

Algoritmos e Programação de Computadores

Introdução à Ponteiros

Prof. Lucas Boaventura lucas.boaventura@unb.br



Introdução

- Em um programa em execução, a manipulação de variáveis indicam escrita e armazenamento de dados em memória
- As variáveis que um programador declara, são abstrações, espaços de memória que são reservados e traduzidos pelo compilador



Introdução

- O nome das variáveis não é importante para o programa, mas sim para o programador
- Já o espaço que elas estão reservadas, é importante para o programa e frequentemente é escondido do programador
- No entanto, variáveis do tipo ponteiro podem ser utilizadas para manipular e armazenar endereços de memória



- Uma variável, ela armazena um valor que pode ser lido e escrito durante a execução do programa
- Um ponteiro é uma variável que armazena um endereço de memória
- tipo_do_ponteiro *nome_do_ponteiro;



```
#include <stdio.h>
int main()
{
   int i;
   int *p;

   return 0;
}
```



- O operador * também pode ser utilizado para multiplicação
- No entanto, devido aos tipos de dados e ordem de utilização, não existe ambiguidade
- Outro operador que já utilizamos bastante é o operador & que retorna o endereço de uma variável



 Um ponteiro pode receber o endereço de uma variável utilizando o operador &

```
int main()
{
    int i;
    int *p;

    i = 0;
    p = &i;
    printf("%p\n", p);
    return 0;
}
```

- Fazer atribuições sem o operador p = i ou i = p, geram um aviso (warning) no compilador
- %p imprime endereço de memória em hexadecimal

Ex saída: 0x7ffc09f6ed68 (Em decimal 167177576)



Endereço de Memória		Valor	Variável
	0x7ffc09f6ed68	0	i
	0x7ffc09f6ed60		
	0x7ffc09f6ed5c	0x7ffc09f6ed68	р



 É possível fazer a atribuição no conteúdo do ponteiro, e não apenas no valor do ponteiro, usando o operador *



```
int main()
{
    int i = 10;
    int *p;
    p = \&i;
    printf("Valor antes: %d/%d\n", i, *p);
    *p = 20;
    printf("Valor depois: %d/%d\n", i, *p);
    return 0;
                   Saída do programa:
}
                    Valor antes: 10/10
                   Valor depois: 20/20
```



- Note que, ao alterar o conteúdo, a variável que é apontada também é alterada
- E se você precisar inicializar o ponteiro com um valor "padrão", antes de decidir qual o valor que ele irá receber?
- Para isso, utiliza-se a atribuição NULL
- Cuidado! Não se pode escrever em conteúdo NULL (runtime error)



```
#include <stdio.h>
int main()
{
    int *p = NULL;
    *p = 10; //Programa termina
    return 0;
}
```

Ex. de execução:

[user@station ~]\$./4 Segmentation fault (core dumped)



Ponte_{

 É possível utilizar os ponteiros dentro de condicionais, verificando se o seu conteúdo é válido ou comparando dois ponteiros

```
int main()
    int *p = NULL;
    int *p1 = NULL;
    if (p == NULL)
    if (p == p1)
    return 0;
```



- Os tipos do ponteiro são muito importantes
- Apesar de ser possível, não é recomendado atribuir ponteiros de diferentes tipos
- O compilador gera um aviso

```
int main()
{
    char *p1;
    int *p2;

    p2 = p1;
    return 0;
}
```

warning: assignment to 'int *' from incompatible pointer type 'char *



- O tipo void é um tipo de ponteiro genérico
- Pode receber valor de qualquer outro
- Ao utilizar, é preciso fazer o cast (conversão) para o tipo desejado

```
int main()
{
    int i = 10;
    int *p;
    void *v;

    v = &i;
    printf("%d\n", *(int*)v);
    return 0;
}
```

```
#include <stdio.h>
int main()
   int x = 10;
   int y = 20;
   int *a;
                                           Endereço
                                                        Valor
                                                                     Variável
                                           de
   printf("Valor antes %d %d\n", x, y);
                                           Memória
   a = &x;
   *a = 5;
                                           0x7f..6
                                                         LIXO
   a = &y;
                                               8
   *a = 6;
   a = NULL;
                                           0x7f..6
   printf("Valor depois %d %d\n", x, y);
                                                         LIXO
   return 0;
                                           0x7f...5
                                                         LIXO
```

```
#include <stdio.h>
int main()
  int x = 10;
   int y = 20;
   int *a;
   printf("Valor antes %d %d\n", x, y);
   a = &x;
   *a = 5;
   a = &y;
   *a = 6;
   a = NULL;
   printf("Valor depois %d %d\n", x, y);
   return 0;
```

Endereço de Memória	Valor	Variável
0x7f6 8	10	X
0x7f6 0	LIXO	
0x7f5 c	LIXO	

```
#include <stdio.h>
int main()
    int x = 10;
   int y = 20;
    int *a;
    printf("Valor antes %d %d\n", x, y);
    a = &x;
    *a = 5;
    a = &y;
    *a = 6;
    a = NULL;
    printf("Valor depois %d %d\n", x, y);
   return 0;
```

```
Endereço
            Valor
                        Variável
de
Memória
0x7f..6
               10
                           X
   8
0x7f..6
              20
0x7f...5
             LIXO
```

```
#include <stdio.h>
int main()
   int x = 10;
   int y = 20;
   int *a;
                                           Endereço
                                                         Valor
                                           de
   printf("Valor antes %d %d\n", x, y);
                                           Memória
   a = &x;
   *a = 5;
                                            0x7f..6
                                                            10
   a = &y;
                                               8
   *a = 6;
   a = NULL;
                                            0x7f..6
   printf("Valor depois %d %d\n", x, y);
                                                            20
   return 0;
                                           0x7f...5
                                                          LIXO
```

Variável

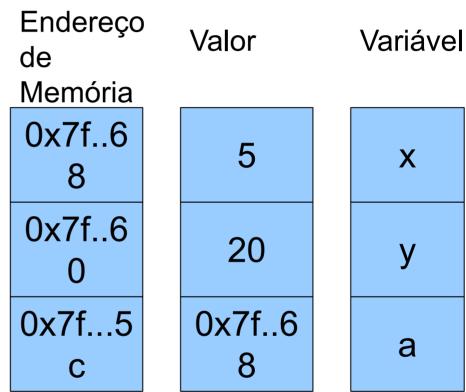
X

a

```
#include <stdio.h>
int main()
   int x = 10;
   int y = 20;
   int *a;
                                           Endereço
                                                        Valor
                                                                     Variável
                                           de
   printf("Valor antes %d %d\n", x, y);
                                           Memória
   a = &x;
   *a = 5;
                                           0x7f..6
                                                           10
   a = &y;
                                                                        X
                                               8
   *a = 6;
   a = NULL;
                                           0x7f..6
   printf("Valor depois %d %d\n", x, y);
                                                           20
   return 0;
                                           0x7f...5
                                                          LIXO
                                                                        a
 Saída:
```

```
#include <stdio.h>
int main()
   int x = 10;
   int y = 20;
   int *a;
                                            Endereço
                                                         Valor
                                                                       Variável
                                            de
   printf("Valor antes %d %d\n", x, y);
                                            Memória
    a = &x;
    *a = 5;
                                            0x7f..6
                                                            10
   a = &y;
                                                                          X
                                               8
   *a = 6;
   a = NULL;
                                            0x7f..6
   printf("Valor depois %d %d\n", x, y);
                                                            20
   return 0;
                                            0x7f...5
                                                         0x7f..6
                                                                          a
                                                             8
```

```
#include <stdio.h>
int main()
    int x = 10;
    int y = 20;
    int *a;
    printf("Valor antes %d %d\n", x, y);
    a = &x;
    *a = 5;
    a = &y;
    *a = 6;
    a = NULL;
    printf("Valor depois %d %d\n", x, y);
   return 0;
```



```
#include <stdio.h>
int main()
   int x = 10;
   int y = 20;
   int *a;
                                            Endereço
                                                         Valor
                                           de
   printf("Valor antes %d %d\n", x, y);
                                            Memória
    a = &x;
   *a = 5;
                                            0x7f..6
                                                             5
   a = &y;
                                               8
    *a = 6;
   a = NULL;
                                            0x7f..6
   printf("Valor depois %d %d\n", x, y);
                                                            20
   return 0;
                                            0x7f...5
                                                         0x7f..6
```

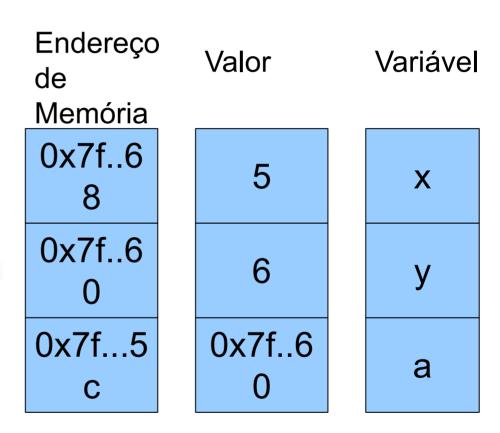
Variável

X

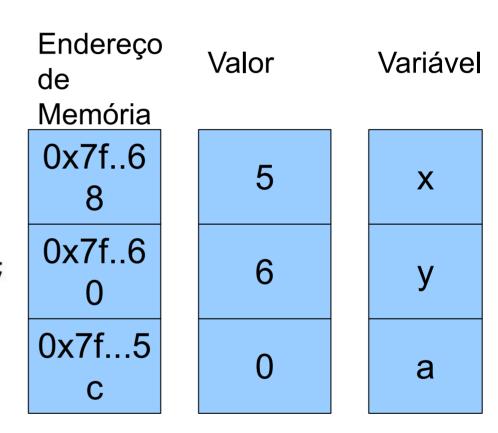
a

Saída:

```
#include <stdio.h>
int main()
    int x = 10;
    int y = 20;
    int *a;
    printf("Valor antes %d %d\n", x, y);
    a = &x;
    *a = 5;
    a = &y;
    *a = 6;
    a = NULL;
    printf("Valor depois %d %d\n", x, y);
   return 0;
```



```
#include <stdio.h>
int main()
    int x = 10;
    int y = 20;
    int *a;
    printf("Valor antes %d %d\n", x, y);
    a = &x;
    *a = 5;
    a = &y;
    *a = 6;
    a = NULL;
    printf("Valor depois %d %d\n", x, y);
   return 0;
```



```
#include <stdio.h>
int main()
   int x = 10;
   int y = 20;
   int *a;
   printf("Valor antes %d %d\n", x, y);
   a = &x;
   *a = 5;
   a = &y;
   *a = 6;
   a = NULL;
   printf("Valor depois %d %d\n", x, y);
   return 0;
 Saída:
 Valor antes 10 20
```

Endereço Valor Variável de Memória 0x7f..6 5 X 8 0x7f..6 6 0x7f...5 a

Valor depois 5 6



- O uso de ponteiros em C é obrigatório quando é desejado que uma função altere algum argumento
- Compare as saídas de dois programas:



```
int func(int *arg)
int func(int arg)
                                      *arg = *arg + 1;
    arg = arg + 1;
                                      printf("Func: %d\n", *arg);
    printf("Func: %d\n", arg);
}
                                  int main()
int main()
                                      int i = 0;
    int i = 0;
                                      func(&i);
    func(i);
                                      printf("Main: %d\n", i);
    printf("Main: %d\n", i);
    return 0;
                                      return 0;
```

Saída: Func: 1 Main: 0

Saída: Func: 1 Main: 1



- Quando se deseja passar uma varíavel que seja alterada por uma função, em C, devemos utilizar ponteiros: passagem por referência
- Quando deseja-se que uma variável NÃO seja alterada pela função que é passada, não se utilizam ponteiros: passagem por valor



- Passagem por referência e passagem por valor, são conceitos que existem em TODAS as linguagens de programação
- Isso define a propriedade de uma função alterar ou não o valor de uma variável
- Em C, deve-se utilizar ponteiros para a passagem por referência



- Por isso, devemos utilizar: scanf("%d", &i);
- Isso acontece pois queremos que a função "scanf" altere o valor de i
- Precisamos passar por referência a variável



- Mas, quando queremos ler uma string, utilizamos scanf("%s", str)
- E não precisamos usar &
- Por quê?



- Ponteiros e arrays são conceitos fortemente ligados na linguagem C
- Arrays são agrupamentos do mesmo tipo de dados
- Quando se deseja utilizar alguma variável do vetor, utilizamos o operador []
- No entanto, a variável do vetor, possui uma a área de memória (ou seja, um ponteiro)



- Por isso, um ponteiro pode ser igualado a um vetor
- Vimos que o operador * pode ser utilizado para acessar o dado do ponteiro
 - printf("%d\n", *p);
- Mas o operador [0] é equivalente:
 - printf("%d\n", p[0]);



```
int main()
                int i = 5;
                int *p = \&i;
                printf("%d\n", *p);
                printf("%d\n", p[0]);
                return 0;
Saída do programa:
```



- Neste exemplo, utilizar p[1] irá acessar uma posição inválida de memória
- segmentation fault (runtime error no MOJ) é um dos problemas mais comuns ao se utilizar ponteiros
- Mas esse é um comando válido ao se utilizar arrays



```
int main()
               int arr[3] = \{2, 4, 8\};
               int *p = arr;
               printf("%d\n", *p);
               printf("%d\n", p[0]);
               printf("%d\n", p[2]);
               return 0;
Saída do programa:
```



- Note a diferença usando array, ponteiro e variável comuns
- Um ponteiro pode ser igualado ao endereço de uma variável (de mesmo tipo)
- Uma variável de ponteiro pode ser igualada a um vetor!



```
int main()
                              Saída do
    int i = 1;
                              programa: 1
    int arr[3] = \{2, 4, 8\};
    int *p;
    p = \&i;
    printf("%d\n", *p);
    p = arr;
    printf("%d\n", *p);
    return 0;
```



```
int main()
                                  Saída do
    int i = 1;
                                   programa: 1
    int arr[3] = \{2, 4, 8\};
    int *p;
    p = \frac{\mathbb{Q}}{\&i};
    printf("%d\n", *p);
    p = arr;
    printf("%d\n", *p);
    return 0;
```



 No entanto, uma variável de vetor NÃO pode ser igualada a vetor

```
int main()
{
    int arr[3] = {2, 4, 8}; error: assignment
    int *p; to expression with
    arr = p; // ERRO array type
    return 0;
}
```



Mas podemos utilizar o operador * para um vetor

```
int main()
{
    int arr[3] = {2, 4, 8};
    int *p;

    printf("%d\n", *arr);
    return 0;
}
```

Saída do programa: 2



- Strings são vetores, por isso ele pode alterar o conteúdo do array sem utilizar o operador &, basta uma referência para string
- scanf("%s", str);



Resumo

- Ponteiros podem ser declarados com operador *
- int *p;
- Os ponteiros armazenam o valor de uma área de memória
- Normalmente, esse valor pode ser acessado novamente com o operador *
- *p = 0;



Resumo

- Funções podem alterar o valor do argumento se forem ponteiros: passagem por referência
- Quando uma função não pode alterar o valor do argumento é passagem por valor
- Arrays e ponteiros são conceitos muito ligados e pode-se utilizar os operador * ou [] para acessar o conteúdo



Dúvidas?

lucas.boaventura@unb.br