



Linguagem de programação

Ponteiros

Introdução

- Ponteiros são um dos recursos mais poderosos da linguagem C; e um dos mais difíceis de dominar
- permitem simular **chamadas por referência**
- permitem criar e manipular **estruturas dinâmicas de dados** (que podem crescer ou diminuir) como:
 - listas encadeadas, filas, pilhas e árvores

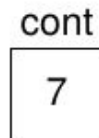
Declarações e Inicialização de Variáveis Ponteiros

- São variáveis que contêm endereços de memória como valor
- Uma variável faz uma referência direta a um valor específico
- Um ponteiro, por outro lado, contém um endereço de uma variável que contém um valor específico.

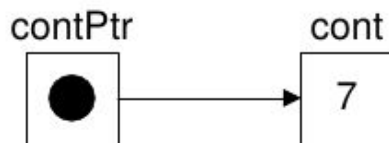
```
int *contPtr, cont;
```

Obs 1.: '*' indica que a variável é um ponteiro.

Obs 2.: o ponteiro deve ser do mesmo tipo que a variável apontada



cont faz uma referência direta a uma variável cujo valor é 7.



contPtr faz uma referência indireta a uma variável cujo valor é 7.

Boas práticas de programação

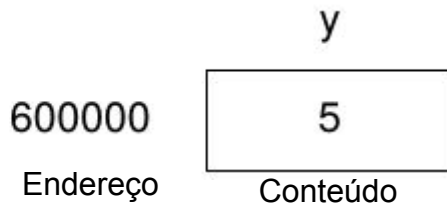
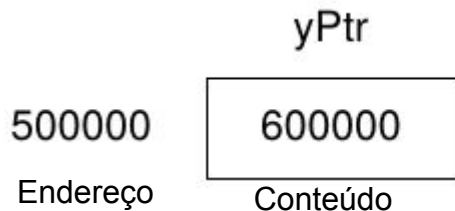
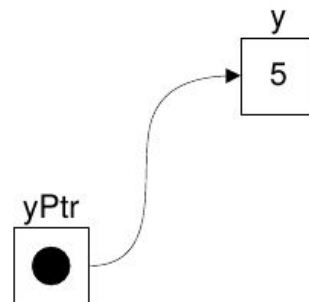
- 1) **Incluir as letras ptr** em nomes de variáveis de ponteiros para tornar claro que essas variáveis são ponteiros e precisam ser manipuladas apropriadamente.

- 1) **Inicialize ponteiros** para evitar resultados inesperados:
 - a) Podem ser inicializados com um **endereço, 0** ou **NULL**
 - b) O mais recomendado é NULL. Quando um ptr recebe NULL, ele não está apontando para lugar algum

Operadores de ponteiros

- `&`: é um operador unário que retorna o endereço do seu operando
- por ex.:

```
int y = 5;  
int *yPtr;  
// yPtr recebe o endereço de y  
// yPtr "aponta" para y  
yPtr = &y;
```



Operadores de ponteiros

- `*` : operador que retorna o valor do objeto ao qual o seu operando aponta.
- Por ex.:

```
int a, *aPtr = NULL;  
a = 7;  
aPtr = &a;
```

```
// endereço de a e valor de aPtr  
printf("%p e %p", &a, aPtr);
```

Ex. de saída: FFF4 e FFF4

```
//valor de a e valor de *aPtr  
printf("%d e %d\n", a, *aPtr);
```

Saída: 7 e 7

Obs.: `%p` mostra o endereço na memória como inteiro hexadecimal

```
//Sabendo que * e & "cancelam-se"  
printf("&*aPtr=%p e *&aPtr=%p", &*aPtr,  
*&aPtr);
```

Ex. de saída: FFF4 e FFF4

Exercício 1

- Qual será a saída do seguinte programa em C?

```
int main() {  
    int *pc, c=5;  
    pc = &c;  
    c = 1;  
    printf("%d %d", c, *pc);  
    return 0; }
```

Resposta:

- a) 5 1
- b) 1 5
- c) 5 5
- d) 1 1

Exercício 1

- Qual será a saída do seguinte programa em C?

```
int main() {  
    int *pc, c=5;  
    pc = &c;  
    c = 1;  
    printf("%d %d", c, *pc);  
    return 0; }
```

Resposta:

- a) 5 1
- b) 1 5
- c) 5 5
- d) 1 1

Relação entre vetor e endereço

- Considere a seguinte leitura de string:

```
char str[20];  
scanf("%s", str);
```

Obs.: como os endereços de cada posição são consecutivos, basta saber a posição 0

- o ‘&’ é utilizado no scanf para fornecer o endereço da variável
- observe que o vetor é passado para o scanf() sem o ‘&’
- o nome de um vetor é o endereço do início do vetor;
 - portanto, o ‘&’ não é necessário
 - **str** equivale a **&str[0]**
 - nome de um vetor é um “**ponteiro**” que aponta para o seu início

Chamando Funções por Referência

- Há 2 maneiras de passar argumentos à uma função: **por valor** e **por referência**

- **Passagem por valor:**

- Como são todas as chamadas de função em C
- No momento em que a função é chamada, uma variável local é criada, na qual é copiado o valor da variável passada como parâmetro
- Alterações na variável local, não refletem na variável “original”

```
void funcao(int a, int b){  
    a += b;  
    printf("Na funcao, a = %d b = %d\n", a,  
b); }
```

```
int main(void){  
    int x = 5, y = 7;  
    funcao(x, y);  
    printf("Na main, x = %d y = %d\n", x,  
y);  
    return 0; }
```

Saída:

Na funcao, a=12 b = 7
Na main, x=5 y = 7

Desvantagens da passagem por valor:

- **Uso ineficiente da memória**, pois a cada argumento tem-se uma variável consumindo memória
 - A situação piora, quando argumentos são vetores
- Para vetores, o **processo de cópia é custoso**
 - Imagine um vetor de 100 posições;
 - a cada chamada da função, 100 operações de cópias serão realizadas.

Chamando Funções por Referência

- O C fornece um meio de simular chamadas por referência, utilizando **ponteiros**
- Em vez do valor, é passado um ponteiro com o endereço da variável
 - permitindo a manipulação direta da variável, e não de uma cópia sua
- Ex.:

```
void troca(int* i, int* j) {  
    int temp = *i;  
    *i = *j;  
    *j = temp;  
}  
  
int main(void) {  
    int *ptrA, *ptrB, a = 10, b = 20;  
    ptrA = &a;  
    ptrB = &b;  
    troca(ptrA, ptrB);  
    printf("a e' %d e b e' %d\n", a, b);  
    return 0; }
```

Expressões e Aritmética de Ponteiros

- Operações aritméticas podem ser realizadas com ponteiros. Um ptr pode ser:
 - incrementado (++) ou decrementado (--);
 - adicionado a um inteiro (+ ou +=);
 - subtraído a um inteiro (- ou -=); ou
 - um ptr pode ser subtraído de outro

Pergunta:

- Considere `int v[100]` no qual o seu início tem endereço 3000

```
int *ptr, v[100]; //considere que &v[0] = 3000
ptr = v; //ou ptr = &v[0];
printf("%d ", ptr); // mostra 3000
ptr += 2;
printf("%d", ptr);
```

- Qual a saída do 2º printf?
(admita 4 bytes p/ armazenar um inteiro na memória)

Resposta:

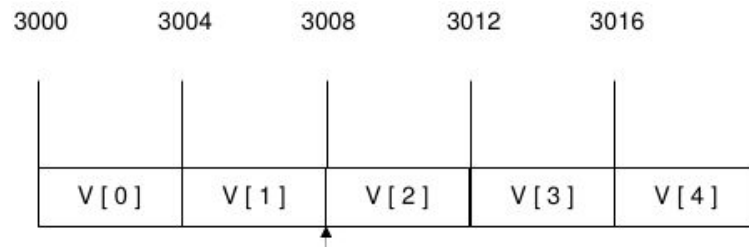
- Considere `int v[100]` no qual o seu início tem endereço 3000

```
int *ptr, v[100]; //considere que &v[0] = 3000
ptr = v; //ou ptr = &v[0];
printf("%d ", ptr); // mostra 3000
ptr += 2;
printf("%d", ptr);
```

- Qual a saída do 2º printf? (admita 4 bytes p/ armazenar um inteiro na memória)

R = 3008.

- Operador aritmético do ponteiro funciona de acordo com o tipo do dado
- `ptr +=2`, o faz apontar para `v[2]`, 8 bytes a mais na memória



$$3008 = 3000 + 2 * 4$$

Exercício 3

- Qual será a saída do seguinte programa em C?

```
int main() {  
    int a = 30, b = 5;  
    int *p = &a, *q = &b;  
    printf("%d", p - q);  
    return 0;  
}
```

- a) 1
- b) Erro de execução
- c) Erro de compilação
- d) 25

Exercício 3

- Qual será a saída do seguinte programa em C?

```
int main() {  
    int a = 30, b = 5;  
    int *p = &a, *q = &b;  
    printf("%d", p - q);  
    return 0;  
}
```

- a) 1
- b) Erro de execução
- c) Erro de compilação
- d) 25

Exercício 3

- Qual será a saída do seguinte programa em C?

```
int main() {  
    int a = 30, b = 5;  
    int *p = &a, *q = &b;  
    printf("%d", p - q);  
    return 0;  
}
```

- a) 1
- b) Erro de execução
- c) Erro de compilação
- d) 25

- Explicação:**

- A declaração consecutiva de variáveis do mesmo tipo, são em endereços também consecutivos;
- A diferença de endereços é expressada em termos de `sizeof(tipo)`
 - nesse caso, `sizeof(int)`
- Logo, 1.

Exercício 4

- Qual será a saída do seguinte programa em C?

```
int main()
{
    char *ptr = "Ponteiro-para-String", i;
    printf("%s", ++ptr);
    return 0;
}
```

- A. Ponteiro-para-String
- B. o
- C. onteiro-para-String
- D. N.D.A.
- E. Erro de compilação

Exercício 4

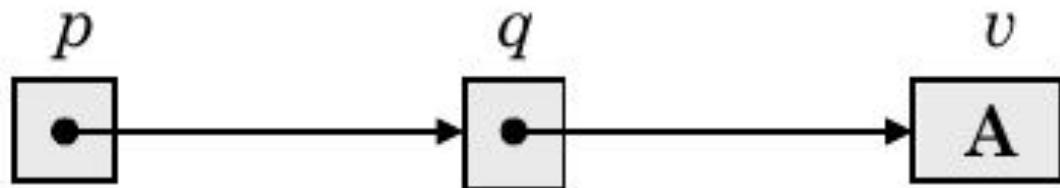
- Qual será a saída do seguinte programa em C?

```
int main()
{
    char *ptr = "Ponteiro-para-String", i;
    printf("%s", ++ptr);
    return 0;
}
```

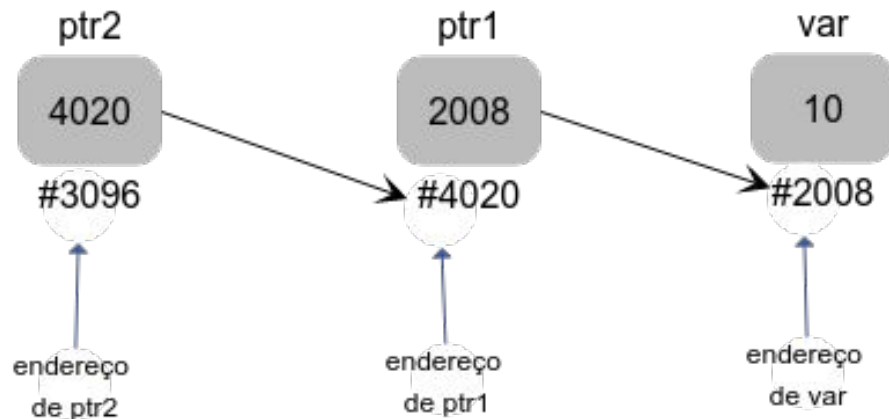
- A. Ponteiro-para-String
- B. o
- C. onteiro-para-String
- D. N.D.A.
- E. Erro de compilação

Ponteiro para ponteiro

- Na linguagem C, um ponteiro pode apontar para outro ponteiro. Permitindo:
 - passagem de matrizes **por referência**
 - **alocação dinâmica** de matrizes
 - operações de manipulação em **estruturas de dados**
 - inserção e remoção em árvores, listas encadeadas, etc.



Ponteiro para ponteiro



- O 1º ponteiro (ptr1) armazena o endereço da variável. E o 2º ponteiro (ptr2), armazena o endereço do 1º ponteiro
- Sintaxe:
 - `int **ptr; // declaração de um prt de ptr`

Exemplo:

Saída:
10 10 10

```
int var = 10;
// ponteiro para var
int *ptr1;
// ponteiro-de-ponteiro para ptr1
int **ptr2;
//armazenando endereço de var em ptr1
ptr1 = &var;
//armazenando endereço de ptr1 em ptr2
ptr2 = &ptr1;
//Mostrando valor de var, var usando
//ponteiro e ponteiro-de-ponteiro
printf("%d %d %d\n", var,*ptr1, **ptr2 );
```

Exercício

- Crie uma função em C para trocar o valor de duas variáveis do tipo *int*, utilizando chamada por referência
 - protótipo da função `void troca(int *x,int *y);`

Exercício

- Crie um programa em C para mostrar na tela todas as letras do alfabeto, utilizando ponteiros

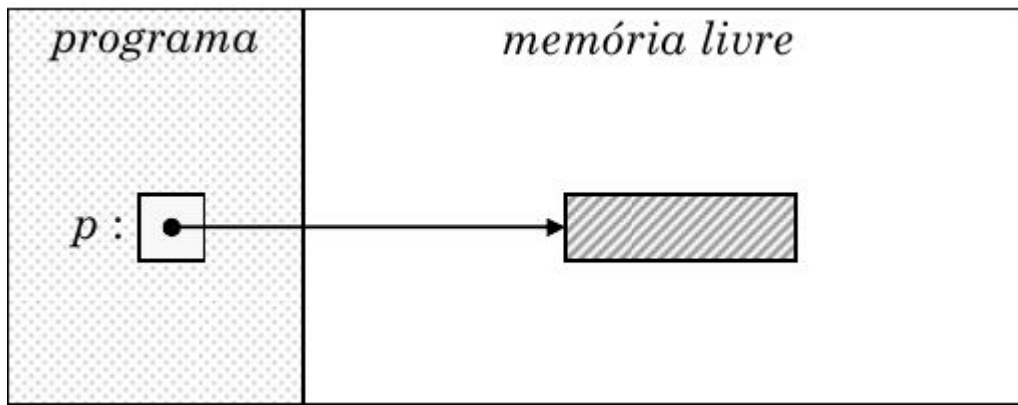
Exercício

- Escreva um programa em C para calcular a soma de todos os elementos em um array usando ponteiros.

Alocação dinâmica

Alocação dinâmica

- É uma das aplicações mais interessantes de ponteiros
- Permite a um programa requisitar **memória adicional** para o armazenamento de dados durante sua execução.



Função *malloc*

- Para requisitar mais espaço de memória, utilizamos a função *malloc()*
- `void* malloc (unsigned int size):`
 - recebe como argumento o tamanho, em bytes, da área a ser alocada
 - Se houver memória disponível, o endereço dessa área é retornado
 - Senão, é retornado NULL
 - Como malloc não sabe o tipo dos dados a serem armazenados, é retornado um ponteiro void*

Função malloc

Ponteiros do tipo `void*` são compatíveis de atribuição com ponteiros de quaisquer outros tipos e, por isso, são denominados ponteiros genéricos.

- Como tipos de dados diferentes requerem espaços diferentes, e ainda podem variar de acordo com a máquina
 - fazemos uso do operador **sizeof** na passagem de parâmetro
 - exemplo alocando espaço para **n** inteiros:
 - `malloc(n*sizeof(int))`

Exemplo: vetores dinâmicos

```
#include <stdio.h>
#include <malloc.h>
int main(void) {
    int *v, n, i;
    printf("\nTamanho do vetor? ");
    scanf("%d",&n);
    v = malloc( n*sizeof(int) );
    if( v==NULL ) return -1;
```

- Como o retorno de malloc é `void*`, o C faz o cast automático para para `int*`
- Então na compilação ocorre:
`v = (int*)malloc(n*sizeof(int));`

```
for(i=0; i<n; i++) {
    printf("\n%d°. Valor? ",i);
    scanf("%d",&v[i]); }
for(i=0; i<n; i++)
    printf("%d ",v[i]);
}
free(v); //libera memória alocada para v
return 0;
}
```

Obs.: recomenda-se sempre desalocar a memória. Cuidado para não desalocar antes do uso!

Variável para tamanho de vetor

- A partir do C99, é permitido declarar o tamanho de um vetor, por meio de uma variável

- ex.:

```
int tam;  
printf("tamanho do vetor\n");  
scanf("%d", &tam);  
int v[tam];
```

- Isso não é boa prática de alocação dinâmica!

- um vez declarado, `v` não pode alterar de tamanho
 - com ponteiros isso é possível usando

realloc()

- Não fornece nenhum mecanismo para detecção de falhas:

```
//se 'tam' ultrapassar o espaço de  
memória disponível, um erro inesperado  
por ocorrer  
char v[tam];  
  
char *v = malloc(tam*sizeof(char));  
if (v == NULL) {  
    //alocação falhou, abortar ou tomar  
    uma ação corretiva }  
}
```


Exemplo: matrizes dinâmicas

```
int **p=NULL;
```

```
int l=5,c=5;
```

```
//1º) malloc aloca espaço para l ponteiros do tipo int. Um "vetor int* com l posições"
```

```
//2º) p aponta para a 1ª posição desse vetor
```

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

```
//1º) Para cada ponteiro p[i], malloc aloca espaço para c inteiros. Um "vetor int com c posições".
```

```
//2º) p[i] aponta para a 1ª posição desse vetor
```

```
for(int i=0;i<l;i++){
```

```
    p[i] = (int*)malloc(sizeof(int)*c);
```

```
}
```

```
for (i = 0; i < l; i++)
```

```
    for (j = 0; j < c; j++)
```

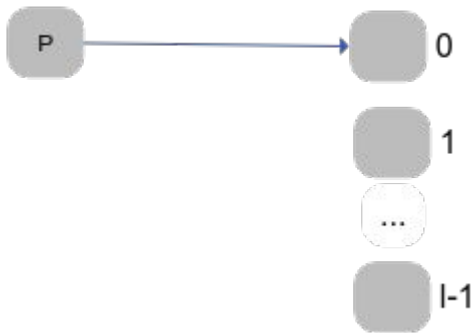
```
        p[i][j] = rand();
```

Exemplo: matrizes dinâmicas (Ilustração)

```
int **p=NULL;
```



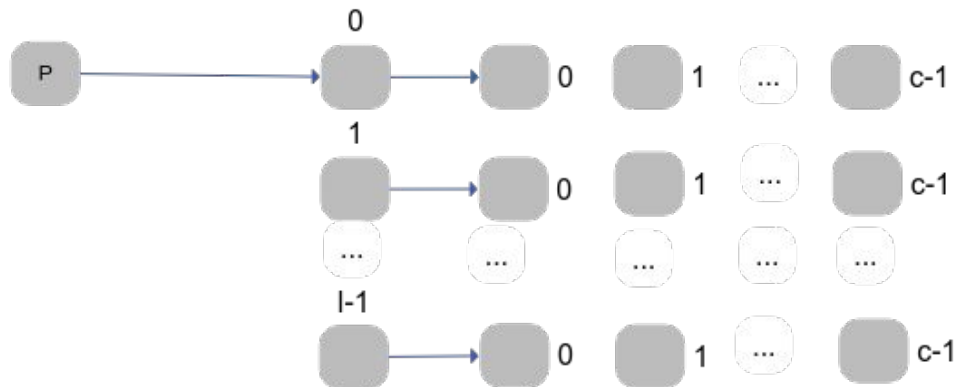
```
p = (int**)malloc(sizeof(int*)*l);
```



```
for(int i=0;i<l;i++){
```

```
    p[i] =(int*)malloc(sizeof(int)*c);
```

```
}
```

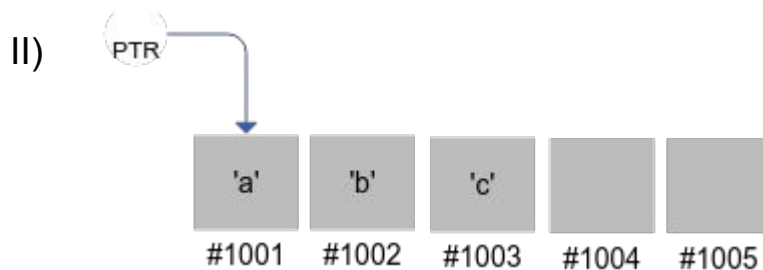
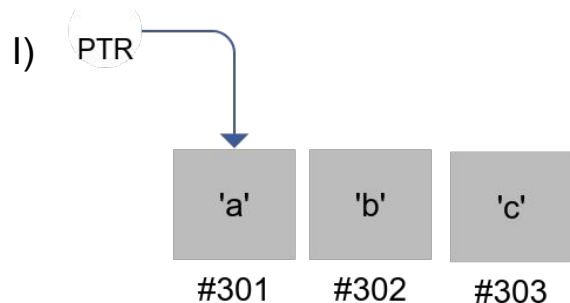


Exercício

1. Crie uma agenda em C para armazenar telefones de n pessoas (n entrado pelo usuário).
2. Crie uma função `void addTelefone(int *agenda, int numero);` que adiciona um número ao 1º espaço em branco da agenda
 - a. dica: utilize um contador passado por referência, para guardar o nº de registros

Realocação de memória

- O C permite dinamicamente alterar a alocação de memória previamente alocada
- Para isso, faz-se uso da função `void* realloc (void* ptr, unsigned size);`
 - altera o tamanho do bloco de memória apontado por `ptr`,
 - movendo o bloco para um novo local (junto com seu conteúdo)
 - o endereço desse local é retornado pela função
 - se a realocação falhar, **NULL** é retornado



Requisitando aumento de um vetor de inteiros

```
int *v, t_novo, t_inicial=2, t_extra=3;

//alocando t_inicial int
v = malloc(t_inicial*sizeof(int));

//preenchendo com valores aleatorios
for(i=0; i<t_inicial; i++)
    v[i]=rand()%100;

//mostrando o vetor inicial
for(i=0; i<t_inicial; i++)
    printf("%d ", v[i]);

//realocando p. (t_novo) int
t_novo = t_extra+t_inicial;
v = realloc(v, (t_novo)*sizeof(int));

//preenchendo o novo espaço do vetor
for(i=t_inicial; i<(t_novo); i++)
    v[i]=rand()%100;

//mostrando o vetor final
for(int i=0; i<(t_novo); i++)
    printf("%d ", v[i]);
```

Exercício

- A partir do exercício anterior, suponha que a agenda necessite suportar k telefones a mais que a sua capacidade atual. Utilizando a função *realloc()*, aumente o tamanho da agenda e adicione mais k telefones nela.