

INSTITUTO FEDERAL DE  
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA  
PARAÍBA  
Campus Campina Grande

Curso de Engenharia da Computação  
Programação e Estruturas de Dados

# Linguagem C, Apontadores e Revisão Prática

Copyright©2018  
Prof. César Rocha  
cesarocha@ifpb.edu.br /

# Objetivos

- *Explorar os conceitos fundamentais acerca do uso de apontadores (ponteiros) na linguagem C*
  - *sintaxe, principais operadores, erros comuns, aritmética de ponteiros, cadeias de caracteres, ponteiros em vetores, vetores de apontadores, acessando vetores com apontadores, indexação, diretrizes básicas para utilizá-los com segurança, etc.*
- *Mostrar vários exemplos de **códigos** que você **deve** testar e solidificar seus conhecimentos*
- *Este módulo é **breve** e aborda alguns assuntos já vistos em disciplinas anteriores*

- Em poucas palavras, um apontador é uma **variável** que guarda **o endereço** de uma área de memória
  - Em programação, uma **variável** é uma área da memória que “sabe como” guardar um valor específico
  - Tal endereço armazenado pelo ponteiro, geralmente, é a **posição em memória de uma outra variável**
- A memória possui endereços para que possamos referenciá-la adequadamente
  - Assim, toda variável declarada possui um endereço relacionado a esta, o qual indica a posição de memória onde se inicia o armazenamento dos bytes relativos ao valor (dados) desta variável

# Graficamente...

- Cada variável possui **seu próprio endereço** de onde foi alocada na memória ( inclusive o apontador! )
- O apontador, por sua vez, **ainda guarda o endereço da Variável 2**
- O apontador **não guarda o conteúdo da variável 2 (128,78)!**

- O **ponteiro** contém o **endereço** de uma variável que possui um valor armazenado

Memória RAM

Endereços

Variáveis



- A linguagem C faz uso maciço de apontadores:
  - pode-se gerar códigos mais **compactos** e **eficientes**
  - em alguns casos, é a única maneira de se realizar uma determinada operação computacional
  - utilizado para que **funções** modifiquem os valores de seus argumentos
  - bastante usados em trabalhos que exijam **alocação dinâmica de memória** (veremos mais adiante)
  - você já deve saber que boa parte da **biblioteca de C** possui dezenas de funções manipulam apontadores (passando e/ou retornando-os)

## Porém...

- Programar em C usando apontadores requer um pouco mais de disciplina e de cuidados
- É fácil criar apontadores que enderecem **posições de memória indevidas**, como as áreas de dados da aplicação ou até mesmo outros programas
- Em alguns casos mais críticos, a utilização de forma incorreta pode provocar uma **parada total do aplicativo ou até mesmo do sistema operacional**
- Somente a **prática** garante a experiência e o domínio total deste recurso

## Sintaxe utilizada em C

- Se uma variável irá conter um ponteiro, ela deve ser declarada como tal
- Uma vez que o apontador tenha sido **declarado e inicializado** com um endereço válido de um elemento, este último poderá ser acessado indiretamente através do apontador
- A **forma geral** para se declarar uma variável ponteiro é a seguinte:

```
<tipo> * <nome_da_variável> ;
```

## Sintaxe utilizada em C

```
int main (void) {  
    int *pntInt;  
    float *pntFloat, salarioMensal;  
    double *d = NULL; /* constante simbólica */  
    char *str = "Apontadores em C";  
}
```

- *O tipo base do ponteiro define o tipo base de variáveis para o qual o ponteiro pode apontar*
- *Sempre procure, após declarar o ponteiro, **inicializá-lo**.*
  - *Isso é muito importante para evitar resultados indesejados!!!*



## Operadores & e \*

- Conforme já foi dito, uma variável pode ser acessada indiretamente através de um ponteiro
  - Para tanto, há **2 operadores unários** fundamentais utilizados na manipulação de pontadores: \* e &.
- O operador unário \* acessa o conteúdo de uma dada posição de memória cujo endereço está armazenado em um apontador
- O operador unário & devolve o endereço de memória inicial de uma variável
  - **Dica:** o operador & pode ser aplicado apenas à variáveis

# Operadores & e \*

- **&** (lê-se: *o endereço de memória de...*) como em:

```
int x = 10;
```

```
int *px; // apontador para um inteiro
```

```
px = &x; // coloca em px o endereço de memória da variável x
```

- *Não são permitidas* construções do tipo:

```
int x = 10;
```

```
int *px; // apontador para um inteiro
```

```
px = &(x + 5); // expressões
```

```
px = &10; // não é possível obter endereços de constantes
```

# Operadores & e \*

- \* (lê-se: *o conteúdo de...*) como em:

```
int y = 10, z;
```

```
int *py; /* apontador para a variavel y */
```

```
py = &y; /* coloca em py o endereço y */
```

```
z = *py; /* Atribui a z o valor (conteúdo) da variável y */
```

```
float f = 10.5;
```

```
float *pf; /* apontador para a variavel f */
```

```
pf = &f; /* coloca em pf o endereço f */
```

```
*pf = 300; /* O que esta instrução faz? */
```

```
pf = 300; /* E esta instrução? Está errada? */
```

## Evite erros!

- Observe, no slide anterior, que é muito fácil apontar para áreas de memória indevidas!
- Por isso, **lembre-se**: quando você declara um ponteiro no seu código, ele **não aponta para lugar nenhum!** Você deve “direcionar” o ponteiro para um endereço válido. Ainda que para **NULL**

```
int *pntInt;
```

```
int i = 100;
```

```
pntInt = NULL; /* está correto */
```

```
pntInt = &i; /* está correto */
```

# Qual a saída do programa abaixo?

```
#include <stdio.h>
void main(void) {
    int x = 50, *p1, *p2;
    p1 = &x; // p1 recebe o endereço de x ou aponta para x
    p2 = p1; // p2 recebe endereço de p1 ou &x
    printf( "%p\n", p2); // &x
    printf( "%p\n", p1); // &x
    printf( "%p\n", &x); // &x
    printf( "%p\n", x); // errado, pois %p
    printf( "%d\n", p2); // errado pois %d
    printf( "%d\n", *p2); // imprime 50
    printf( "%p\n", &p2); // endereço onde p2 foi alocado
    printf( "%d", x); // imprime 50
}
```

# Como cavar a própria cova!

```
int *p1;  
p1 = 5000;
```

*O ponteiro não foi  
inicializado! E se o endereço  
5000 estiver alocado ao SO?*

```
int *p2;  
*p2 = 5000;
```

*Erro gravíssimo! Reze para  
não travar a máquina...*



```
int *p3;  
float a;  
p3 = &a;
```

*Incompatibilidade de tipos  
(talvez apenas um WARN)*

```
int *p4;  
float b, c;  
p4 = &b;  
c = *p4;
```

*Apenas 2 bytes são  
transferidos para c, e não os  
4 bytes do tipo float em b!*

# Aritmética de ponteiros

- Podemos realizar duas operações aritméticas básicas com ponteiros: **adição** e **subtração**
- Existem algumas regras que governam isso:
  - **R1:** se um ponteiro aponta para um objeto, a aritmética será feita levando-se em conta o **tamanho do objeto**
  - **R2:** cada vez que um ponteiro é incrementado (ou decrementado) o “salto” será feito **de acordo com o tipo base do ponteiro**
  - **R3:** não se pode adicionar ou subtrair constantes **float** ou **double** a ponteiros
  - **R4:** é claro, **não se pode dividir ou multiplicar** ponteiros

# Aritmética de ponteiros

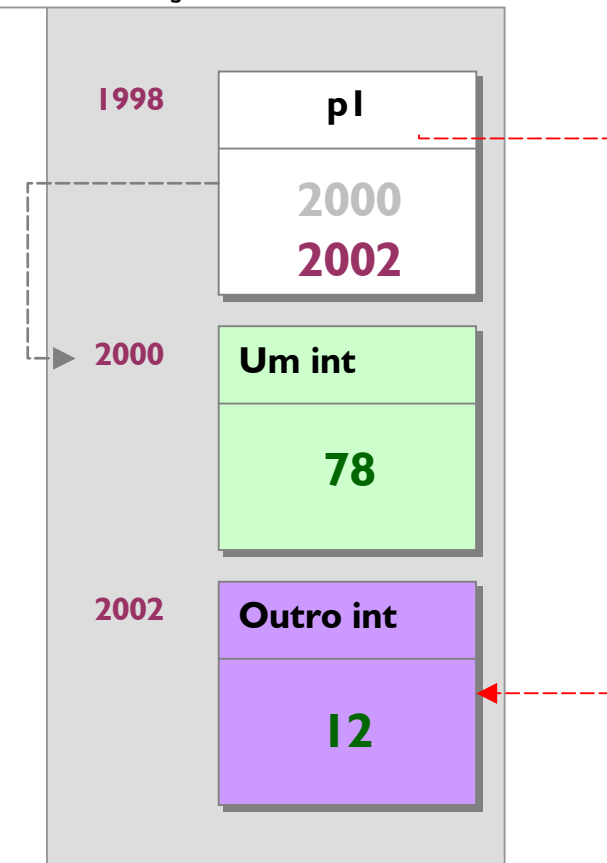
## ■ Exemplo:

- Se um ponteiro *pl* aponta para um inteiro localizado no endereço 2000 (assuma inteiro ocupando 2 bytes). Qual o resultado final após a expressão:  
`pl++;` ?
- **Conclusão:** Na figura ao lado, quando *pl* foi incrementado, ele “saltou” 2 bytes para o próximo inteiro

Memória RAM

Endereços

Variáveis





## Aritmética de ponteiros

- *Você não está limitado apenas a operações de incremento e decremento*
- *O comando **pl += 8** é válido. O ponteiro irá saltar para o oitavo elemento a partir do elemento em que ele estiver, atualmente, apontando.*
- *Mas, **pl += 5.86;** não o é. Isso por causa da regra **R3** vista anteriormente.*
- **Dica:** *utilize **parênteses** para evitar enganos:*
  - **\*pt + 3 == 10;** *é bem diferente de...*
  - **\*(pt + 3) == 10;**

# Apontadores e cadeias de caracteres

- Você já deve saber que constantes do tipo cadeia são dados na forma:
  - “**Eu sou uma cadeia de caracteres**”
  - Esta cadeia de caracteres é, na verdade, **um vetor de caracteres**. O tamanho armazenado na memória é o tamanho da cadeia + 1, para conter o **‘\0’** no final.
- Poderemos ter num programa a seguinte construção

```
char *mensagem = "O livro é um alimento";  
mensagem = "vetor anterior foi perdido!";
```

  - Um ponteiro para caracteres, por definição, **aponta sempre para o primeiro caractere da cadeia!**
  - Na linguagem C **não existem** operadores para manipular cadeia de caracteres, isto é feito **através de funções**.

# Apontadores e cadeias de caracteres

## ■ **Cuidado:**

```
char *nome = "José";  
char *nomeCompleto = null;  
if (*nome == "José"){  
    *nomeCompleto = *nome + "da Silva";  
} Código errado!
```

*Consulte a API C sobre as funções **strcmp** e **strcat**. Existem ainda outras funções: **strcpy**, **strlen**...*

```
char *nome = "José";  
char *nomeCompleto = NULL;  
if (strcmp( nome, "José") == 0 ){  
    nomeCompleto =  
        strcat( nome, "da Silva" );  
} Correto!
```

# Apontadores e vetores

- *Em C, qualquer operação que possa ser realizada com índices de um vetor também pode ser realizada com apontadores. Veja o código abaixo:*

```
#include <stdio.h>
void main(void) {
    char str[] = "Cesar"; // char str[] = {'C', 'e', 's', 'a', 'r', '\0' };
    char *pt;
    pt = str; // pt irá apontar para &str[0]
    str[2] = 'A'; /* Atribui A ao terceiro elemento de str */
    putchar( *(pt + 2) ); /* ou pt[2] */
}
```

# Apontadores e vetores

- Muitas vezes, um ponteiro e um vetor se confundem no código devido a uma regra básica:
  - O nome de um vetor é sempre interpretado como o endereço do primeiro elemento deste.
- Por exemplo:

```
int a [3] = {1,2,3};  
int *p, *q;  
p = a; // a mesma coisa que &a[0];  
q = p;
```
- Após este código, **a**, **p** e **q** referem-se todos ao mesmo endereço

## Apontadores e vetores

- Assim, de forma semelhante, ponteiros também podem ser indexados como arrays:  
 $a[1] = 7$  ou  $p[1] = 7$  ou até  $*(p+1) = 7$
- **Mas lembre-se:** arrays e ponteiros *não são a mesma coisa!*
  - Um ponteiro ocupa uma palavra de memória enquanto que, normalmente, um array ocupa várias posições
  - Um vetor *não pode* mudar de endereço!
  - Qual o significado de  $a = \&salario;$  ?

# Apontadores e vetores

**Portanto:**

<code>*ptInt</code>	<code>==</code>	<code>p[0]</code>	<code>==</code>	<code>vetorInt[0]</code>
<code>*(ptInt + 1)</code>	<code>==</code>	<code>p[1]</code>	<code>==</code>	<code>vetorInt[1]</code>
<code>*(ptInt + 2)</code>	<code>==</code>	<code>p[2]</code>	<code>==</code>	<code>vetorInt[2]</code>
<code>*(ptInt + n)</code>	<code>==</code>	<code>p[n]</code>	<code>==</code>	<code>vetorInt[n]</code>

```
#include <stdio.h>
void main(void) {
    char vetor[] = "Cesar"; // char vetor[] = {'C', 'e', 's', 'a', 'r', '\0'};
    char *pv = vetor; // ou char *pv = &vetor[0];
    pv[ 2 ] = 'Z'; // ou *( pv+2 ) = 'Z'; ou vetor[2] = 'Z';
    *( pv+1 ) = 'R'; // ou p[ 1 ] = 'R'; ou vetor[1] = 'R';
    printf("%c", *( pv - 2 ) );
    printf("%c", pv[ 1 ] );
}
```

# Varrendo vetores com ponteiros

- Pode-se varrer os elementos de um vetor via **apontadores** e também via **indexação**

```
char v1[] = "Maria";
```

```
char v2[] = "Zezinho"; // char v2[] = {'Z', 'e', ..., '\0'};
```

```
char *p;
```

```
p = v1; // ou p = &v1[0];
```

```
while ( *p )
```

```
    putchar( *p++ );
```

```
p = v2; // ou p = &v2[0];
```

```
for ( register i = 0; v2[ i ]; ++i )
```

```
    putchar( *( p + i ) ); // ou p[i] ou v2[i] ou *(vetor + i)
```



# Apontadores vs. Indexação

- E então, devo **apontar ou indexar?**
  - A literatura diz que o uso de apontadores é mais adequado quando o vetor é **acessado contiguamente**, em ordem ascendente ou descendente
    - a codificação escrita com apontadores é mais rápida, uma vez que não se faz necessário o cálculo da posição de memória a partir do índice normal
    - Porém, o programador deve lembrar-se de reinicializar o ponteiro, após utilizá-lo
    - No geral, se você se sente confiante em utilizar ponteiros, faça-o.
  - O uso de índices é, geralmente, mais fácil de ser compreendido por programadores iniciantes

# Vetores de apontadores

- Ponteiros podem ser organizados em arrays como qualquer outro tipo de dado:
- Por exemplo:

```
int *inteiros[10];  
int i = 20, j = 40;  
inteiros[0] = &i;  
inteiros[1] = &j;  
printf("i = %d j= %d", *inteiros[0], *inteiros[1]);
```

- **Lembre-se** que `*inteiros[]` não é um ponteiro para inteiros, e sim um **array de ponteiros para inteiros**

# Vetores de apontadores

- Vetores de ponteiros são, geralmente, *usados em ponteiros para cadeias de caracteres*
  - Um exemplo bastante interessante e útil em programação consiste em construir uma função capaz de retornar mensagens a partir de um código:

```
void erros( int cod ) {  
    char *mensagemErro[] = {  
        "Impressora desconectada\n",  
        "Rede fora do ar\n",  
        "Arquivo não encontrado\n"  
    } ;  
    printf( "%s", mensagemErro[ cod ] );  
}
```

# Passagem de parâmetros para funções

- Em C, as funções são blocos de construção essenciais no desenvolvimento de um sistema
  - Consiste no local onde as atividades do programa são codificadas e executadas
  - C é uma **linguagem imperativa** e não é preciso dizer que a programação imperativa dá ênfase **às funções** (ou verbos) na hora de modelar um problema do mundo real
- Podemos passar parâmetros para as funções de duas formas:
  - **Passagem por valor**: este método cria uma cópia do valor de um argumento e o passa como parâmetro da função
  - **Passagem por referência**: este método não passa uma cópia, e sim o endereço do argumento para o parâmetro

# Passagem de parâmetros para funções

- Na passagem de **parâmetros por valor**, modificações feitas nos parâmetros dentro da função **não serão** refletidas fora desta
  - Quando a função termina, a cópia também desaparece.

```
int x = 5, int y = 2; // declaração de variáveis
```

```
trocar( x, y ); // chamada de função
```

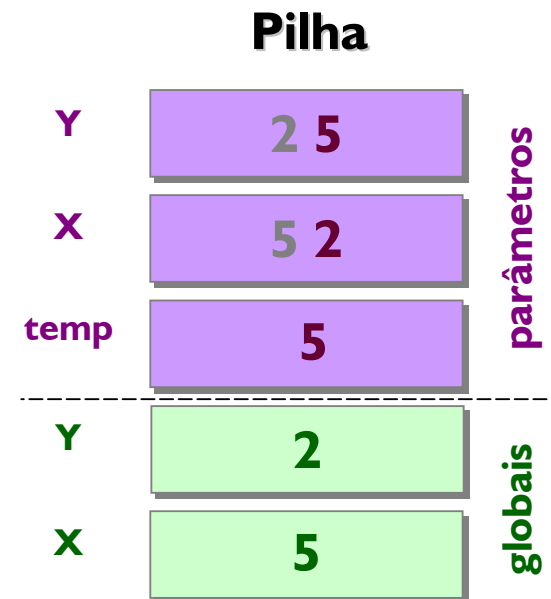
```
void trocar( int x, int y ) {
```

```
    int temp = x;
```

```
    x = y;
```

```
    y = temp;
```

```
}
```



# Passagem de parâmetros para funções

- Na passagem de **parâmetros por referência**, (usando ponteiros) faz com que o endereço do argumento seja passado como parâmetro
  - Alterações **serão** refletidas fora da função

```
int x = 5, int y = 2; // declaração de variáveis
```

```
trocar( &x, &y ); // chamada função
```

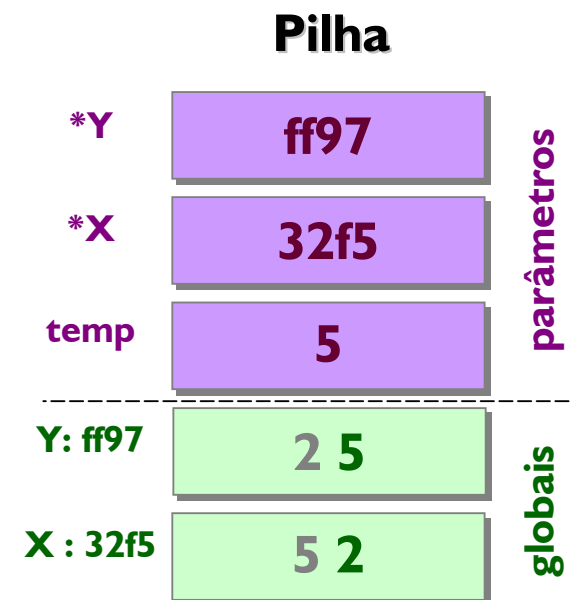
```
void trocar( int *x, int *y ) {
```

```
    int temp = *x;
```

```
    *x = *y;
```

```
    *y = temp;
```

```
}
```



## ■ Qual(is) o(s) erro(s)?

```
char *cliente; char endereco[40];  
int idade = 50, *p1, *p2;  
  
*cliente = "Lojas Americanas";  
endereco = "Av. Sol, 555";  
*p1 = 50;  
p2 = idade;  
if ( p1 < p2 )  
    printf( "p1 em área de memória baixa. \n" );  
exit(0);
```

## ■ Qual(is) o(s) erro(s)?

char \*cliente; char endereco;

int idade = 50, \*p1, \*p2;

\*cliente = "Lojas Americanas";

endereco = "Av. Sol, 555";

\*p1 = 50;

p2 = idade;

if ( p1 < p2 )

printf( "p1 em área de memória baixa. \n" );

exit(0);

1) Ponteiro não inicializado!

2) Inicialização array errada!

3) Área de memória desconhecida irá receber 50 em p1

4) Endereço 50 em p2 pode estar alocado a outro programa ou dados ou SO. Código perigoso!

5) O SO decide onde colocar as variáveis. Printf pode não ser executado amanhã!

Preste atenção nos pontos críticos!




## ■ Qual(is) o(s) erro(s)? (Versão 2)

```
char *pl, s[81];  
pl = s;  
do {  
    scanf( " %80[ ^\n]", s ); //não captura brancos no início,  
    limite de 80, permitindo brancos no meio da cadeia e todos  
    caracteres menos enter  
    while ( *pl ) printf ( "%c", *pl++ ) ;  
} while ( strcmp( s, "fim" ) );  
exit(0);
```

## ■ Qual(is) o(s) erro(s)?

```
char *pl, s[81];  
1 pl = s;  
do {  
    scanf( " %80[^\n]", s ); //não captura brancos, limite de 80 ate que o usuário enter  
    // imprime cada caractere  
    while ( *pl ) printf ( "%c", *pl++ ) ;  
} while ( strcmp( s, "fim" ) );  
exit(0);
```



1) Na primeira iteração do laço, *pl* aponta para *S*. O *while* imprime a string corretamente, mas...

2) e no final da impressão e início da segunda interação, pra onde *pl* está apontando? Por onde a impressão vai começar neste momento?

O que deve ser feito para corrigir o programa?

## *Para um bom aproveitamento:*

- *Instale a IDE de sua preferência e configure logo seu ambiente de trabalho*
- *Codifique os exemplos mostrados nestes slides e verifique pontos de dúvidas*
- *Resolva os exercícios da **lista de apontadores***
- *Procure o professor ou monitor da disciplina e questione conceitos, listas, etc.*
- *Não deixe para codificar tudo e acumular assunto para a primeira avaliação.*
  - *Este é apenas um dos assuntos abordados na prova!*