Uso de Ponteiros na linguagem C

Prof. Marcelo Costa, MSc
marcelo.costa@uva.edu.br

Agenda

- Como funcionam os ponteiros
- Declarando e utilizando ponteiros
- Ponteiros e Vetores

Introdução aos ponteiros

- Para ser um bom programador em C é fundamental que se tenha um bom domínio deles.
- Ponteiros são utilizados mesmo nos exemplos mais simples na linguagem C

Como Funcionam os Ponteiros

- Os ints guardam inteiros. Os floats guardam números de ponto flutuante. Os chars guardam caracteres.
- Ponteiros guardam endereços de memória.
- Quando você anota o endereço de um colega você está criando um ponteiro. O ponteiro é este seu pedaço de papel.
- O C funciona assim. Você anota o endereço de algo numa variável ponteiro para depois usar.

Tipo de ponteiros

- No C quando declaramos ponteiros nós informamos ao compilador para que tipo de variável vamos apontá-lo.
- Um ponteiro int aponta para um inteiro, isto é, guarda o endereço de um inteiro.
- Um ponteiro char aponta para um tipo char.

Declaração de ponteiros

- Forma geral:

tipo_do_ponteiro *nome_da_variável;

- É o asterisco (*) que faz o compilador saber que aquela variável não vai guardar um valor mas sim um endereço para aquele tipo especificado.
- int *pt; - ponteiro para um inteiro
- char *temp,*pt2; - dois ponteiros para caractere
- Isto significa que eles apontam para um lugar indefinido.
- Usar o ponteiro nestas circunstâncias pode levar a um travamento do micro, ou a algo pior

Inicialização de ponteiros

- O ponteiro deve ser inicializado (apontado para algum lugar conhecido) antes de ser usado!
- Deixamos o compilador indicar o endereço de memória do ponteiro.
- Exemplo:

```
int count=10;
int *pt;
pt=&count;
```

- Criamos um inteiro count com o valor 10 e um apontador para um inteiro pt.
- A expressão &count nos dá o endereço de count, o qual armazenamos em pt.
- Em outras palavras, o ponteiro pt guarda o endereço de count.

Operadores básicos * e &

- O operador * indica o conteúdo do ponteiro, ou seja, aponta para o **valor** da variável referenciada pelo ponteiro:

```
int count=10;
int *pt;
pt=&count;
```

*pt = 10

- O operador & indica o endereço do ponteiro, ou seja, aponta para o **valor do endereço de memória** referenciada pelo ponteiro:

&pt = HAFFF

Exemplos de ponteiros

```
#include <stdio.h>

int main ()

{
    int num,valor;
    int *p;
    num=55;
    p=&num; /* Pega o endereco de num */
    valor=*p; /* Valor e igualado a num de uma maneira indireta */
    printf ("\n\n%d\n",valor);
    printf ("Endereco para onde o ponteiro aponta: %p\n",p);
    printf ("Valor da variavel apontada: %d\n",*p);
    return(0);

}
```

Exemplos de ponteiros

```
#include <stdio.h>

int main ()

{
    int num,*p;
    num=55;
    p=&num; /* Pega o endereco de num */
    printf ("\nValor inicial: %d\n",num);
    *p=100; /* Muda o valor de num de uma maneira indireta */
    printf ("\nValor final: %d\n",num);
    return(0);

}
```

Operações aritméticas de ponteiros – igualdade de ponteiros

- Se temos dois ponteiros p1 e p2 podemos igualá-los fazendo p1=p2.
- Estamos fazendo com que p1 aponte para o mesmo lugar que p2. (mesmo endereço de memória)
- A variável apontada por p1 tenha o mesmo conteúdo da variável apontada por p2 devemos fazer *p1=*p2.
- Novamente o & é utilizado para o ponteiro apontar para endereços de memória das variáveis.

```
int num,*p;
num=55;
p=&num
```

- P tem o mesmo endereço de memória de num

Incremento e decremento de ponteiros

- Quando incrementamos um ponteiro ele passa a apontar para o próximo valor do mesmo tipo para o qual o ponteiro aponta.
- Isto é, se temos um ponteiro para um inteiro e o incrementamos ele passa a apontar para o próximo inteiro.
- Esta é mais uma razão pela qual o compilador precisa saber o tipo de um ponteiro: se você incrementa um ponteiro char* ele anda 1 byte na memória e se você incrementa um ponteiro double* ele anda 8 bytes na memória.
- O decremento funciona de forma semelhante.

Exemplo de operações

- Supondo que p é um ponteiro, as operações são escritas como:

  p++; (caminha para próxima posição do ponteiro)

  p--; (caminha para a posição anterior do ponteiro)

- Incremento do conteúdo do ponteiro:

  (*p)++;

- Soma e subtração de inteiros com ponteiros.

  *p=*p+15; ou *p+=15; (adiciona valor ao conteúdo do ponteiro)

- Se quiser andar 15 posições no ponteiro:

  p = p + 15 ou p+=15

- E se você quiser usar o conteúdo do ponteiro 15 posições adiante:

  *(p+15);

Comparação de ponteiros

- Em primeiro lugar, podemos saber se dois ponteiros são iguais ou diferentes == e !=. OU seja, se apontam para o mesmo endereço de memória
- Operações do tipo >, <, >= e <= estamos comparando qual ponteiro aponta para uma posição mais alta na memória. Uma comparação entre ponteiros pode nos dizer qual dos dois está "mais adiante" na memória.
- **Importante:** Há entretanto operações que você não pode efetuar num ponteiro. Você não pode dividir ou multiplicar ponteiros, adicionar dois ponteiros, adicionar ou subtrair floats ou doubles de ponteiros.

Exercícios

- Explique a diferença entre
- p++; (*p)++; *(p++);
- O que quer dizer *(p+10);?
- Explique o que você entendeu da comparação entre ponteiros?
- Gerar um programa para testar essas diferenças.

Exercício

- Qual o valor de y no final do programa? Tente primeiro descobrir e depois verifique no computador o resultado. A seguir, escreva um /* comentário */ em cada comando de atribuição explicando o que ele faz e o valor da variável à esquerda do '=' após sua execução.

```
int main()
{
    int y, *p, x;
    y = 0;
    p = &y;
    x = *p;
    x = 4;
    (*p)++;
    x--;
    (*p) += x;
    printf ("y = %d\n", y);
    return(0);
}
```

Ponteiros e Vetores

- Quando você declara uma matriz da seguinte forma:

  tipo_da_variável nome_da_variável [tam1][tam2] ... [tamN];

- O compilador C calcula o tamanho, em bytes, necessário para armazenar esta matriz. Este tamanho é:

  tam1 x tam2 x tam3 x ... x tamN x tamanho_do_tipo

- O nome da variável que você declarou é na verdade um ponteiro para o tipo da variável da matriz. Este conceito é fundamental. Eis porque: Tendo alocado na memória o espaço para a matriz, ele toma o nome da variável (que é um ponteiro) e aponta para o primeiro elemento da matriz.

Ponteiro e Vetores

- Então como é que podemos usar a seguinte notação?

nome_da_variável[índice]

- Isto pode ser facilmente explicado desde que você entenda que a notação acima é absolutamente equivalente a se fazer:

*(nome_da_variável+índice)

Índices negativos de ponteiros

- Apesar de, na maioria dos casos, não fazer muito sentido,
- Poderíamos ter índices negativos. Estaríamos pegando posições de memória antes do vetor.
- Isto explica porque o C não verifica a validade dos índices. Ele não sabe o tamanho do vetor. Ele apenas aloca a memória, ajusta o ponteiro do nome do vetor para o início do mesmo e, quando você usa os índices, encontra os elementos requisitados.

Usando ponteiros para acessar matriz

- Quando temos que varrer todos os elementos de uma matriz de uma forma sequencial, podemos usar um ponteiro, o qual vamos incrementando.

```
int main ()
{
    float matrx [50][50];
    int i,j;
    for (i=0;i<50;i++)
      for (j=0;j<50;j++)
         matrx[i][j]=0.0;
    return(0);
}
```

Utilizando ponteiros

```
int main ()
{
    float matrx [50][50];
    float *p;
    int count;
    p=matrx[0];
    for (count=0;count<2500;count++)
    {
        *p=0.0;
        p++;
    }
    return(0);
}
```

- No primeiro programa, cada vez que se faz matrx[i][j] o programa tem que calcular o deslocamento para dar ao ponteiro. Ou seja, o programa tem que calcular 2500 deslocamentos. No segundo programa o único cálculo que deve ser feito é o de um incremento de ponteiro. Fazer 2500 incrementos em um ponteiro é muito mais rápido que calcular 2500 deslocamentos completos

Ponteiros como vetores

- O nome de um vetor é um ponteiro constante. Sabemos também que podemos indexar o nome de um vetor. Como consequência podemos também indexar um ponteiro qualquer.

```
#include <stdio.h>

int main ()

{
    int matrx [10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    int *p;
    p=matrx;
    printf ("O terceiro elemento do vetor e: %d",p[2]);
    return(0);

}
```

- Podemos ver que p[2] equivale a *(p+2).

```
#include <stdio.h>

void StrCpy (char *destino,char *origem)

{
	while (*origem)
	{
		*destino=*origem;
		origem++;
		destino++;
	}
	*destino='\0';

}

int main ()

{
	char str1[100],str2[100],str3[100];
	printf ("Entre com uma string: ");
	gets (str1);
	StrCpy (str2,str1);
	StrCpy (str3,"Voce digitou a string ");
	printf ("\n\n%s%s",str3,str2);
	return(0);
}
```

Comentando o exemplo

- Na verdade é assim que funções como gets() e strcpy() funcionam. Passando o ponteiro você possibilita à função alterar o conteúdo das strings. Você já estava passando os ponteiros e não sabia.
- No comando while (*origem) estamos usando o fato de que a string termina com'\0' como critério de parada.
- Quando fazemos origem++ e destino++ o valor do ponteiro-base da string é alterado.
- No C, são passados para as funções cópias dos argumentos.
- Desta maneira, quando alteramos o ponteiro origem na função StrCpy() o ponteiro str2 permanece inalterado na função main().

Endereços de elementos de vetores

- A notação a seguir é válida:

&nome_da_variável[índice]

- retorna o endereço do ponto do vetor indexado por índice. Isto seria equivalente a nome_da_variável + indice.
- É interessante notar que, como consequência, o ponteiro nome_da_variável tem o endereço &nome_da_variável[0], que indica onde na memória está guardado o valor do primeiro elemento do vetor.

int *pont = &vetor[0]

Vetores de ponteiros

- Podemos construir vetores de ponteiros como declaramos vetores de qualquer outro tipo. Uma declaração de um vetor de ponteiros inteiros poderia ser:

```
int *pmatrx [10];
```

- No caso acima, pmatrx é um vetor que armazena 10 ponteiros para inteiros.

Exercícios

- Fizemos a função StrCpy(). Faça uma função StrLen() e StrCat() que funcionem como as funções strlen() e strcat() de string.h respectivamente

Ponteiros para Ponteiros

- Um ponteiro para um ponteiro é como se você anotasse o endereço de um papel que tem o endereço da casa do seu amigo. Podemos declarar um ponteiro para um ponteiro com a seguinte notação:

```
tipo_da_variável **nome_da_variável;
```

- Algumas considerações: **nome_da_variável é o conteúdo final da variável apontada; *nome_da_variável é o conteúdo do ponteiro intermediário.
- Para fazer isto basta aumentar o número de asteriscos na declaracão.

Exemplo de ponteiros para ponteiro

```
#include <stdio.h>

int main()

{
    float fpi = 3.1415, *pf, **ppf;
    pf = &fpi; /* pf armazena o endereco de fpi */
    ppf = &pf; /* ppf armazena o endereco de pf */
    printf("%f", **ppf); /* Imprime o valor de fpi */
    printf("%f", *pf); /* Tambem imprime o valor de fpi */
    return(0);

}
```

Exercícios

- Verifique o programa abaixo. Encontre o seu erro e corrija-o para que escreva o numero 10 na tela.

```
#include <stdio.h>

int main()

{
    int x, *p, **q;
    p = &x;
    q = &p;
    x = 10;
    printf("\n%d\n", &q);
    return(0);

}
```

Cuidados ao usar ponteiros

- O principal cuidado ao se usar um ponteiro deve ser: saiba sempre para onde o ponteiro está apontando. Isto inclui: nunca use um ponteiro que não foi inicializado.

```
int main () /* Errado - Nao Execute */
{
    int x,*p;
    x=13;
    *p=x;
    return(0);
}
```

- O ponteiro p pode estar apontando para qualquer lugar. Você estará gravando o número 13 em um lugar desconhecido. Com um número apenas, você provavelmente não vai ver nenhum defeito. Agora, se você começar a gravar números em posições aleatórias no seu computador, não vai demorar muito para travar o micro

Exercícios

- Escreva um programa que declare uma matriz 100x100 de inteiros. Você deve inicializar a matriz com zeros usando ponteiros para endereçar seus elementos. Preencha depois a matriz com os números de 1 a 10000, também usando ponteiros.

Exercícios

- Crie um programa em C que peça ao usuário três números inteiros e armazene em três variáveis inteiras através do uso de um ponteiro. Após o usuário inserir cada número, mostre o número exibido, porém mostre através do ponteiro.

```
int main(void)

{

    int int1, int2, int3;

    int *ptr_int = &int1;

     printf("Inteiro 1: ");

    scanf("%d", ptr_int);

    printf("Numero inserido: %d\n", *ptr_int);

     ptr_int = &int2;

    printf("Inteiro 2: ");

    scanf("%d", ptr_int);

    printf("Numero inserido: %d\n", *ptr_int);

    ptr_int = &int3;

    printf("Inteiro 3: ");

    scanf("%d", ptr_int);

    printf("Numero inserido: %d\n", *ptr_int);

    return 0;

}
```

Passagem por referência com ponteiros

- Para passarmos uma variável para uma função e fazer com que ela seja alterada, precisamos passar a referência dessa variável, em vez de seu valor.
- Quando passamos um valor, a função copia esse valor e trabalhar somente em cima da cópia dessa variável, e não na variável em si. Por isso nossas variáveis nunca eram alteradas quando passadas para funções.
- Por referência entenda endereço, um local. No caso, em vez de passar o valor da variável, **na passagem por referência vamos passar o endereço da variável** para a função.

Como fazer?

- Basta colocar o operador & antes do argumento que vamos enviar, e colocar um asterisco * no parâmetro da função, no seu cabeçalho de declaração, para dizer a função que ela deve esperar um endereço de memória, e não um valor.
- Para que uma função altere o valor de uma variável, é necessário que essa função atue no endereço de memória, e não o valor.
- Para isso, temos que passar o endereço da variável para a função (usando &), e a função tem que ser declarada de modo a esperar um ponteiro (usando *).

Exemplo de código: Passagem por referência em C - Crie um programa que recebe um inteiro e dobra seu valor

```
void dobra(int *num)
{
    (*num) = (*num) * 2;
}
int main(void)
{
    int num;
    printf("Insira um numero: ");
    scanf("%d", &num);
    dobra(&num);
    printf("O dobro dele eh: %d\n", num);
    return 0;
}
```

Trocar o valor de dois números em C

- Declaramos uma função que irá receber dois ENDEREÇOS de memória, ou seja, irá receber dois ponteiros. Esses ponteiros têm os locais onde as variáveis ***x*** e ***y*** foram armazenadas.
- Para alterar esses valores, vamos trabalhar com ***(*x)*** e ***(*y)***.

```
vvoid troca(int *x, int *y)
{
    int tmp;
    tmp = *x;
    *x = *y;
    *y = tmp;
}
int main(void)
{    int x, y;
    printf("Insira o numero x: ");
    scanf("%d", &x);
    printf("Insira o numero y: ");
    scanf("%d", &y);
    troca(&x, &y);
    printf("Agora x=%d e y=%d\n", x, y);
    return 0; }
```

Comentários do exemplo

- Para fazer duas variáveis trocarem de valores entre si, iremos precisar de uma variável temporária.
- Primeiro guardamos o valor de ***x*** na variável temporária (vamos precisar depois). Em seguida, fazemos ***x*** receber o valor de ***y***.
- Agora é ***y*** que tem que receber o valor de ***x***. Mas x mudou de valor.
- Precisamos pegar o antigo valor de ***x***. Foi por isso que guardamos esse antigo valor de ***x*** na variável temporária. Logo, agora é só fazer com que ***y*** pegue o valor da variável temporária, e teremos os valores invertidos.

Utilizando Vetor como parâmetros

- Deve ser sempre passado como ponteiro
- O C recebe o endereço do vetor para utilizar dentro da função
- O valor pode ser alterado dentro da função pois é passador como referência

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

//protótipo da função media
float media (int n, float *vnotas);

int main (void)
{
  float vnotas[10];
  float media_notas;
  int i;

  /* leitura das notas */
  for (i = 0; i < 10; i++)
  {
    printf("Digite os valores das notas: ");
    scanf("%f", &vnotas[i]);
  }

  //chamada da função
  media_notas = media(10,vnotas);

  printf ( "\nMedia = %.1f \n", media_notas );

  system("pause");
  return 0;
}
```

```c
/* Função para cálculo da média
   Parâmetros:
      Recebe a quantidade de elementos n
      Recebe o endereço inicial do vetor notas em *vnotas
   Retorno:
     Retorna a media na variavel m
*/
float media (int n, float *vnotas)
{
  int i;
  float m = 0, soma = 0;

  //fazendo a somatória das notas
  for (i = 0; i < n; i++)
    soma = soma + vnotas[i];

  //dividindo pela quantidade de elementos n
  m = soma / n;

  //retornando a média
  return m;
}
```

Retornando Vetor em uma função

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

char* vetorS(){
   char valor[10]="Hello";
   return valor;
}

int main(){
   char vetor[10];
    strcpy(vetor,vetorS());
    printf(vetor);
   system(“PAUSE");
}
```

Exercicios

- Fazer um programa que leia uma string e criar uma função que receba uma string e retorne um vetor apenas com as vogais da frase.
- Dicas:
  - Criar um array apenas para alocar espaço para o array
  - Utilizar apenas ponteiros dentro da função
  - Utilizar um array para armazenar as vogais e retornar o ponteiro para o array

Função sizeof()

- O C vê sua memória RAM como um vetor enorme de bytes, que vão desde o número 0 até o tamanho dela (geralmente alguns Gb).
- Sempre que declaramos uma variável em C, estamos guardando, selecionando ou alocando um espaço de bytes desses, e dependendo do tipo de variável, o tanto de memória reservada varia.
- Podemos descobrir quantos bytes certa variável ocupa através da função *sizeof()*. Essa função recebe uma variável como argumento, ou as palavras reservadas que representam as variáveis: char, int, float etc

Exemplo sizeof()

```
int main(void)
{
    char caractere;
    int inteiro;
    float Float;
    double Double;
    printf("Caractere: %d bytes \t em %d\n", sizeof(caractere), &caractere);
    printf("Inteiro: %d bytes \t em %d\n", sizeof(inteiro), &inteiro);
    printf("Float: %d bytes \t em %d\n", sizeof(Float), &Float);
    printf("Double: %d bytes \t em %d\n", sizeof(Double), &Double);
    return 0;
}
```

Endereços de ponteiros

- O mesmo ocorre para um ponteiro. Sabemos que os ponteiros, ou apontadores, armazenam o endereço de **apenas um** bloco de memória.
- Quando um ponteiro aponta para uma variável que ocupa vários bytes, adivinha pra qual desses bytes o ponteiro realmente aponta? Ou seja, o endereço que o tipo ponteiro armazena, guarda o endereço de qual byte?
- **Do primeiro. Sempre do primeiro.**
- E como ele sabe o endereço dos outros? Os outros estão em posições vizinhas!

Função malloc

- Utilizada para alocar um espaço de memória predefinido na linguagem C
- Retorna um endereço para a primeira posição de memória da área alocada
- Deve ser atribuída SEMPRE a um ponteiro
- Sintaxe:

```
void *malloc(size_t numero_de_bytes);
```

Exemplo: char * a = malloc(500);

- Utilizar o sizeof para alocar um espaço de acordo com o tamanho do tipo de dados a ser armazenado;

Exemplos

```
char *nome = (char *) malloc(21*sizeof(char));

Int *pares = (int *) malloc(100 *sizeof(int));
```

Comando free

- Libera o espaço de memória que foi previamente alocado.
- Recebe um ponteiro, o que foi usado para receber o endereço do bloco de memória alocada, e não retorna nada.
- Sintaxe:

Free(ponteiro)

Exercicio

- Alterar o exercício anterior para alocar espaço de memória para o ponteiro que conta as vogais.
- Utilizar o free para liberar o endereço de memória utilizado no malloc