

Introdução à linguagem C

Diego Raphael Amancio

Baseado no material do Prof. Thiago A. S. Pardo e do Prof. André Backes



Ponteiros

- Ponteiro é uma variável que **guarda o endereço** de memória de outra variável.

Memória

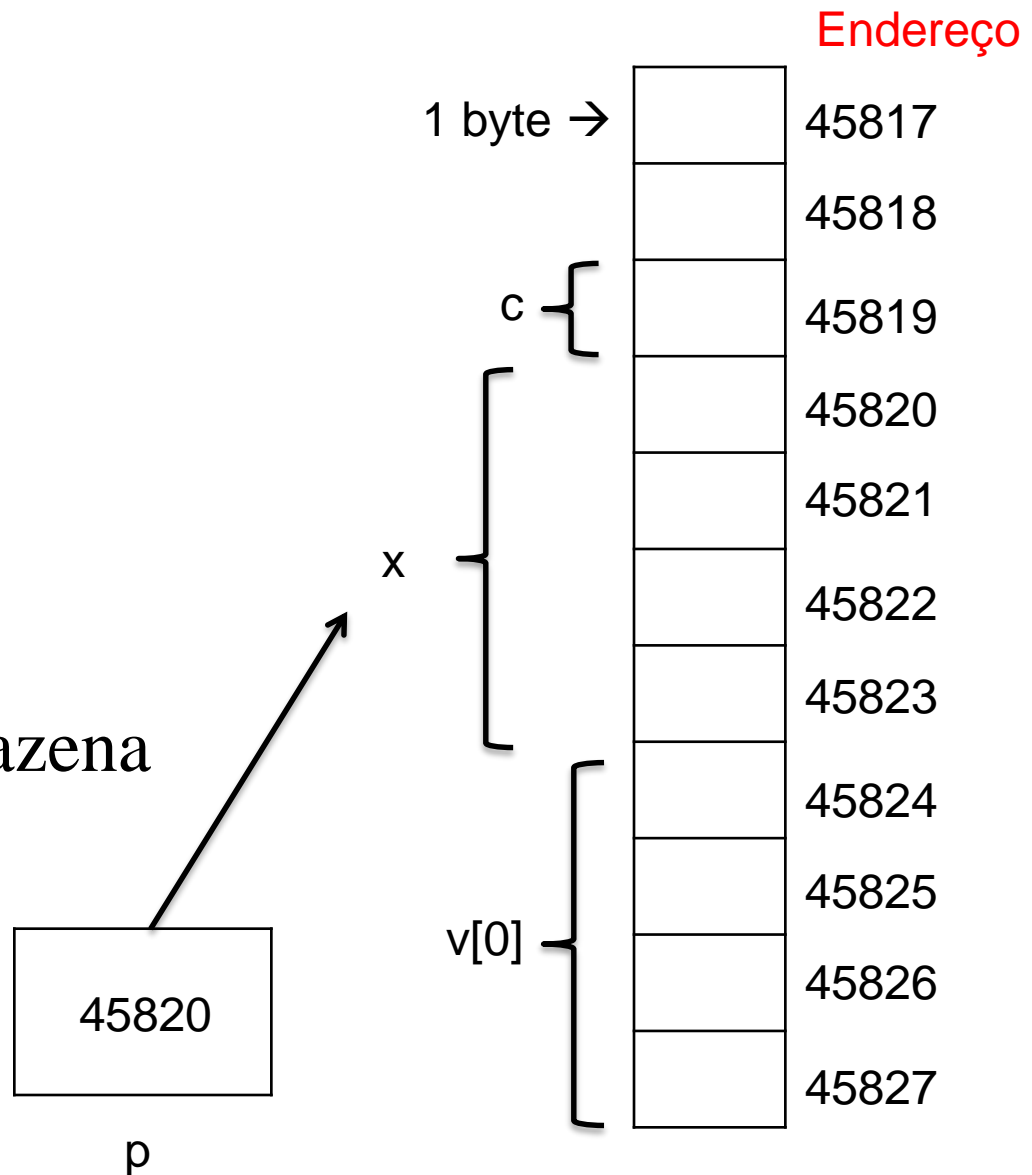
```
char c;
```

```
int x;
```

```
int v[10];
```

- Um ponteiro armazena **endereços**

```
int *p = &x;  
//p = 45820
```



Sintaxe

- Forma geral: **tipo *identificador;**
 - **tipo**: qualquer tipo válido em C.
 - **identificador**: qualquer identificador válido em C.
 - *****: símbolo para declaração de ponteiro. Indica que o **identificador** aponta para uma variável do tipo **tipo**.
- Exemplo:
 - `int *p;`

Operadores de Ponteiros

■ Os operadores de ponteiros são:

□ &

□ *

Operador &

- &: operador unário
- Devolve o endereço de memória de seu operando.

- ☐ Usado durante inicializações de ponteiros.

- ☐ Exemplos

```
int *p, acm = 35;
```

```
p = &acm; /*p recebe “o endereço de” acm*/
```

Operador *

- Devolve o **valor** da variável apontada.

- Ex.:

```
int *p, q, acm = 35;
```

```
p = &acm;
```

```
q = *p;    /* q recebe o conteúdo da variável  
            “no endereço” p */
```

- O valor de q é 35

Exercício

- Seja a seguinte seqüência de instruções em um programa C:

```
int *pti;  
int i = 10;  
pti = &i;
```

Qual afirmativa é **falsa**?

- ☐ a. pti armazena o endereço de i
- ☐ b. *pti é igual a 10
- ☐ c. ao se executar *pti = 20; i passará a ter o valor 20
- ☐ d. ao se alterar o valor de i, *pti será modificado
- ☐ e. pti é igual a 10

Ponteiros genéricos

- Pode apontar para todos os tipos de dados existentes ou que serão criados

```
void *ptr;
```

Ponteiros genéricos

```
void *pp;
```

```
int *p1, p2 = 10;
```

```
p1 = &p2;
```

```
pp = &p2;           //Endereço de int
```

```
pp = &p1;           //Endereço de int *
```

```
pp = p1;            // Endereço de int
```

Ponteiros genéricos

- Acesso depende do **tipo**!

```
void *pp;
```

```
int p2 = 10;
```

```
pp = &p2;
```

```
printf ( "%d\n", *pp );
```

Ponteiros genéricos

- Acesso depende do **tipo**!

```
void *pp;
```

```
int p2 = 10;
```

```
pp = &p2;
```

```
printf ( "%d\n", *pp ); //Erro
```

```
printf( "%d\n", * ( (int *) pp ) );
```

Ponteiros genéricos

- Aritmética de ponteiros:

```
void *p = 0x9C4; //2500
```

```
p++; //2501 -- Sempre soma 1 byte
```

```
p = p + 15; //2516
```

```
p--; // 2515 -- Sempre subtrai um byte
```

O programador deve considerar o tipo

Ponteiros para ponteiros

Armazena o endereço de ponteiros

```
int x = 10
```

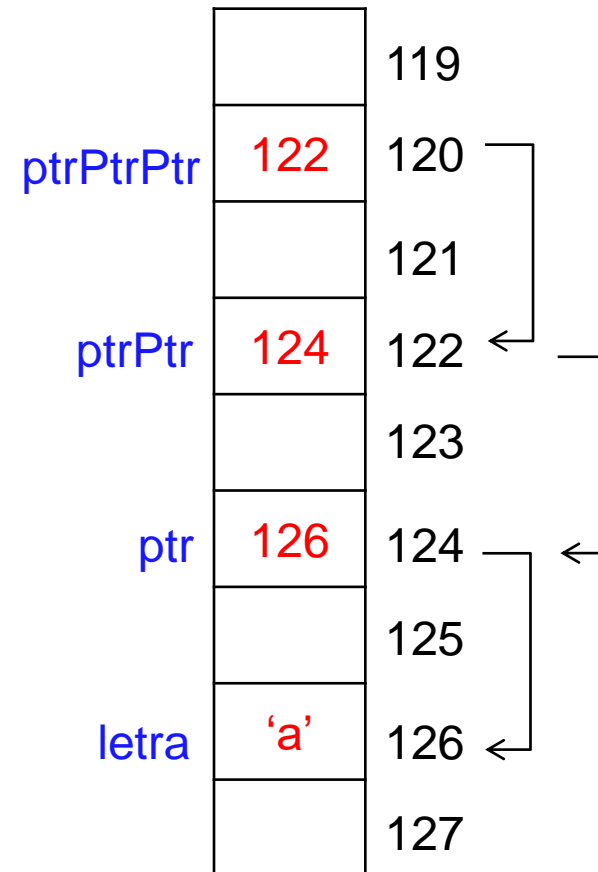
```
int *p = &x;
```

```
int **p2 = &p;
```

END	VARIÁVEL	CONTEÚDO
130	<u>int **p2</u>	132
131		
132	<u>int *p</u>	134
133		
134	<u>int x</u>	10
135		

Ponteiro para ponteiro

```
char letra = 'a';  
  
char *ptr = &letra;  
char **ptrPtr = &ptr;  
char ***ptrPtrPtr = &ptrPtr;
```



Ponteiros para ponteiros

Armazena o endereço de ponteiros

```
int *p;
```

```
int **r;
```


```
p = &a;
```

```
r = &p;
```

```
c= **r + b
```


Exercício

1. Escreva uma função mm que receba um vetor inteiro $v[0..n-1]$ e os endereços de duas variáveis inteiras, digamos min e max, e deposite nessas variáveis o valor de um elemento mínimo e o valor de um elemento máximo do vetor. Escreva também uma função main que use a função mm.
2. **Escreva uma função que troque os valores de duas variáveis**



Alocação dinâmica


- Para que serve?

Alocação dinâmica

- Cadastro de funcionários de uma empresa?

```
struct func_type funcionarios[1000];
```

Pode haver desperdício ou faltar espaço.



Malloc (stdlib.h)

```
void *malloc ( unsigned int num );
```

```
//array de 50 inteiros
```

```
int *v = (int *) malloc ( 200 );
```

```
//string de 200 caracteres
```

```
char *c = (char *) malloc( 200 );
```

Para reservar memória dinamicamente

- Para reservar memória dinamicamente num vetor, usa-se ponteiros da seguinte forma:

```
float *x;
```

```
x = ( float * ) malloc ( nx * sizeof ( float ) ) ;
```

nx é o numero de elementos que vai ter o array

Para reservar memória dinamicamente

- Para liberar a memória alocada dinamicamente:

```
float *x;
```

```
x = ( float * ) malloc ( nx * sizeof ( float ) ) ;
```

```
//nx é o numero de elementos que vai ter o array
```

```
free(x);
```


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

int main()
{
    //Se não tiver memória suficiente,
    //retorna NULL
    int *p = (int *) malloc( 5 * sizeof(int) );
    if ( p == NULL ) exit(1);

    int i;
    for ( i = 0; i < 5; i++ )
        scanf( "%d", &p[i] );

    //Libera memória
    free(p);

    return 0;
}
```



calloc (stdlib.h)

```
void *calloc (unsigned int num,  
              unsigned int size );
```

num: número de unidades alocadas

size: tamanho de cada unidade


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

int main()
{

    //Se não tiver memória suficiente,
    //retorna NULL
    int *p = (int *) calloc( 5, sizeof(int) );
    if ( p == NULL ) exit(1);

    int i;
    for ( i = 0; i < 5; i++ )
        scanf( "%d", &p[i] );

    //Libera memória
    free(p);

    return 0;
}
```



malloc vs. calloc

- Malloc

- ☐ Apenas aloca a memória

- Calloc

- ☐ Aloca a memória
- ☐ Inicializa todos os bits da memória com zero

realloc (stdlib.h)

- Útil para **alocar** ou **realocar** memória durante a execução.
 - Ex.: Aumentar a quantidade de memória já alocada.

```
void *realloc ( void *ptr, unsigned int num );
```

OS DADOS EM PTR NÃO SÃO PERDIDOS

realloc

```
void *realloc ( void *ptr, unsigned int num );
```

- ptr: ponteiro para bloco já alocado
- num: número de bytes a ser alocado
- Retorna ponteiro para primeira posição do array ou NULL caso não seja possível alocar

```
int *v = (int *) malloc ( 50 * sizeof(int) );  
v = (int *) realloc( v, 100 * sizeof(int) );
```

realloc

- Se `ptr == NULL`, então `realloc` funciona como `malloc`

```
int *p;
```

```
p = (int *) realloc( NULL, nx * sizeof ( int ) );
```

```
p = (int *) malloc( nx * sizeof(int) );
```

realloc

- Realloc pode ser utilizado para liberar a memória

```
int *p = ( int * ) malloc ( nx * sizeof ( int ) );  
p = ( int * ) realloc ( p, 0 );
```

realloc

- Realloc pode ser utilizado para liberar a memória

```
int *p = ( int * ) malloc ( 50 * sizeof ( int ) );  
p = ( int * ) realloc ( p, 100 * sizeof ( int ) );
```

QUAL O PROBLEMA DO CÓDIGO ACIMA ?

realloc

- Realloc pode ser utilizado para liberar a memória

```
int *p = ( int * ) malloc ( 50 * sizeof ( int ) );  
p = ( int * ) realloc ( p, 100 * sizeof ( int ) );
```

PONTEIRO PODE SER NULL E O VETOR ALOCADO
ANTERIORMENTE É PERDIDO

Ponteiros: utilidade

- Retornar mais de um valor em uma função

- Exemplo: swap(a,b)

- Alias de vetor

```
int v[5] = {1,2,3,4,5}
```

```
//v é um ponteiro que aponta para v[0]
```

```
// v[3] == *(v+3)
```

Ponteiros e vetores

```
int v[5] = {1,2,3,4,5}
```

- $v+i$ equivale a $\&v[i]$
- $*(v+i)$ equivale a $v[i]$

Ponteiros como aliases

Se *v* é um vetor de inteiros, então há equivalência?

```
int v[] = {1,2,3};  
printf( "%d", *(v+1) );  
printf( "%d", *(++v) );
```

Ponteiros constantes

Se *v* é um vetor de inteiros, então há equivalência?

```
int v[] = {1,2,3};  
printf( "%d", *(v+1) );  
printf( "%d", *(++v) );
```

Não há equivalência, pois o nome do vetor
é um ponteiro constante

Operações com ponteiros

```
int *px, *py;
```

```
....
```

```
if( px < py )
```

```
px = py + 5;
```

```
px - py; //número de variáveis entre px e py
```

```
px++;
```

Obs.: se px é ponteiro para int, incrementa
o tamanho de um inteiro



Arrays de ponteiros

Exemplo:

Criação de matriz de inteiros

```
int *vet [ size ];
```

Cada posição pode ser alocada
dinamicamente

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

```
int main()
```

```
{
```

```
    int *pvet[2];
```

```
    int x = 10;
```

```
    int y[2] = {20, 30};
```

```
    pvet[0] = &x;
```

```
    pvet[1] = y;
```

```
    printf( "pvet[0]: %p\n", pvet[0] ); // &x
```

```
    printf( "pvet[1]: %p\n", pvet[1] ); // &y[0]
```

```
    printf( "*pvet[0]: %d\n", *pvet[0] ); //x
```

```
    printf( "pvet[1][1]: %d\n", pvet[1][1] ); //y[1]
```

```
}
```



Exercício

- Faça um programa que permita criar uma lista de alunos (gravados pelo seu número USP), sendo que o número de alunos é determinado pelo usuário.
 - Utilize a **alocação dinâmica** de vetores



Estruturas

- **Coleção de variáveis** organizadas em um único conjunto.
 - Possivelmente coleção de tipos distintos
- As variáveis que compreendem uma estrutura são comumente chamadas de **elementos** ou **campos**.

Exemplo-estruturas

■ Definição x Declaração

```
struct pessoa  
{  
    char nome[30];  
    int idade;  
};
```

- Permite declarar variáveis cujo tipo seja **pessoa**.

Usando typedef nas estruturas

```
struct a{  
    int x;  
    char y;  
};
```

```
typedef struct a MyStruct;
```

```
int main(){  
    MyStruct b; /*declaração da var b, cujo tipo é MyStruct*/  
}
```

Acesso aos dados da Estrutura

- É feito via o ponto (.)

```
int main (void){  
    MyStruct obj;  
    obj.x = 10;  
    obj.y = 'a';  
}
```

```

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

struct pessoa {
    char nome[50], rua[50];
    int idade, numero;
};

int main() {

    struct pessoa p;
    strcpy(p.nome, "Nome" );
    strcpy(p.rua, "Street 4" );
    p.idade = 27;
    p.numero = 1874;

    return 0;

}

```

INICIALIZAÇÃO CAMPO A CAMPO

```

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

struct pessoa {
    char nome[50], rua[50];
    int idade, numero;
};

int main() {

    struct pessoa p = { "Nome",
        "Street 4", 27, 1874 };

    //Campos não inicializados
    //explicitamente são inicializados
    //com zero
    struct pessoa p2 = { "Nome2",
        "Street 4", 27 };

    return 0;

}

```

INICIALIZAÇÃO COMO VETOR

**ATRIBUIÇÃO COMO
VARIÁVEL NORMAL**



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

struct ponto {
    int x,y;
};

struct ponto_B {
    int x,y;
};

int main() {

    struct ponto p1, p2 = {1,2};
    struct ponto_B p3 = {3,4};

    p1 = p2; //OK
    p1 = p3; //Erro ! Tipos diferentes

    return 0;
}
```

Aninhamento de estruturas

```
struct tipo_struct1 {...};
```

```
struct tipo_struct2 {  
    ...  
    struct tipo_struct1 nome;  
}
```

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

```
struct endereco {
    char rua[80];
    int numero;
};
```

```
struct pessoa {
    char nome[50];
    int idade;
    struct endereco ender;
};
```

```
int main() {

    struct pessoa p;
    p.idade = 31;
    p.ender.numero = 103;

    return 0;
}
```




Ponteiros para estruturas

- Utilidade?



Ponteiros para estruturas

■ Utilidade

1. Evitar **overhead** em chamadas de funções
2. Criação de **listas encadeadas**

Ponteiros para estruturas

■ Exemplo

...

```
struct bal {  
    char name[30];  
    float balance;  
} person;
```

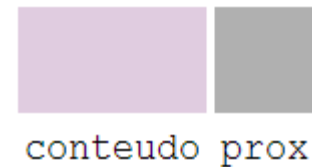
```
struct bal *p;
```

```
p = &person; //Apontando para  
p->balance; //Acesso ao conteúdo
```

Listas encadeadas

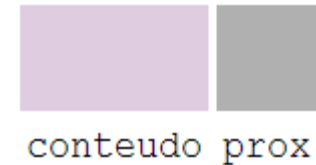
- É uma representação de uma sequência de objetos na memória do computador

```
struct cel {  
    int conteudo;  
    struct cel *prox;  
};  
typedef struct cel celula;
```



Operações

```
celula c;  
celula *p;
```



...

```
p = &c;           //Endereço da célula  
p->conteudo;       //Conteudo atual  
p->prox->conteudo; //Conteudo do próximo
```

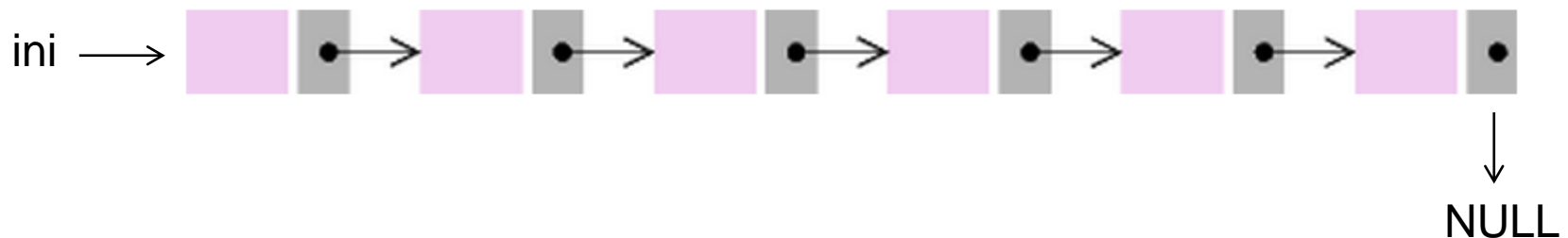
Lista sem cabeça

- Primeiro elemento faz parte da lista

- Inicialização

celula *ini;

ini = NULL //Lista esta vazia



Exercício

- Escreva uma função que imprime o conteúdo de uma lista sem cabeça

```
void imprima ( celula *ini );
```

Exemplo

```
void imprima( celula *ini )  
{  
    celula *p;  
    for (p = ini; p != NULL; p = p->prox)  
        printf( "%d\n", p->conteudo);  
}
```




Exercício

- Escreva uma função que busca o inteiro x na lista. A função devolve um ponteiro para o registro encontrado ou NULL, caso não encontre.

Exercício

- Escreva uma que insira uma nova célula com conteúdo x em uma lista ordenada. A lista deve continuar ordenada após a inserção
 - Utilize alocação dinâmica