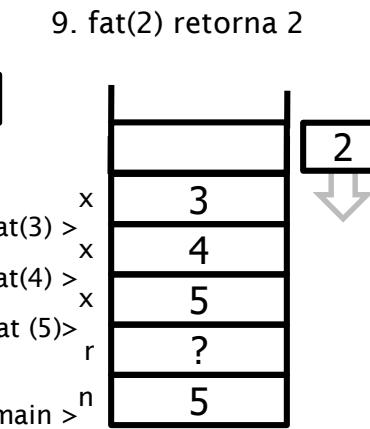
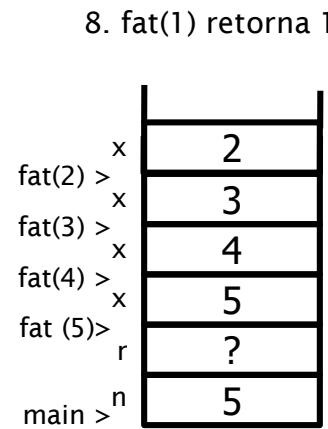
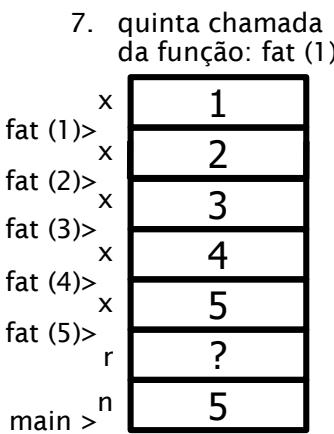


Pilha de execução – Fatorial (recursiva)

```
#include <stdio.h>

long fat (unsigned int x)
{
    if (x <= 1)
        return 1;
    return x * fat (x-1);
}

int main (void)
{
    int n = 5;
    long r;
    r = fat(n);
    printf("fat (%d) = %d\n", n, r);
    return 0;
}
```



Pilha de execução – Fatorial (recursiva)

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

```
long fat (unsigned int x)
{
    if (x <= 1)
        return 1;
    return x * fat (x-1);
}
```

fat(3)

```
int main (void)
{
    int n = 5;
    long r;
    r = fat(n);
    printf("fat (%d) = %d\n", n, r);
    return 0;
}
```

7. quinta chamada
da função: fat (1)

x	1
fat (1)>	2
fat (2)>	3
fat (3)>	4
fat (4)>	5
fat (5)>	?
main >	n 5

8. fat(1) retorna 1

x	2
fat(2) >	3
fat(3) >	4
fat(4) >	5
fat(5) >	?
main >	n 5

9. fat(2) retorna 2

x	3
fat(3) >	4
fat(4) >	5
fat(5) >	?
main >	n 5

10. fat(3) retorna 6

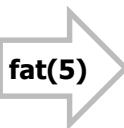
x	4
fat (4)>	5
fat (5)>	?
main >	n 5

Pilha de execução – Fatorial (recursiva)

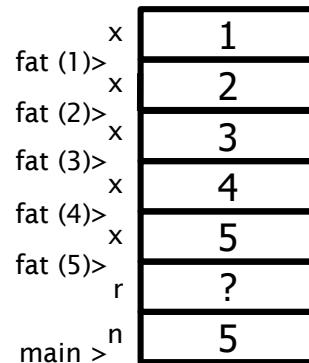
```
#include <stdio.h>
```

```
long fat (unsigned int x)
{
    if (x <= 1)
        return 1;
    return x * fat (x-1);
}
```

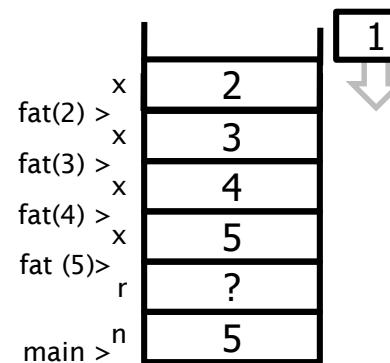
```
int main (void)
{
    int n = 5;
    long r;
    r = fat(n);
    printf("fat (%d) = %d\n", n, r);
    return 0;
}
```



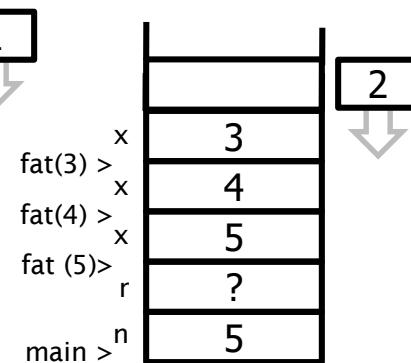
7. quinta chamada
da função: fat (1)



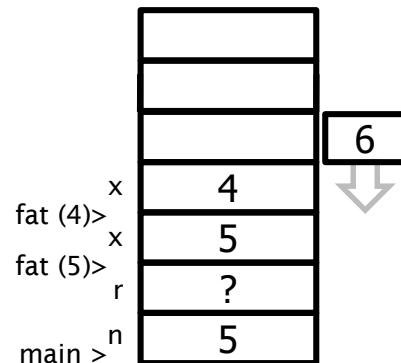
8. fat(1) retorna 1



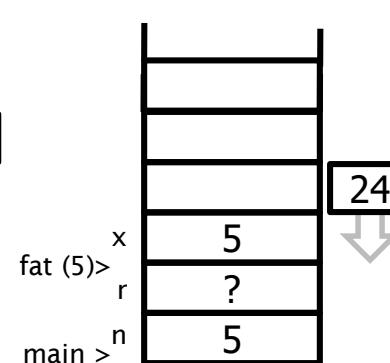
9. fat(2) retorna 2



10. fat(3) retorna 6



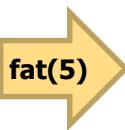
11. fat(4) retorna 24



Pilha de execução – Fatorial (recursiva)

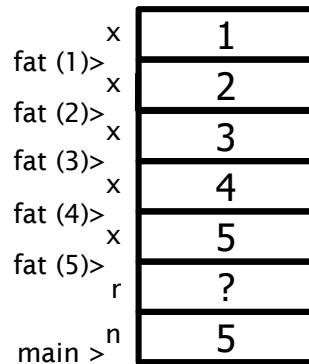
```
#include <stdio.h>
```

```
long fat (unsigned int x)
{
    if (x <= 1)
        return 1;
    return x * fat (x-1);
}
```

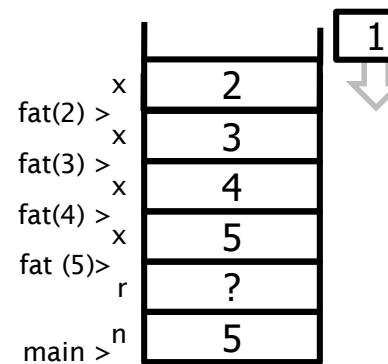


```
int main (void)
{
    int n = 5;
    long r;
    r = fat(n);
    printf("fat (%d) = %d\n", n, r);
    return 0;
}
```

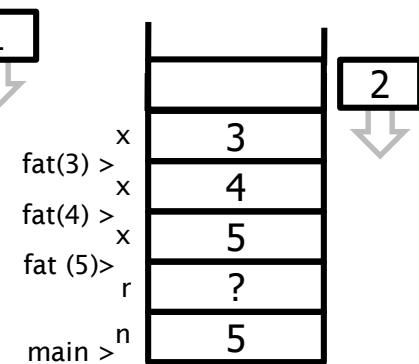
7. quinta chamada
da função: fat (1)



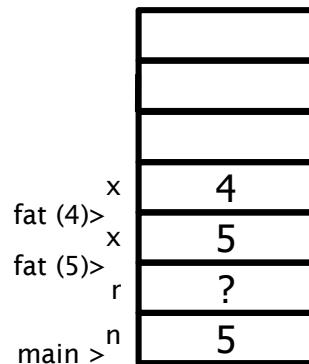
8. fat(1) retorna 1



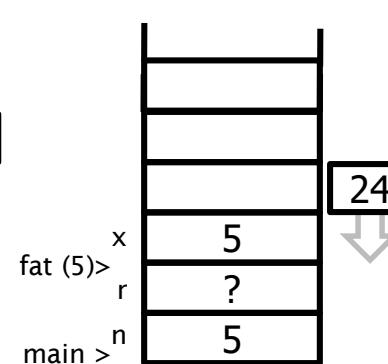
9. fat(2) retorna 2



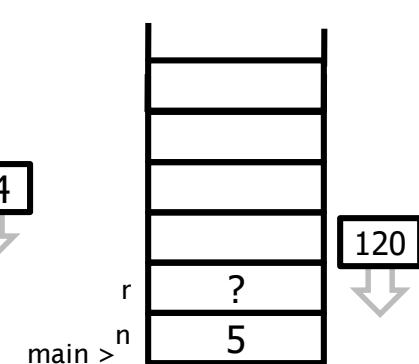
10. fat(3) retorna 6



11. fat(4) retorna 24



12. fat(5) retorna 120

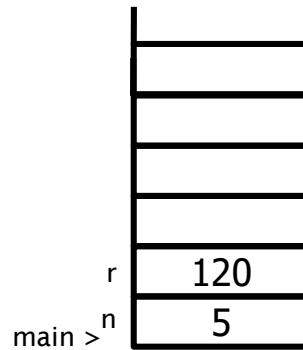


Pilha de execução – Fatorial (recursiva)

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

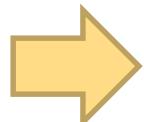
```
long fat (unsigned int x)
{
    if (x <= 1)
        return 1;
    return x * fat (x-1);
}
```

13. exibe resultado



fat(5) = 120

```
int main (void)
{
    int n = 5;
    long r;
    r = fat(n);
    printf("fat (%d) = %d\n", n, r);
    return 0;
}
```

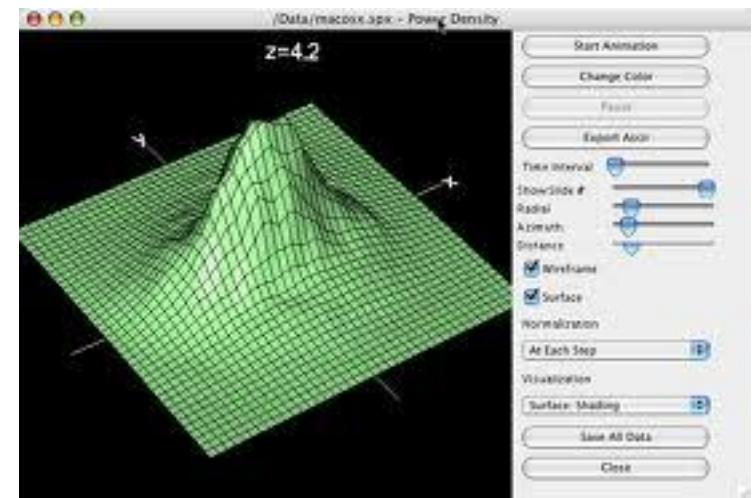


Entrada e Saída de Dados

Comandos de Entrada e Saída

Os recursos de entrada e saída de dados pelos computadores evoluiu muito. Originalmente era necessários se entrar com os dados em forma de bits manipulando chavinhos.

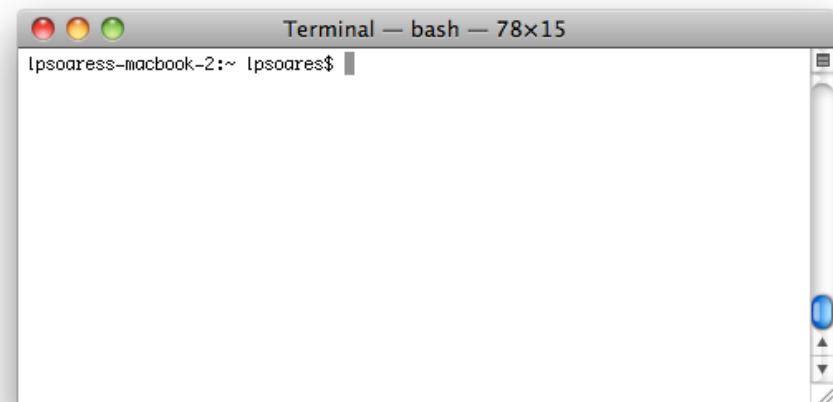
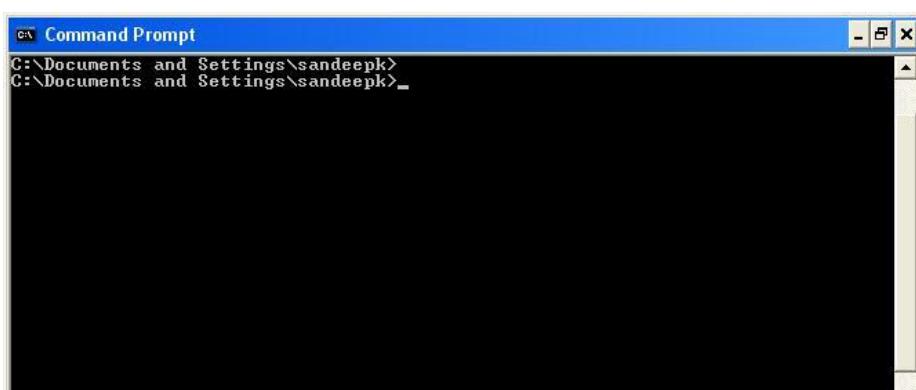
Atualmente a forma mais comum de entrada e saída de dado é por janelas, onde os programas apresentam informações bem diagramadas e campos específicos para a entrada de dados



Entrada e Saída pelo Console

O console é ainda um recurso muito usado para o desenvolvimento.

Entrada e saída de dados de forma simples, permite depurar aplicações.



linguagem C: printf e scanf

O printf e o scanf são as formas mais simples de saída e entrada de dados. Ambos precisam da biblioteca <stdio.h>

sintaxe:

```
printf ("formato", valor1, valor2, valor3,...);
```

```
scanf("formato", &variável1, &variável2,...);
```

para entrada de dados é necessário passar o endereço de memória da variável, se a variável não for um ponteiro ou vetor, esta deve ser passada com o "&"

Alternativamente para a entrada de dados ainda existe:
getch(), getche(), gets() e fgets()

Entrada e Saída: formato do printf

Especificação de formato:

- %c *especifica um **char***
- %d *especifica um **int***
- %u *especifica um **unsigned int***
- %f *especifica um **double** (ou **float**)*
- %e *especifica um **double** (ou **float**) no formato científico*
- %g *especifica um **double** (ou **float**) no formato mais apropriado (%f ou %e)*
- %s *especifica uma cadeia de caracteres*
- %o *exibe como um octonal*
- %x *exibe como um hexadecimal*
- %p *exibe como um ponteiro*

Adicionalmente ainda se pode usar o complemento l e h para long e short respectivamente.
Exemplo %lf, %le, %lg %hf, %he, %hg

Entrada e Saída: printf

Função “printf”:

possibilita a saída de valores segundo um determinado formato

```
printf (formato, lista de constantes/variáveis/expressões...);
```

```
printf ("%d %g", 33, 5.3);
```

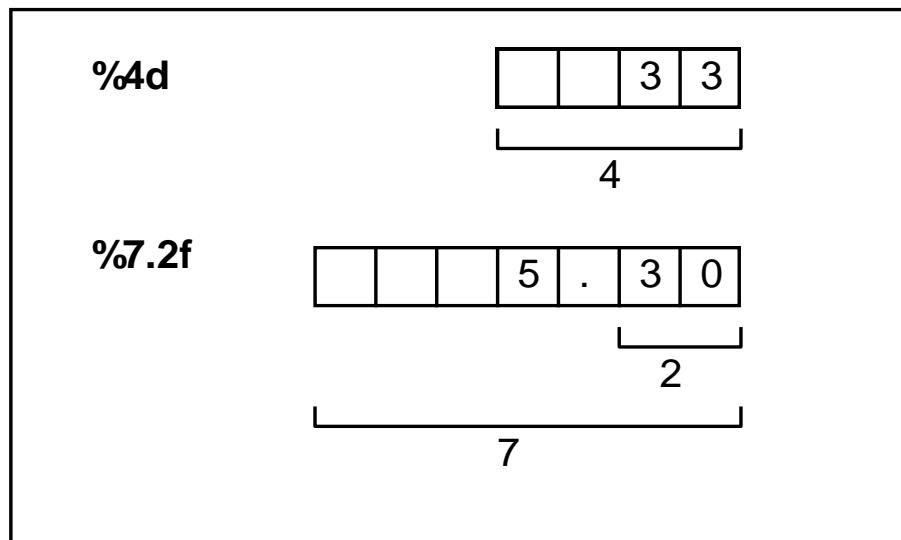
```
33 5.3
```

```
printf ("Inteiro = %d      Real = %g", 33, 5.3);
```

```
Inteiro = 33      Real = 5.3
```

Entrada e Saída: ajuste de impressão

Especificação de tamanho de campo:



Formatando dados

Além disso existem recursos para formatar melhor a entrada e saída. As instruções de controle mais usadas são:

\n mudança de linha.

\t tabulação.

\a sinal de áudio.

Entrada e Saída: exemplo

Impressão de texto:

```
printf("Curso de Estruturas de Dados\n");
```

```
Curso de Estruturas de Dados
```

Entrada e Saída: scanf

captura valores fornecidos via teclado

```
scanf (formato, lista de endereços das variáveis...);
```

```
int n;  
scanf ("%d", &n);
```

valor inteiro digitado pelo usuário é armazenado na variável n

Entrada e Saída: especificação de formato

Caracteres diferentes dos especificadores no formato servem para cercar a entrada

Espaço em branco dentro do formato faz com que sejam "pulados" eventuais brancos da entrada

`%d`, `%f`, `%e` e `%g` automaticamente pulam os brancos que precederem os valores numéricos a serem capturados

```
scanf ("%d:%d", &h, &m);
```

valores (inteiros) fornecidos devem ser separados pelo caractere dois pontos (:)

getch() e getche()

O comando getch() permite ler um caractere do console por vez, já o getche() lê o caractere e exibe ele diretamente no console (ou seja o faz um eco);

Os caracteres são lidos e convertidos para inteiros sem sinal que podem ser usados com o char.

```
int getch( void );
```

```
int getche( void );
```

gets() e fgets()

O comando gets permite ler uma string diretamente do console, porém é um comando que não verifica o tamanho máximo especificado. O fgets já possui esse controle, portanto deve ser usado preferencialmente:

```
char *gets( char *str );
```

```
char *fgets( char *str, int count, FILE *stream );
```

Arquivos

Arquivos são estruturas de dados que ficam armazenados em dispositivos secundários de memória. Em geral:

- Mais lentos;
- Maior capacidade;
- Persistentes.

Os principais dispositivos de armazenamento atualmente são:

- Discos Rígidos (HDs)
- Pen drives
- CDs e DVDs



Os arquivos armazenados nestes dispositivos possuem sempre um identificação (nome) e sua localização. Outros atributos como data e tamanho também são normalmente usados no índice.

Acessando Arquivos

Arquivos são lidos como uma sequencia de bytes

Estes bytes podem estar organizados de forma que só um programa específico entenda;

Estes bytes podem ser também somente caracteres (ASCII), assim sua leitura é trivial. Programas de edição de texto puro trabalham com estes arquivos.

Os arquivos podem ser divididos em dois sub-grupos:

- Texto;
- Binários.

Detalhes

As funcionalidades de I/O do C descendem do "portable I/O package" escrito por Mike Lesk no Bell Labs nos anos 70.

C não possui um acesso aleatório direto por suas funções padrões. Para ir para uma certa posição se deve saltar até esse ponto e começar a ler os dados dessa posição (seek)

Leitura em C

A linguagem C prove funções padronizadas para a leitura e escrita de arquivos.

Em linguagem C existe um tipo de dado especial chamado FILE que permite a manipulação de arquivos.

Este tipo é encontrado dentro de <stdio.h>

Leitura em C

Os métodos mais usados para abrir e fechar um arquivo são `fopen()` e `fclose()` respectivamente:

```
FILE *fopen( const char *fname, const char *mode );
int fclose( FILE *stream );
```

Para a manipular textos em arquivos, existem as funções `fscanf()`, `fgets()` e `fgetc()` e para a escrita `fprintf()`, `fputs()` e `fputc()`.

```
int fscanf( FILE *stream, const char *format, ... );
char *fgets( char *str, int num, FILE *stream );
int fgetc( FILE *stream );
int fprintf( FILE *stream, const char *format, ... );
int fputs( const char *str, FILE *stream );
int fputc( int ch, FILE *stream );
```

Para manipular arquivos binários existem as funções `fread()` e `fwrite()` que permitem se trabalhar com blocos de dados.

```
int fread( void *buffer, size_t size, size_t num, FILE *stream );
int fwrite( const void *buffer, size_t size, size_t count, FILE *stream );
```

Métodos de Acesso em C

As funções para operar arquivos de C precisam definir um dos seguintes modos de operação:

r - leitura

w - escrita (arquivo não precisa existir)

a - atualizar (arquivo não precisa existir)

r+ - leitura e escrita, posicionando no começo do documento

w+ - leitura e escrita sobre escrevendo arquivo

a+ - leitura e escrita para atualizar arquivo no final dele

Para arquivos binários é necessário se adicionar um “b” no modo para o sistema tratar o arquivo de forma adequada.

Exemplo de Leitura com C

Testando escrever em um arquivo de texto

```
FILE *fp;  
fp=fopen("teste.txt", "w");  
fprintf(fp, "Testando 1 2 3\n");  
fclose(fp);
```

Testando escrever em um arquivo binário

```
FILE *fp;  
fp=fopen("teste.bin", "wb");  
char x[26]="ABCDEFGHIJKLMNOPQRSTUVWXYZ";  
fwrite(x, sizeof(x[0]),  
sizeof(x)/sizeof(x[0]), fp);
```

Exemplo Mais Completo em Texto

```
#include <stdio.h>
int main() {
    FILE *arq;
    float mat[3][3];
    int c,l;
    arq = fopen("arquivo.txt", "w");
    for(l=0;l<3;l++) {
        for(c=0;c<3;c++) {
            fscanf(arq, "%f", &mat[l][c]);
            printf(" %f ",mat[l][c]);
        }
        printf("\n");
    }
    fclose(arq);
    return 0;
}
```

Navegação em arquivos binários

A linguagem C fornece algumas funções para se navegar internamente aos arquivos.

long ftell(FILE *stream) : retorna a posição atual no arquivo;

int fseek(FILE *stream, long offset, int origin) : move o indicador do arquivo para uma posição específica;

void rewind(FILE *stream) : move o indicador do arquivo o início do arquivo.

int fgetpos(FILE *stream, fpos_t *position) : recupera o indicador da posição no arquivo;

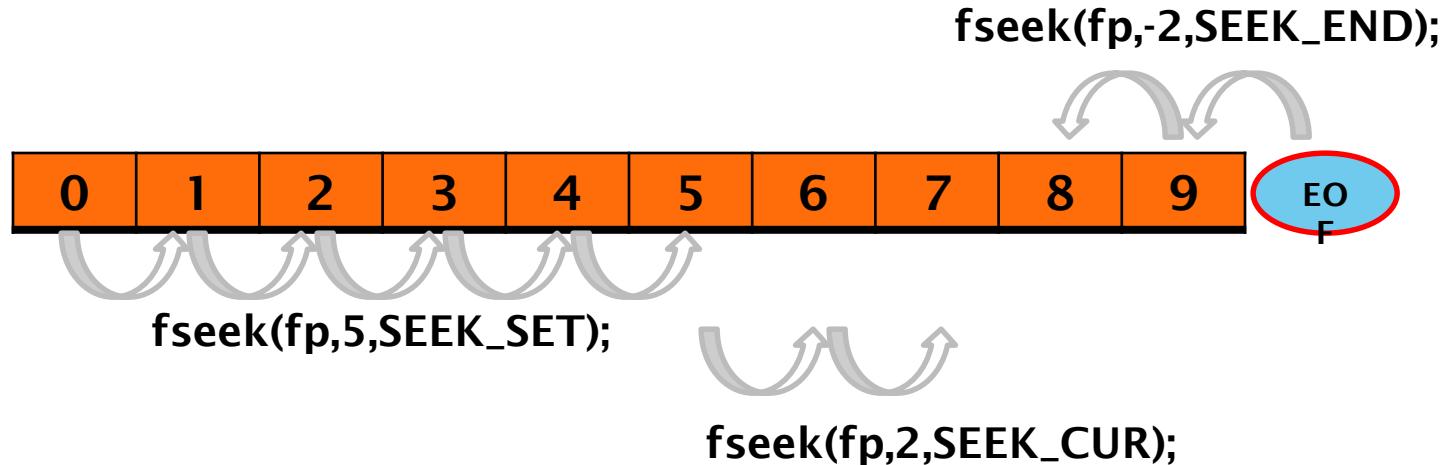
int fsetpos(FILE *stream, const fpos_t *position) : move o indicador do arquivo para uma posição específica;

Constantes

- EOF um inteiro negativo do tipo int usado para indicar o fim de um arquivo
- SEEK_CUR um número inteiro que pode ser passado para o fseek() para solicitar um posicionamento em relação à posição do arquivo corrente
- SEEK_END um número inteiro que pode ser passado para o fseek() para solicitar um posicionamento relativo ao fim do arquivo corrente
- SEEK_SET um número inteiro que pode ser passado para o fseek() para solicitar o posicionamento em relação ao início do arquivo

Exemplo de Navegação

exemplo



Exemplo Completo em Binário

```
#include <stdio.h>
#include <stdlib.h>
int main () {
    FILE * pFile;
    long lSize;
    char * buffer;
    size_t result;
    pFile = fopen( "myfile.bin" , "rb" );
    if(pFile==NULL) {fputs("File error",stderr); exit(1);}
    fseek(pFile , 0 , SEEK_END);
    lSize = ftell(pFile);
    rewind(pFile);
    buffer = (char*) malloc(sizeof(char)*lSize);
    if(buffer == NULL) {fputs("Memory error",stderr); exit(2);}
    result = fread(buffer,1,lSize,pFile);
    if(result != lSize) {fputs("Reading error",stderr); exit(3);}
    fclose(pFile);
    return 0;
}
```

Operações Sobre o Próprio Arquivo

remove o arquivo:

```
int remove( const char *fname )
```

renomeia o arquivo:

```
int rename( const char *oldname, const char *newname );
```

dúvidas?