DEPARTAMENTO DE INFORMÁTICA
PUC·RIO

PONTIFÍCIA UNIVERSIDADE CATÓLICA
DO RIO DE JANEIRO

INF 1010 – Estrutura de Dados Avançadas

Aula 01 – Revisão da Linguagem C

2020.1

Prof. Augusto Baffa
<abaffa@inf.puc-rio.br>

LES ECOA PUCRIO
GAMES visionlab visualization laboratory

Estrutura de um Programa C

Inclusão de bibliotecas auxiliares: **#include <nome>**

Definição de constantes: **#define nome valor**

Funções auxiliares

Função Principal (início da execução de um programa): **int main(void)**

Geração de Executavel com GCC

code/intro/hello.c

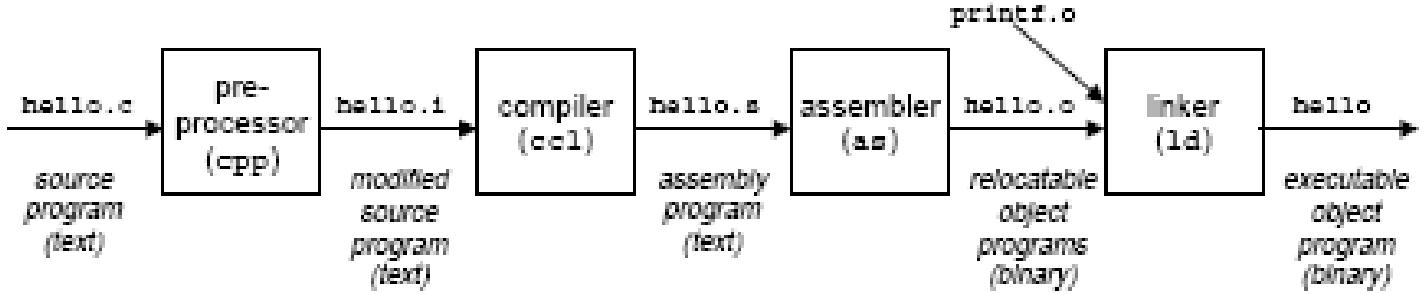
```
1 #include <stdio.h>
2
3 int main()
4 {
5         printf("Hello World\n");
6 }
```

code/intro/hello.c

- O programa **fonte** (arquivo texto) deve ser "traduzido“ para uma sequência de instruções de linguagem de máquina, que é armazenada em um arquivo binário (**executável**)
  - essa tradução é realizada em 4 passos

Passos para a geração do executável

gcc -o hello hello.c



- Passo 1: pré-processamento
- Passo 2: compilação
- Passo 3: montagem
- Passo 4: linkedição (amarração, ligação)

Revisão Linguagem C

PONTEIROS

Ponteiros

- **Ponteiro é tipo especiais de dado que armazenam um endereços de memória.**

- Uma variável do tipo ponteiro deve ser declarada da seguinte forma:

```
tipo *nome_variavel;
```

- A variável ponteiro armazena um endereço de memória de uma outra variável do tipo especificado.

```
int *ponteiro_a; float *ponteiro_b;
```

Operadores de Ponteiros

- Existem dois **operadores** relacionados a ponteiros:
  - Operador unário **&** (“endereço de”) retorna o endereço de memória de uma variável:

```
int *p;
int a = 40;
p = &a;
```

  - Operador unário ***** (“conteúdo de”) retorna o conteúdo do endereço indicado pelo ponteiro:

```
printf("%d", *p);
```

Operadores de Ponteiros

```
#include <stdio.h>

int main(void)
{
   int a;
   int *p;                    /* Declaração */
   p = &a;                    /* Inicialização */
   a = 0;
   *p = 2;

   printf("%d", a);
   return 0;
}
```

Saída em tela:

```
2
```

Operadores de Ponteiros

```
#include <stdio.h>

int main(void)
{
   int a;
   int *p = &a;      /* Declaração e Inicialização */

   *p = 10;

   printf("%d", a);
   return 0;
}
```

Saída em tela:

```
10
```

Operadores de Ponteiros

```
#include <stdio.h>

int main(void)
{
   int num, q = 1;
   int *p;
   num = 100;
   p = &num;
   q = *p;

   printf("%d", q);
   return 0;
}
```

Saída em tela:

```
100
```

Ponteiros: cuidados

```
int main ( void )
{
    int a, b, *p;
    a = 2;
    *p = 3;
    b = a + (*p);
    printf(" %d ", b);
    return 0;
}
```

- erro na atribuição *p = 3
  - utiliza a memória apontada por p para armazenar o valor 3, sem que p tivesse sido inicializada, logo
  - armazena 3 num espaço de memória desconhecido

Ponteiros: cuidados

- Como o operador * de ponteiros é igual ao operador * utilizado na multiplicação, deve-se ter cuidado no uso desses operadores:

```
#include <stdio.h>

int main()
{
   int b, a;
   int *c;
   b = 10;
   c = &a;
   *c = 11;
   a = b * c;
   printf("%d", a);
   return 0;
}
```

Ocorre um erro de compilação, pois o * é interpretado como operador de ponteiro sobre c

Ponteiros: cuidados

- Para corrigir é necessário isolar o operador de ponteiro na expressão:

```
#include <stdio.h>

int main()
{
   int b, a;
   int *c;
   b = 10;
   c = &a;
   *c = 11;
   a = b * (*c);
   printf("%d", a);
   return 0;
}
```

Passagem de Parâmetros

- Os parâmetros de uma função são o mecanismo utilizado para passar a informação de um trecho de código para o interior da função.

- **Há dois tipos de passagem de parâmetros:**

  - Passagem por valor.

  - Passagem por referência.
    - **Mais eficiência:** as funções recebem os endereços para as variáveis já inicializadas e o tamanho do endereço é sempre o mesmo, assim não há problema envolvendo cópias e inicialização.

    - **Mais liberdade:** possibilita a criação de funções que podem retornar mais do que um valor.

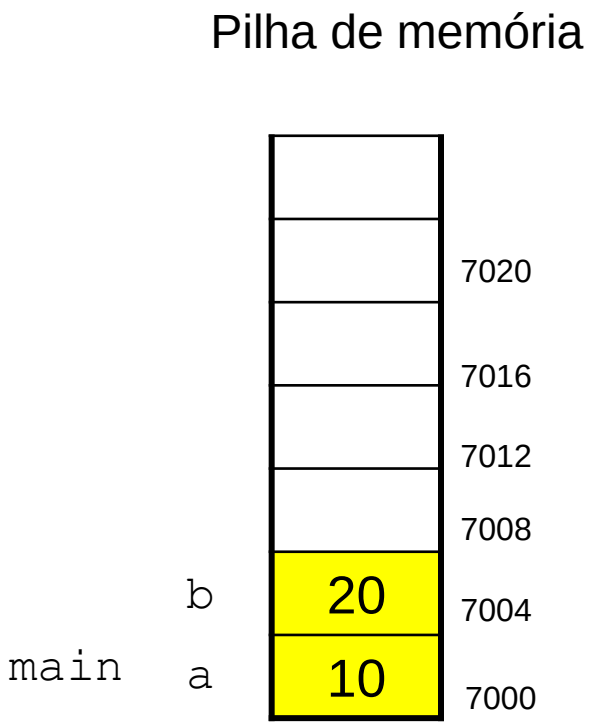
Passagem de Parâmetros por Valor

```
#include <stdio.h>

void troca(int a, int b);

int main (void)
{
       int a=10, b=20;
       troca(a,b);
       printf(" A=%d  B=%d\n",a,b);
}

void troca(int a, int b) {
       int tmp=b;
       b=a;
       a=tmp;
}
```

```
 A=10  B=20

Press any key to continue
```

Passagem de Parâmetros por Valor

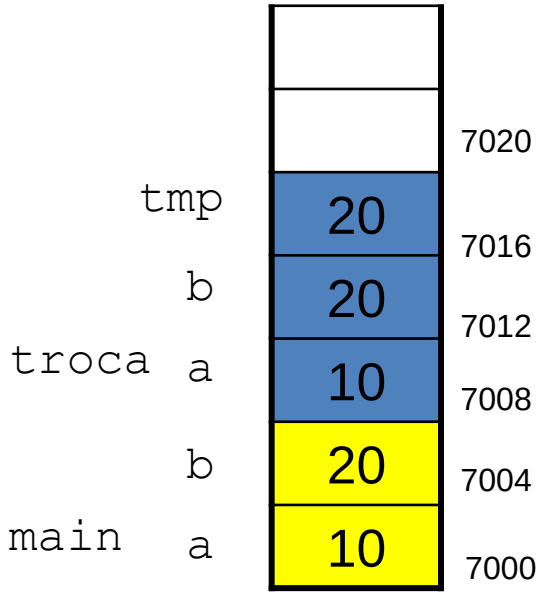
```
#include <stdio.h>

void troca(int a, int b);

int main (void)
{
        int a=10, b=20;
        troca(a,b);
        printf(" a=%d  b=%d\n",a,b);
}

void troca(int a, int b) {
        int tmp=b;
        b=a;
        a=tmp;
}
```



Passagem de Parâmetros por Valor

```
#include <stdio.h>

void troca(int a, int b);

int main (void)
{
        int a=10, b=20;
        troca(a,b);
        printf(" a=%d  b=%d\n",a,b);
}

void troca(int a, int b) {
        int tmp=b;
    ⇨   b=a;
        a=tmp;
}
```

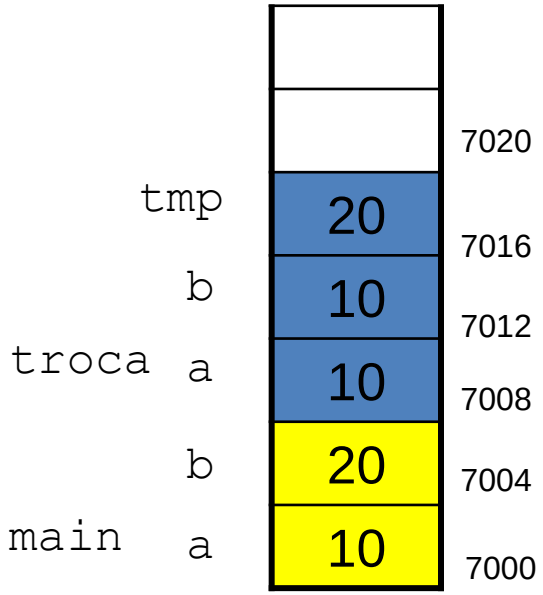
Pilha de memória



Passagem de Parâmetros por Valor

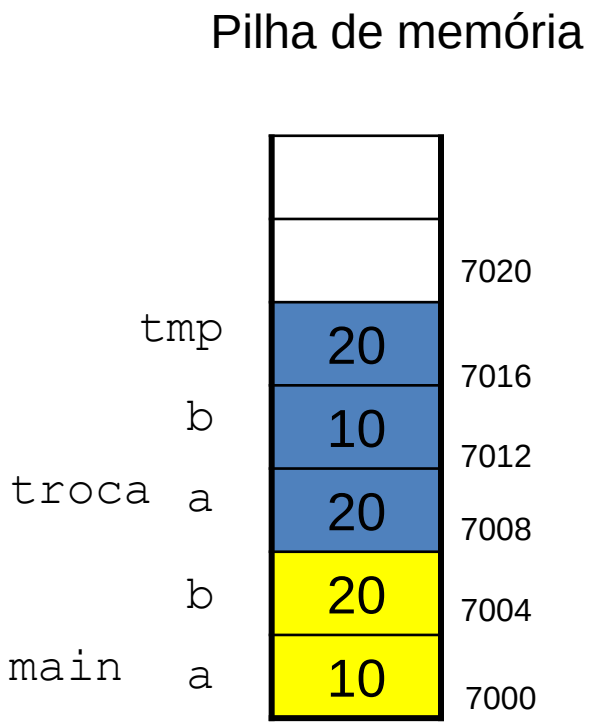
```
#include <stdio.h>

void troca(int a, int b);

int main (void)
{
        int a=10, b=20;
        troca(a,b);
        printf(" a=%d  b=%d\n",a,b);
}

void troca(int a, int b) {
        int tmp=b;
        b=a;
        a=tmp;
}
```

Pilha de memória



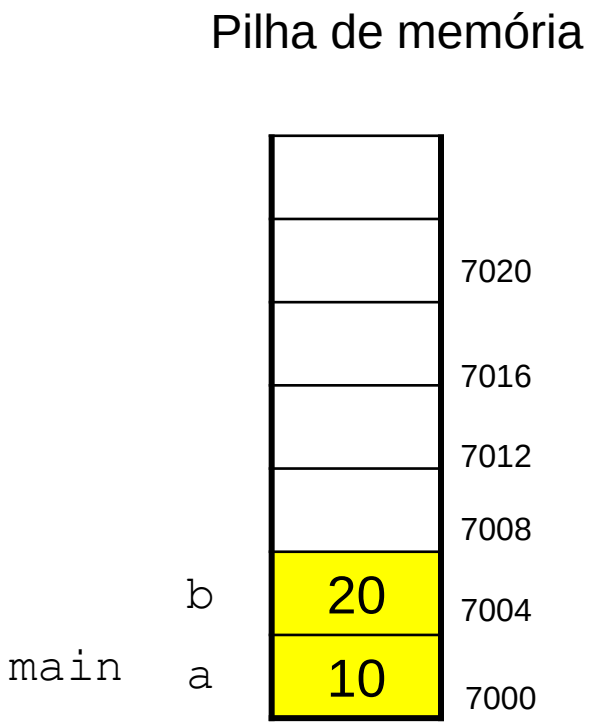
Passagem de Parâmetros por Valor

```
#include <stdio.h>

void troca(int a, int b);

int main (void)
{
        int a=10, b=20;
        troca(a,b);
        printf(" a=%d  b=%d\n",a,b);
}

void troca(int a, int b) {
        int tmp=b;
        b=a;
        a=tmp;
}
```



Passagem de Parâmetros por Valor

```
#include <stdio.h>

void troca(int a, int b);

int main (void)
{
        int a=10, b=20;
        troca(a,b);
   ⇨    printf(" a=%d  b=%d\n",a,b);
}

void troca(int a, int b) {
        int tmp=b;
        b=a;
        a=tmp;
}
```



Passagem de Parâmetros por Referência

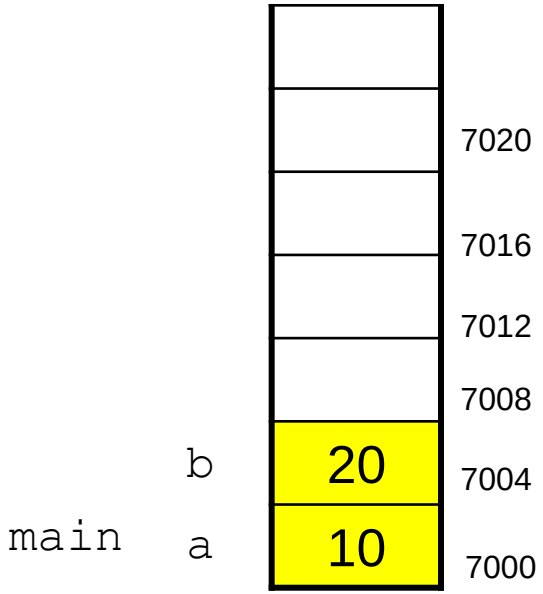
```
#include <stdio.h>

void troca(int *pa, int *pb);

int main (void)
{
       int a=10, b=20;
       troca(&a,&b);
       printf(" a=%d  b=%d\n",a,b);
}

void troca(int *pa, int *pb) {
       int tmp=*pb;
       *pb=*pa;
       *pa=tmp;
}
```

```
 a=20  b=10

Press any key to continue
```

Passagem de Parâmetros por Referência

```
#include <stdio.h>

void troca(int *pa, int *pb);

int main (void)
{
        int a=10, b=20;
        troca(&a,&b);
        printf(" a=%d  b=%d\n",a,b);
}

void troca(int *pa, int *pb) {
        int tmp=*pb;
        *pb=*pa;
        *pa=tmp;
}
```

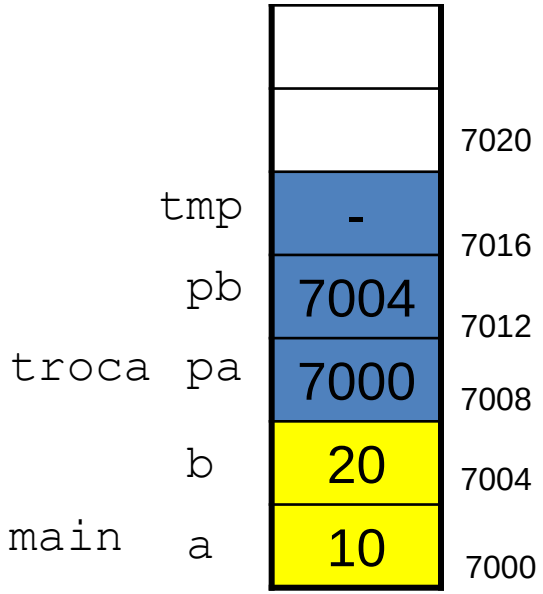
Pilha de memória



Passagem de Parâmetros por Referência

```
#include <stdio.h>

void troca(int *pa, int *pb);

int main (void)
{
        int a=10, b=20;
        troca(&a,&b);
        printf(" a=%d  b=%d\n",a,b);
}

void troca(int *pa, int *pb) {
        int tmp=*pb;
        *pb=*pa;
        *pa=tmp;
}
```

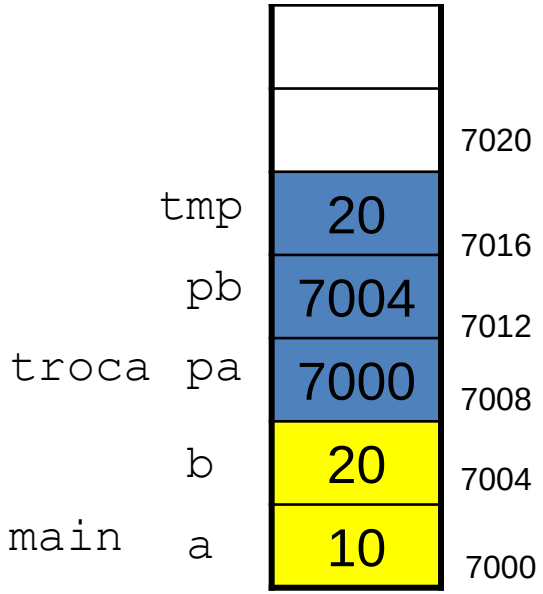
Pilha de memória



Passagem de Parâmetros por Referência

```
#include <stdio.h>

void troca(int *pa, int *pb);

int main (void)
{
        int a=10, b=20;
        troca(&a,&b);
        printf(" a=%d  b=%d\n",a,b);
}

void troca(int *pa, int *pb) {
        int tmp=*pb;
        *pb=*pa;
        *pa=tmp;
}
```

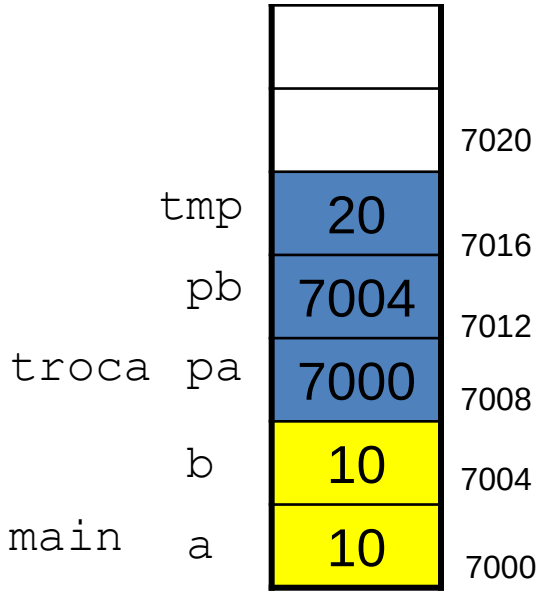
Pilha de memória



Passagem de Parâmetros por Referência

```
#include <stdio.h>

void troca(int *pa, int *pb);

int main (void)
{
        int a=10, b=20;
        troca(&a,&b);
        printf(" a=%d  b=%d\n",a,b);
}

void troca(int *pa, int *pb) {
        int tmp=*pb;
        *pb=*pa;
        *pa=tmp;
}
```

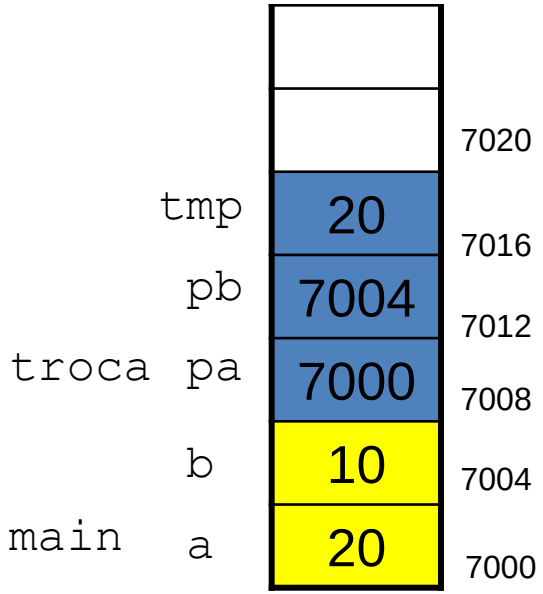
Pilha de memória



Passagem de Parâmetros por Referência

```
#include <stdio.h>

void troca(int *pa, int *pb);

int main (void)
{
        int a=10, b=20;
        troca(&a,&b);
        printf(" a=%d  b=%d\n",a,b);
}

void troca(int *pa, int *pb) {
        int tmp=*pb;
        *pb=*pa;
        *pa=tmp;
}
```

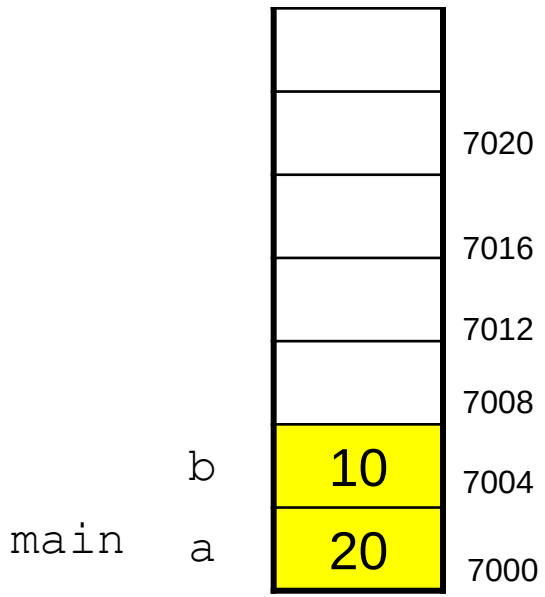
Pilha de memória



Passagem de Parâmetros por Referência

```
#include <stdio.h>

void troca(int *pa, int *pb);

int main (void)
{
        int a=10, b=20;
        troca(&a,&b);
    ⇨   printf(" a=%d  b=%d\n",a,b);
}

void troca(int *pa, int *pb) {
        int tmp=*pb;
        *pb=*pa;
        *pa=tmp;
}
```

Pilha de memória




```
 a=20  b=10

Press any key to continue
```

Revisão Linguagem C

ALOCAÇÃO DINÂMICA

Vetores - Declaração e Inicialização

- **Declaração de um vetor:**

```
int meu_vetor[10];
```

  – Reserva um espaço de memória para armazenar 10 valores inteiros no vetor chamado meu_vetor.

- **Inicialização de algumas posições do vetor meu_vetor:**

```
meu_vetor[0] = 5;
meu_vetor[1] = 11;
meu_vetor[4] = 0;
meu_vetor[9] = 3;
```

5	11	?	?	0	?	?	?	?	3
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*

Vetores - Declaração e Inicialização

- **Exemplos de Declaração:**

```
int a, b[20];

float c[10];

double d[30], e, f[5];
```

- **Declaração e Inicialização:**

```
int teste[5] = {12, 5, 34, 32, 9};

float vetor1[3] = {2.5, 5.8, 10.1};
```

Uso da Memória

- **Uso por variáveis globais (e estáticas):**
  - espaço reservado para uma variável global fica disponível enquanto o programa estiver sendo executado;
- **Uso por variáveis locais:**
  - espaço disponível apenas enquanto a função que declarou a variável está sendo executada;
  - liberado para outros usos quando a execução da função termina;
- **Variáveis globais ou locais podem ser simples ou vetores:**
  - para vetor, é necessário informar o número máximo de elementos pois o compilador precisa calcular o espaço a ser reservado;

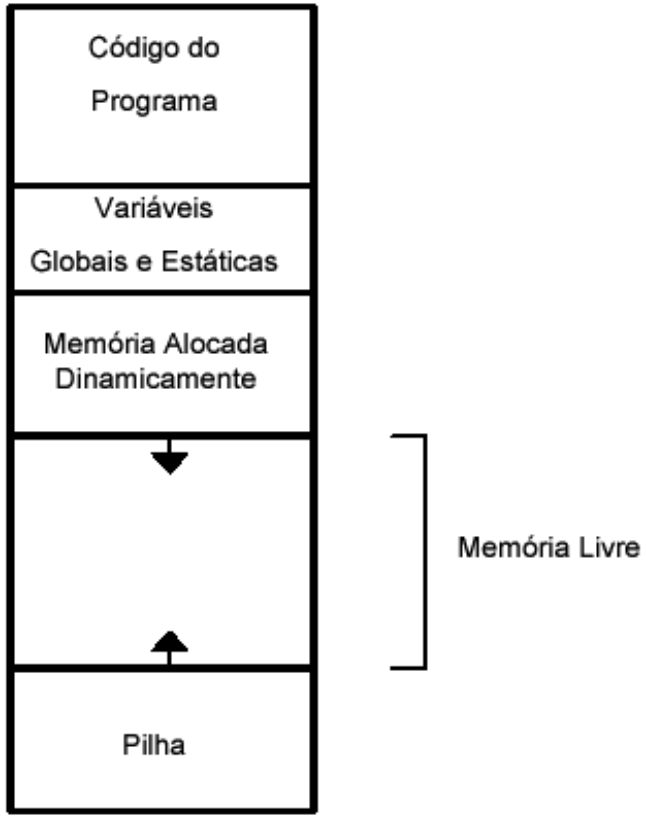
Alocação Dinâmica

- A biblioteca padrão <`stdlib.h`> possui um conjunto de funções para tratar a alocação dinâmica de memória:

```
void * malloc(int num_bytes);
```

```
int sizeof(type t);
```

```
void free(void * p);
```



Alocação Dinâmica

- Função "**malloc**":

```
void * malloc(int num_bytes);
```

  - Recebe como parâmetro o número de bytes que se deseja alocar;
  - Retorna um ponteiro genérico para o endereço inicial da área de memória alocada, se houver espaço livre:
    - ponteiro genérico é representado por `void*`
    - ponteiro é convertido automaticamente para o tipo apropriado
    - ponteiro pode ser convertido explicitamente
  - Retorna um endereço nulo (`NULL`), se não houver espaço livre.

Alocação Dinâmica

- Função "**sizeof**":

```
int sizeof(type t);
```

  - Retorna o número de bytes necessários para representar valores de um determinado tipo.
  - Exemplos:
    - char (1 byte)
    - int (4 bytes)
    - double (8 bytes)

- Função "**free**":

```
void free(void * p);
```

  - Recebe como parâmetro o ponteiro da memória a ser liberada
    - A função free deve receber um endereço de memória que tenha sido alocado dinamicamente

Alocação Dinâmica - Exemplo

- Alocação dinâmica de um vetor de inteiros com 10 elementos

```
int *v;
v = (int *) malloc(10*sizeof(int));
```

  - O malloc retorna o endereço da área alocada para armazenar valores inteiros;
  - O ponteiro é convertido para um ponteiro de inteiro;
  - O ponteiro de inteiro *v recebe endereço inicial do espaço alocado;

Alocação Dinâmica - Exemplo

- Alocação dinâmica de um vetor de inteiros com 10 elementos

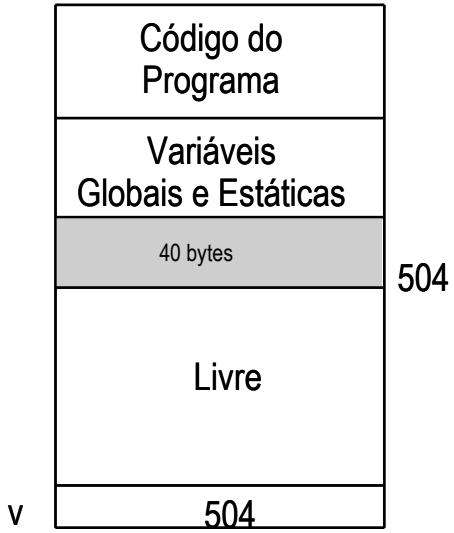
```
int *v;
v = (int *) malloc(10*sizeof(int));
```

1 - Declaração: int *v

Abre-se espaço na pilha para o ponteiro (variável local)

2 - Comando: v = (int *) malloc (10*sizeof(int))

Reserva espaço de memória da área livre e atribui endereço à variável



Alocação Dinâmica - Exemplo

```
int *v;
v = (int *) malloc(10*sizeof(int));
```

- `v` armazena endereço inicial de uma área contínua de memória suficiente para armazenar 10 valores inteiros;

- `v` pode ser tratado como um vetor declarado estaticamente;
  - `v` aponta para o inicio da área alocada
  - `v[0]` acessa o espaço para o primeiro elemento
  - `v[1]` acessa o segundo
  - .... até `v[9]`

Revisão Linguagem C

STRINGS

Caracteres

- A linguagem C fornece um tipo de dado especial para armazenar caracteres: **`char`**
  - Tamanho de char = 1 byte = 8 bits = 256 valores distintos;
  - Exemplo de declaração:

```
char letra1;
```

- Caracteres são representados através de **códigos numéricos**.
  - Exemplo: Tabela **ASCII**

Códigos ASCII de alguns caracteres

(sp representa espaço)

	0	1	2	3	4	5	6	7	8	9
30			sp	!	"	#	$	%	&	'
40	(	)	*	+	,	-	.	/	0	1
50	2	3	4	5	6	7	8	9	:	;
60	<	=	>	?	@	A	B	C	D	E
70	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y
90	Z	[	\	]	^	_	`	a	b	c
100	d	e	f	g	h	i	j	k	l	m
110	n	o	p	q	r	S	t	u	v	w
120	x	y	z	{	|	}	~			

Exemplo:

82	105	110	32	100	101	32	74	97	110	101	105	114	111
R	i	o		d	e		J	a	n	e	i	r	o

Leitura de caracteres

- É possível utilizar o **scanf** para ler caracteres e cadeias de caracteres.
  - O formato **%c** permite a leitura de um único caractere:

```
char a;
...
scanf("%c", &a);
...
```

Cadeias de caracteres

- Uma cadeia de caracteres, mais conhecida como **string**, é uma sequência de letras e símbolos.

- A linguagem C não possui um tipo de dado para armazenar strings, mas ela permite a criação de vetores de caracteres.

  - Vetor do tipo **char**, terminado pelo caractere **nulo** ('\0')
    - é necessário reservar uma posição adicional no vetor para o caractere de fim da cadeia

  - As funções que manipulam cadeias de caracteres:
    - recebem como parâmetro um vetor de char
    - processam caractere por caractere até encontrar o caractere nulo, sinalizando o final da cadeia

Cadeias de caracteres

- A **inicialização de uma cadeia de caracteres** pode ser feita de forma semelhante a inicialização de um vetor (colocando cada caracteres entre aspas duplas)

- •A linguagem C também fornece uma forma de inicialização **mais simples** utilizando aspas duplas (o caractere nulo é representado implicitamente)

```
int main ( void )
{
   char cidade[ ] = "Rio";
   printf("%s \n", cidade);
   return 0;
}
```

≡

```
int main ( void )
{
   char cidade[ ]={'R', 'i', 'o', '\0'};
   printf("%s \n", cidade);
   return 0;
}
```

```
Rio

Press any key to continue
```

Leitura de Cadeias de caracteres

- É possível utilizar o **scanf** para ler caracteres e cadeias de caracteres.
  - O formato **%s** permite a leitura de uma cadeia de caracteres não brancos:

```
char cidade[81];
...
scanf("%s", cidade);
...
```

  - Não é necessário usar &cidade pois cidade é um vetor (ponteiro)
  - Somente lê palavras simples, se o usuário digitar “Rio de Janeiro”, somente “Rio” será capturado, pois %s lê somente uma sequência de caracteres **não brancos**.

Exemplo

```
#include <stdio.h>

int comprimento (char* s)
{
   int i;
   int n = 0;  /* contador */
   for (i=0; s[i] != '\0'; i++)
      n++;
   return n;
}

int main (void)
{
   int tam;
   char cidade[] = "Rio de Janeiro";
   tam = comprimento(cidade);
   printf("A string \"%s\" tem %d caracteres\n", cidade, tam);
   return 0;
}
```

Manipulação de Cadeias de caracteres

- Biblioteca de cadeias de caracteres **string.h**:
  - **strlen**: determina o comprimento de uma cadeia;
  - **strcpy**: copia uma cadeia origem para outra destino;
  - **strcat**: concatena duas cadeias;
  - **strcmp**: compara duas cadeias;

Constante de Cadeia de Caracteres

- Representada por sequência de caracteres delimitada por aspas duplas;

- Comporta-se como uma expressão constante, cuja avaliação resulta no ponteiro para onde a cadeia de caracteres está armazenada;

```
#include <string.h>

int main ( void )
{
   char cidade[4];
   strcpy (cidade, "Rio");
   printf ( "%s \n", cidade );
   return 0;
}
```

```
#include <string.h>

int main (void)
{
   char *cidade;
   cidade = "Rio";
   printf ( "%s \n", cidade );
   return 0;
}
```

```
cidade = (char*)malloc((4+1)*sizeof(char));
```

Constante de Cadeias de caracteres

- Exemplos:

```
char s1[] = "Rio de Janeiro";
```

  - s1 é um vetor de char, inicializado com a cadeia **`Rio de Janeiro`**, seguida do caractere nulo
  - s1 ocupa 15 bytes de memória
  - é válido escrever s1[0]='X‘, alterando o conteúdo da cadeia para **`Xio de Janeiro`**, pois s1 é um vetor, permitindo alterar o valor de seus elementos

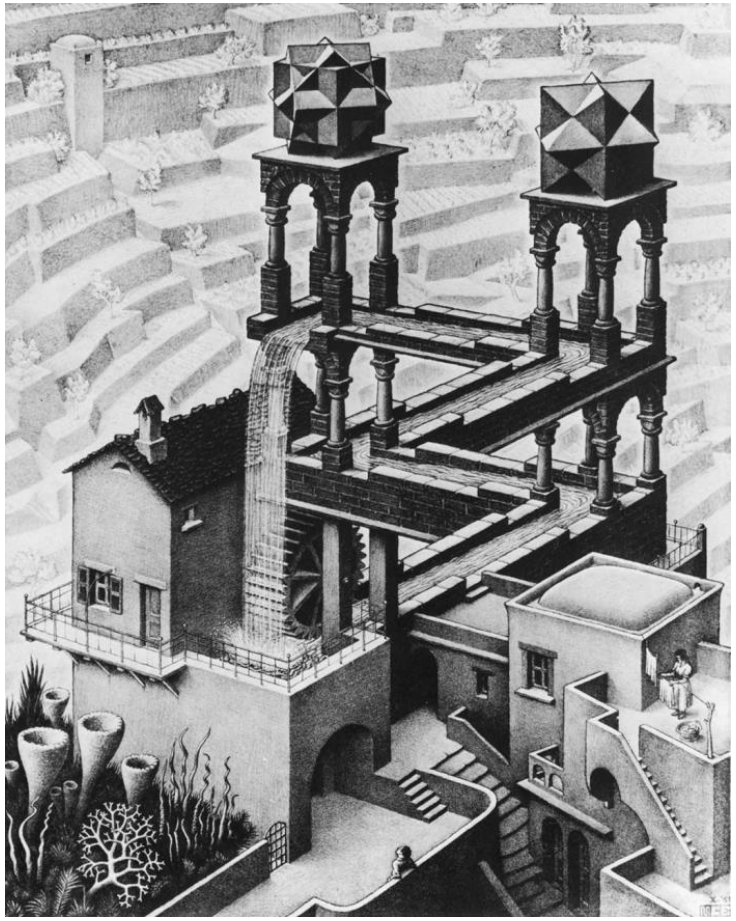
```
char* s2 = "Rio de Janeiro";
```

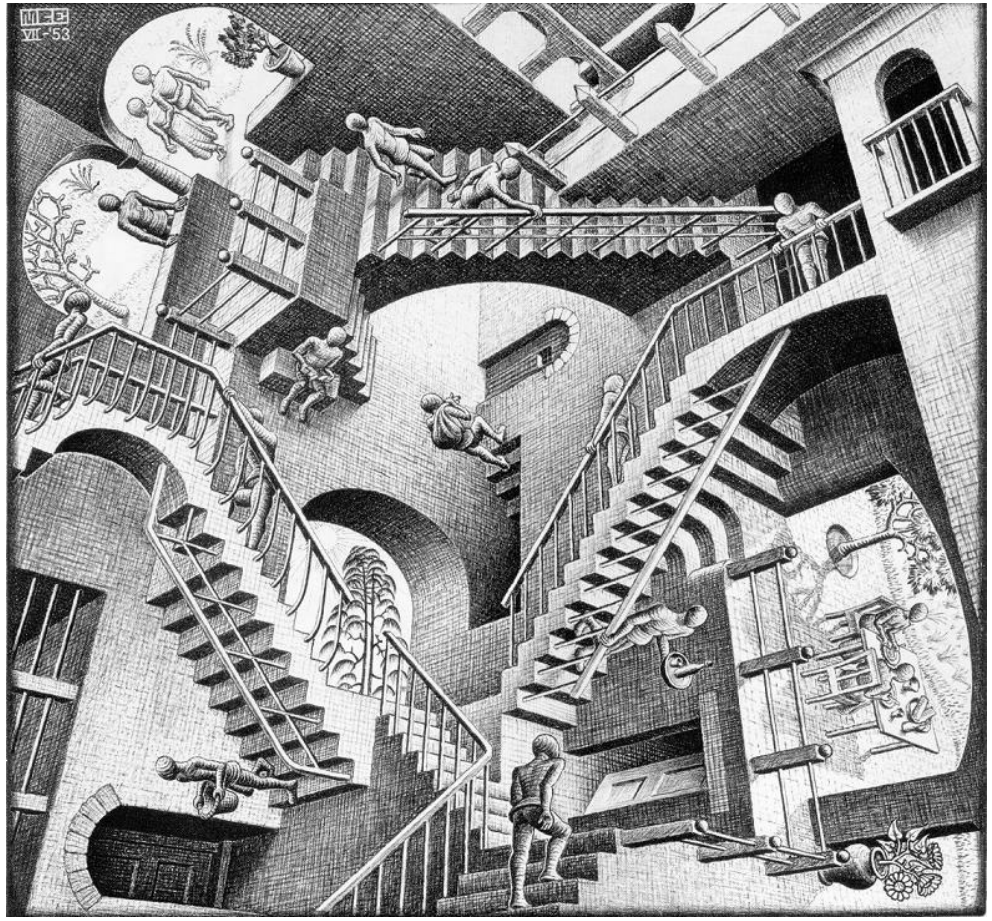
  - s2 é um ponteiro para char, inicializado com o endereço da área de memória onde a constante **`Rio de Janeiro`** está armazenada
  - s2 ocupa 4 bytes (espaço de um ponteiro)
  - não é válido escrever s2[0]='X‘, pois não é possível alterar um valor constante

Revisão Linguagem C

RECURSÃO

Introdução





Definições Recursivas

- Em uma definição recursiva um item é definido em termos de si mesmo, ou seja, **o item que está sendo definido aparece como parte da definição**;

- Em todas as funções recursivas existe:
  - Caso base (um ou mais) cujo resultado é imediatamente conhecido;
  - Passo recursivo em que se tenta resolver um sub-problema do problema inicial.

Definições Recursivas

- Função fatorial: $fat(n) = n \times n - 1 \times \ldots \times 1$

- A função fatorial pode ser definida recursivamente como:

- $$fat(n) = \begin{cases} 1 & , se\ n = 0 \\ n \times fat(n-1), & \underbrace{se\ n > 0}_{\text{Chamada Recursiva}} \end{cases}$$

  ← Caso Base

  ← Passo Recursivo

- O **caso base** é uma situação trivial da função, onde calcular o valor da função é imediato e direto.

Funções Recursivas

- Na programação, função recursiva é aquela que faz uma chamada para si mesma;

- Essa chamada pode ser:
  - direta: uma função A chama a ela própria
  - indireta: função A chama uma função B que, por sua vez, chama A

```
/* Recursao direta */
void func_rec(int n)
{
   ...
   func_rec(n-1);
   ...
}
```

Funções Recursivas

- Função recursiva para cálculo de fatorial:

$$fat(n) = \begin{cases} 1 & , se\ n = 0 \\ n \times \underbrace{fat(n-1)}_{\text{Chamada Recursiva}} & , se\ n > 0 \end{cases}$$

Caso Base

Passo Recursivo

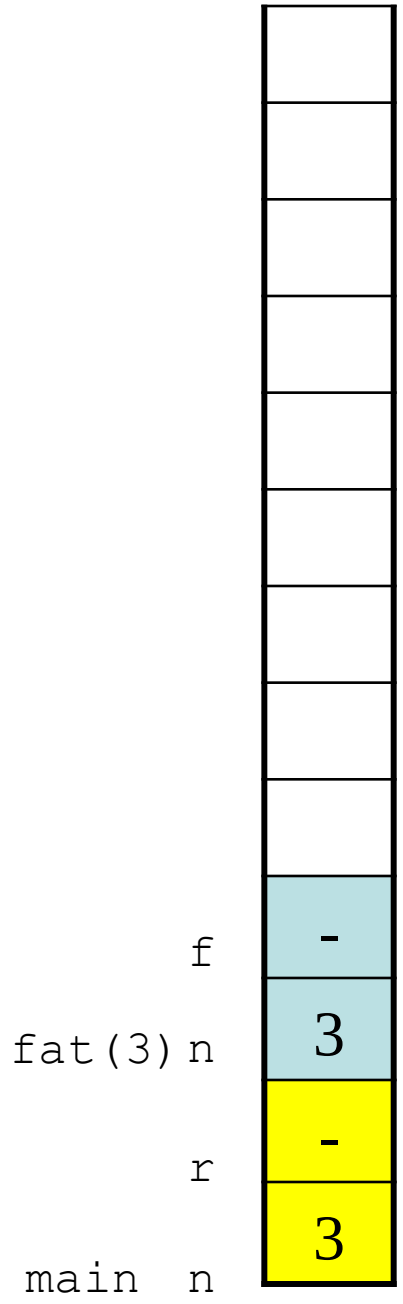
```
/* Função recursiva para cálculo do fatorial */
int fat (int n)
{
   if (n==0)
      return 1;
   else
      return n*fat(n-1);
}
```

Caso BASE

Passo Recursivo

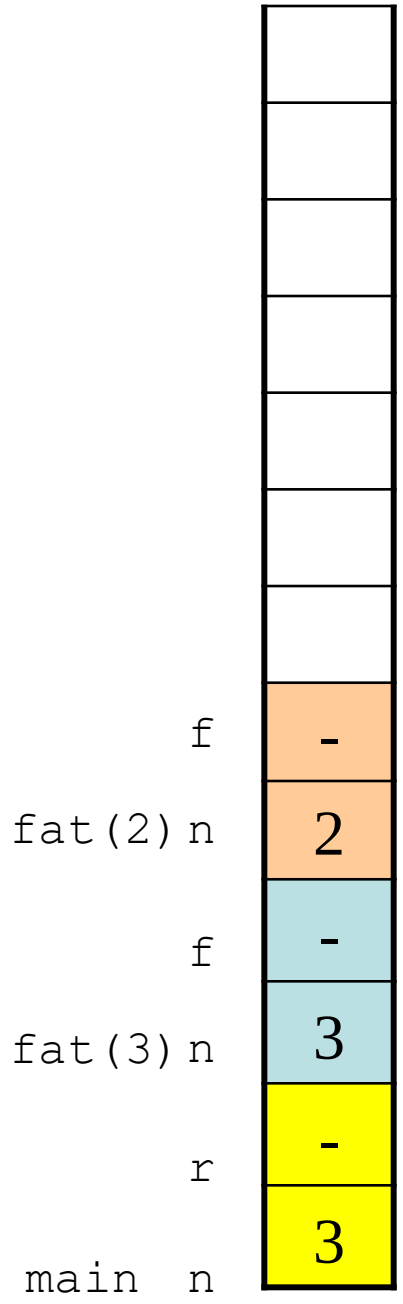
Funções Recursivas

```
#include <stdio.h>
int fat (int n);
int main (void)
{  int n = 3;
   int r;
   r = fat ( n );
   printf("Fatorial de %d = %d \n", n, r);
   return 0;
}

/* Função recursiva para cálculo do fatorial */
int fat (int n)
{
   int f;
   if (n==0)
      f=1;
   else
      f= n*fat(n-1);
   return f;
}
```



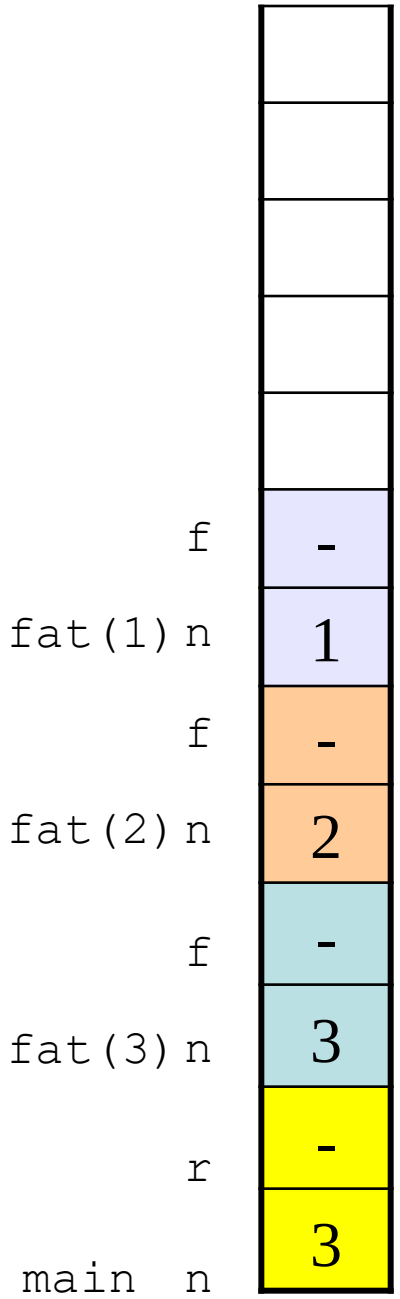
Funções Recursivas

```
#include <stdio.h>
int fat (int n);
int main (void)
{   int n = 3;
    int r;
    r = fat ( n );
    printf("Fatorial de %d = %d \n", n, r);
    return 0;
}

/* Função recursiva para cálculo do fatorial */
int fat (int n)
{
    int f;
    if (n==0)
        f=1;
    else
        f= n*fat(n-1);
    return f;
}
```



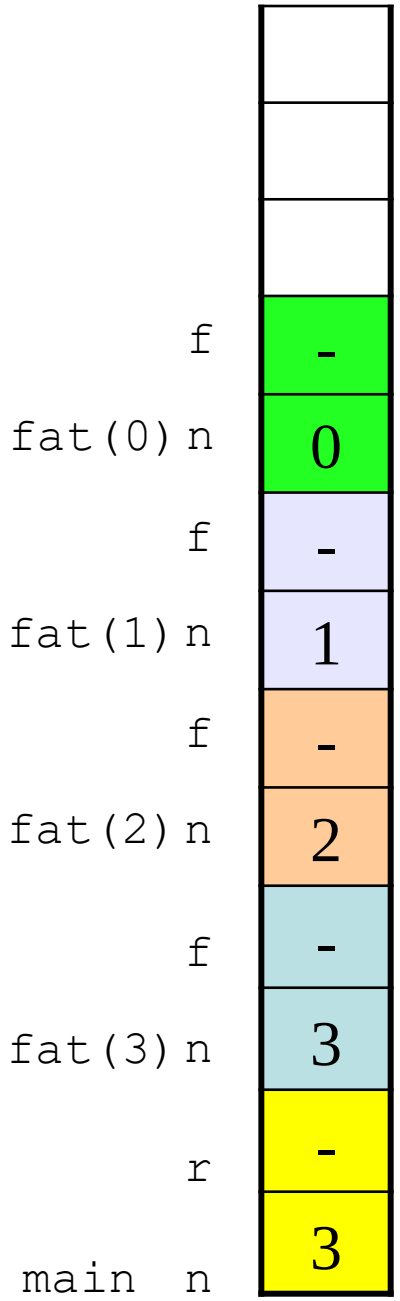
Funções Recursivas

```
#include <stdio.h>
int fat (int n);
int main (void)
{   int n = 3;
    int r;
    r = fat ( n );
    printf("Fatorial de %d = %d \n", n, r);
    return 0;
}

/* Função recursiva para cálculo do fatorial */
int fat (int n)
{
    int f;
    if (n==0)
        f=1;
    else
        f= n*fat(n-1);
    return f;
}
```



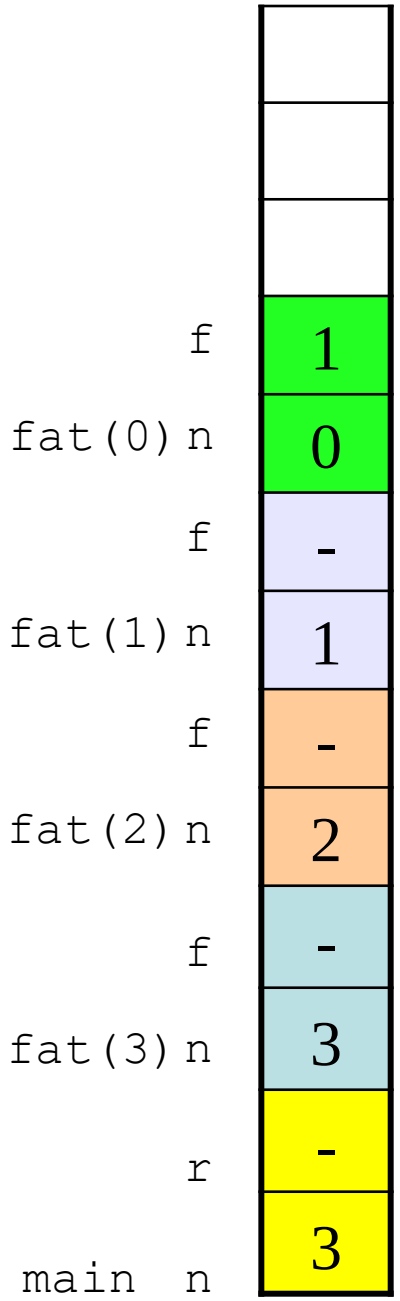
Funções Recursivas

```
#include <stdio.h>
int fat (int n);
int main (void)
{  int n = 3;
   int r;
   r = fat ( n );
   printf("Fatorial de %d = %d \n", n, r);
   return 0;
}

/* Função recursiva para cálculo do fatorial */
int fat (int n)
{
   int f;
   if (n==0)
      f=1;
   else
      f= n*fat(n-1);
   return f;
}
```



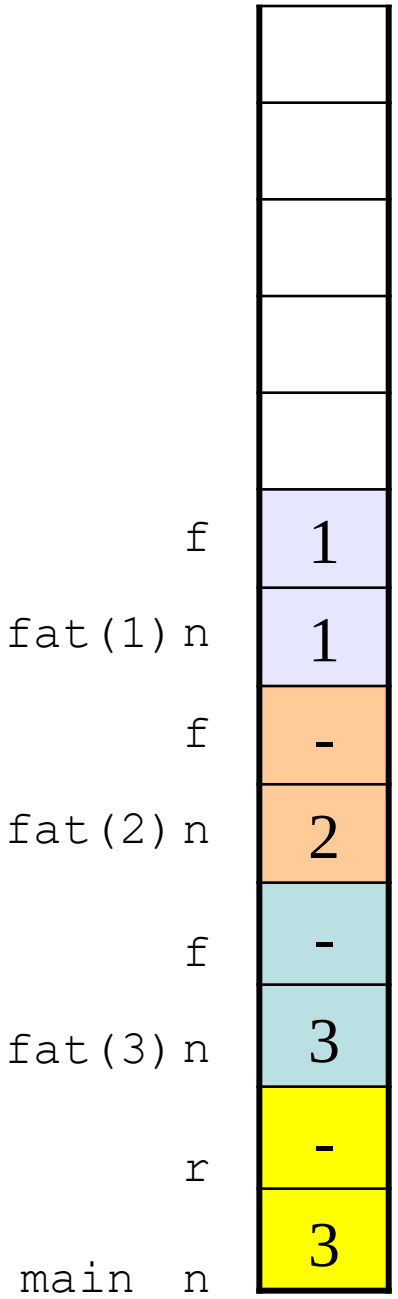
Funções Recursivas

```
#include <stdio.h>
int fat (int n);
int main (void)
{   int n = 3;
    int r;
    r = fat ( n );
    printf("Fatorial de %d = %d \n", n, r);
    return 0;
}

/* Função recursiva para cálculo do fatorial */
int fat (int n)
{
    int f;
    if (n==0)
       f=1;
    else
       f= n*fat(n-1);
    return f;
}
```



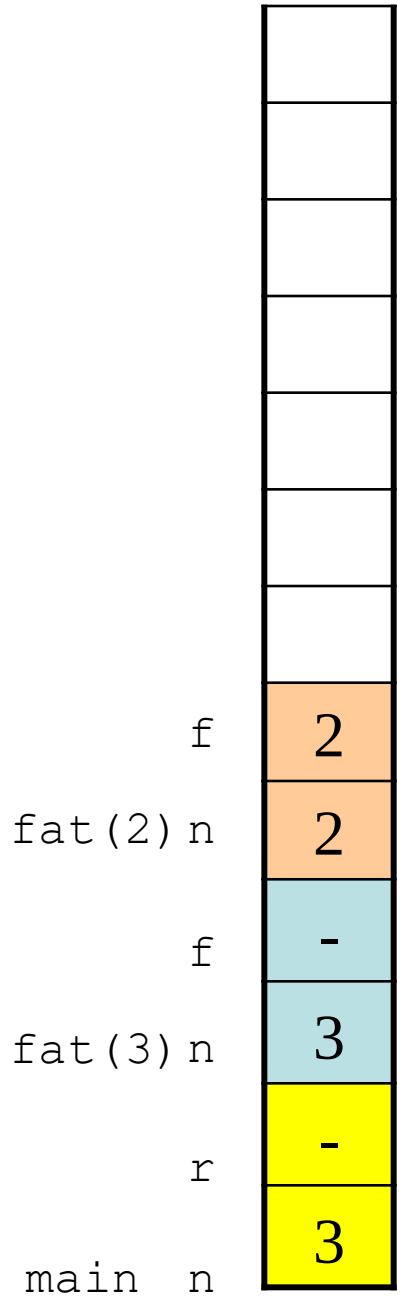
Funções Recursivas

```
#include <stdio.h>
int fat (int n);
int main (void)
{   int n = 3;
    int r;
    r = fat ( n );
    printf("Fatorial de %d = %d \n", n, r);
    return 0;
}

/* Função recursiva para cálculo do fatorial */
int fat (int n)
{
    int f;
    if (n==0)
        f=1;
    else
        f= n*fat(n-1);
    return f;
}
```



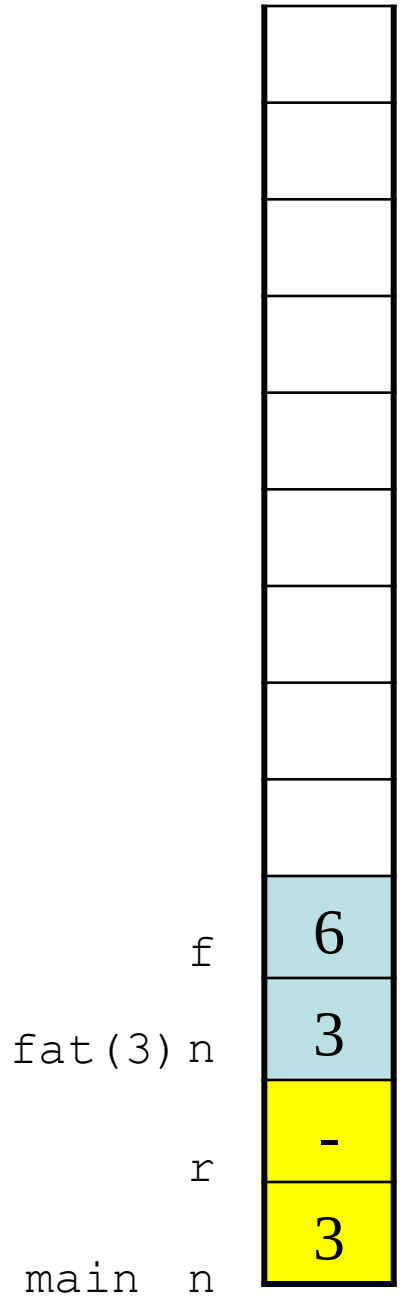
Funções Recursivas

```
#include <stdio.h>
int fat (int n);
int main (void)
{   int n = 3;
    int r;
    r = fat ( n );
    printf("Fatorial de %d = %d \n", n, r);
    return 0;
}

/* Função recursiva para cálculo do fatorial */
int fat (int n)
{
    int f;
    if (n==0)
        f=1;
    else
        f= n*fat(n-1);
    return f;
}
```



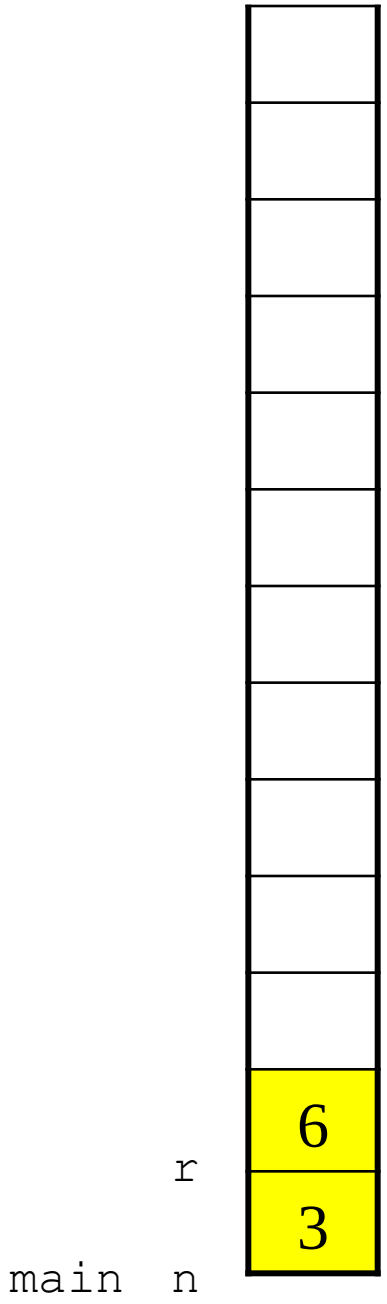
Funções Recursivas

```
#include <stdio.h>
int fat (int n);
int main (void)
{  int n = 3;
   int r;
   r = fat ( n );
   printf("Fatorial de %d = %d \n", n, r);
   return 0;
}

/* Função recursiva para cálculo do fatorial */
int fat (int n)
{
   int f;
   if (n==0)
      f=1;
   else
      f= n*fat(n-1);
   return f;
}
```



Funções Recursivas

```
#include <stdio.h>
int fat (int n);
int main (void)
{  int n = 3;
   int r;
   r = fat ( n );
   printf("Fatorial de %d = %d \n", n, r);
   return 0;
}

/* Função recursiva para cálculo do fatorial */
int fat (int n)
{
   int f;
   if (n==0)
      f=1;
   else
      f= n*fat(n-1);
   return f;
}
```



Funções Recursivas

- Exemplo: série de Fibonacci

$$fib(n) = \begin{cases} 0, \text{se } n = 0 \\ 1, \text{se } n = 1 \\ fib(n-1) + fib(n-2), \text{se } n > 1 \end{cases}$$

2 casos BASE

Passo Recursivo

Manipulação de Cadeias de Caracteres

- É possível utilizar **funções recursivas** para manipular cadeias de caracteres;

- Baseiam-se em uma definição recursiva de cadeias de caracteres:
  - Uma cadeia de caracteres é:
    - a cadeia de caracteres vazia; ou
    - um caractere seguido de uma cadeia de caracteres.

Manipulação de Cadeias de Caracteres

- Implementação recursiva da função “imprime invertido”:

```
void imprime_inv(char* s)
{
  if (s[0] != '\0')
  {
     imprime_inv(&s[1]);
     printf("%c", s[0]);
  }
}
```

```
int main (void)
{
  char cidade[] = "Rio de Janeiro";
  imprime_inv(&cidade[0]);
  return 0;
}
```

Revisão Linguagem C

ESTRUTURAS

Dados Compostos

- Até agora somente utilizamos tipos de dados simples:
  - `char, int, float, double`.
- Muitas vezes precicamos manipular **dados compostos ou estruturados**.
- Exemplos:
  - ponto no espaço bidimensional:
    - representado por duas coordenadas (x e y), mas tratado como um único objeto (ou tipo)
  - registro a aluno:
    - aluno representado pelo seu nome, número de matrícula, endereço, etc ., estruturados em um único objeto (ou tipo)

Ponto

X
Y

Aluno

Nome	
Matr	
End	Rua
	No
	Compl

Tipo Estrutura

- Tipo estrutura (`struct`):
  - tipo de dado com campos compostos de tipos mais simples
  - elementos acessados através do operador acesso “ponto”(.)

```
struct ponto        /* declara ponto do tipo struct */
{   float x;
    float y;
};

int main(void)
{
    struct ponto p; /* declara p como variável do tipo struct ponto */
    p.x = 10.0;     /* acessa os elementos de ponto */
    p.y = 5.0;
    printf(“%f %f”, p.x, p.y);
}
```

Tipo Estrutura - Exemplo

```
/* Captura e imprime as coordenadas de um ponto qualquer */
#include <stdio.h>
struct ponto {
   float x;
   float y;
};
int main (void)
{
   struct ponto p;
   printf("Digite as coordenadas do ponto(x y): ");
   scanf("%f %f", &p.x, &p.y);
   printf("O ponto fornecido foi: (%.2f,%.2f)\n", p.x, p.y);
   return 0;
}
```

Basta escrever &p.x em lugar de &(p.x).

O operador de acesso ao campo da estrutura tem precedência sobre o operador “endereço de”

Ponteiro de Estruturas

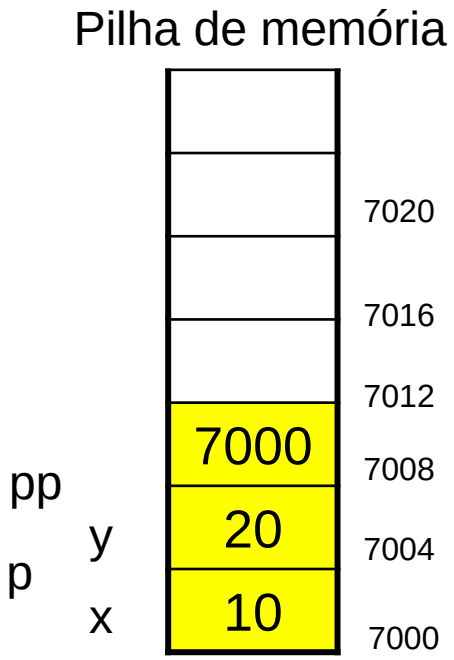
- É possível utilizar ponteiros para estruturas:

```
struct ponto p;
struct ponto *pp=&p;

...

/* formas equivalentes de acessar o valor de um campo x */
(*pp).x = 12.0;
pp->x = 12.0;
p.x = 12.0;
(&p)->x = 12.0;
```

- Ponteiros para estruturas:
  - acesso ao *valor* de um campo x de uma variável estrutura p: `p.x`
  - acesso ao *valor* de um campo x de uma variável ponteiro pp: `pp->x`
  - acesso ao *endereço* do campo x de uma variável ponteiro pp: `&pp->x`

Ponteiro de Estruturas

```
struct ponto {
   float x;
   float y;
};

int main ( )
{
 struct ponto p = { 10.,20};
 struct ponto *pp=&p;
 ...
}
```



- Qual o valor de...?

p.y pp->x &(pp->y) &((&p)->y)

~~pp.x~~ (&p)->x &(p.y)

Alocação dinâmica de estruturas

- É possível alocar estruturas dinamicamente:
  - o tamanho do espaço de memória alocado dinamicamente é dado pelo operador sizeof aplicado sobre o tipo estrutura;
  - a função `malloc` retorna o endereço do espaço alocado, que é então convertido para o tipo ponteiro da estrutura.

```
struct ponto* p;
p = (struct ponto*) malloc (sizeof(struct ponto));

...
p->x = 12.0;
...
free(p);
```

Definição de Novos Tipos

- É possível utilizar o comando `typedef` para definir o nome de um tipo estruturado. Exemplo:

```
struct ponto {
   float x;
   float y;
};

typedef struct ponto Ponto;
typedef struct ponto *PPonto;
```

`ponto` representa uma estrutura com 2 campos do tipo `float`

`Ponto` representa o tipo de estrutura `ponto`

`PPonto` representa o tipo ponteiro para a estrutura `Ponto`

- Simplifica a utilização da estrutura ponto:

```
Ponto p1;
PPonto p2;
p1.x = 10.0;
p2 = (PPonto)malloc(sizeof(Ponto));
p2->x = 5.0;
```

Definição de Novos Tipos

- É possível combinar o comando `typedef` com a declaração da estrutura:

```
typedef struct ponto
{
   float x;
   float y;
} Ponto;
```

  - `struct ponto` representa uma estrutura com 2 campos do tipo `float`
  - `Ponto` representa o tipo de estrutura `ponto`

Estruturas Aninhadas

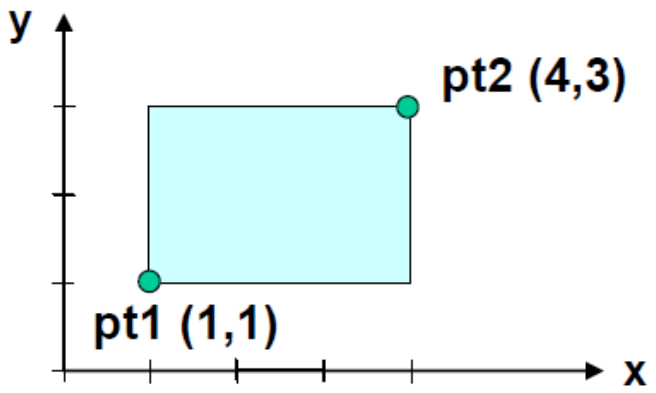
- Os campos de uma estrutura podem ser outras estruturas
  - Exemplo: Definição de um círculo composto por um ponto central e um raio.

```
struct ponto
{
   float x;
   float y;
};
typedef struct ponto Ponto;

struct circulo
{
   Ponto p;  /* centro do círculo */
   float r;  /* raio do círculo */
};
typedef struct circulo Circulo;
```

Estruturas Aninhadas

- Estrutura de um retângulo:

```
struct ponto
{
   float x;
   float y;
};
typedef struct ponto Ponto;

struct retangulo
{
   Ponto p1;
   Ponto p2;
};
typedef struct retangulo Retangulo;
```




```
Retangulo meu_retangulo;
meu_retangulo.pt1.x = 1.0;
meu_retangulo.pt1.y = 1.0;
meu_retangulo.pt2.x = 4.0;
meu_retangulo.pt2.y = 3.0;
```

Revisão Linguagem C

VAMOS EXERCITAR?

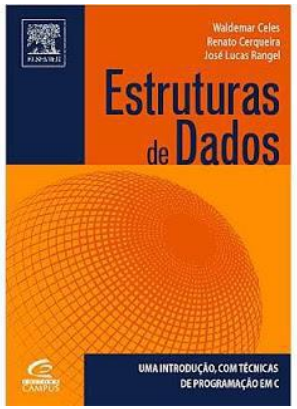
Exercíco: Vetores de Ponteiros para Estruturas

- **Exemplo:** Crie um programa para registrar uma tabela com dados de alunos.
  - Deverá ser organizada em um vetor de ponteiros.
  - Utilizando a estrutura de dados abaixo.
  - Faça todas as alocações de memória dinamicamente.
  - Escreva as funções de manipulação de strings de forma recursiva:
    - Imprime String
    - Comprimento de String

  - Estrutura de dados de cada aluno:
    - matrícula: número inteiro
    - nome: cadeia com até 80 caracteres
    - endereço: cadeia com até 120 caracteres
    - telefone: cadeia com até 20 caracteres

- Obs: Não é necessário criar uma interface para coletar os dados que serão inseridos para testar a tabela com os dados. Podem ser inseridos programaticamente (hard-coded).

Leitura Complementar



- Celes, W., Cerqueira, R., Rangel, J.L., **Introdução a Estruturas de Dados – Uma introdução com técnicas de programação em C**, Ed. Campus, 2004
  - **Ponteiros:** Capítulo 4 – Funções: Ponteiros e Endereços de Variáveis
  - **Alocação Dinâmica:** Capítulo 5 – Vetores e alocação dinâmica
  - **Strings:** Capítulo 7 – Cadeias de caracteres
  - **Estruturas:** Capítulo 8 – Tipos Estruturados
  - **Recursão:** Capítulo 4 – Funções: Recursividade