



# Algoritmos e Programação de Computadores

Introdução à Ponteiros

Prof. Lucas Boaventura  
[lucas.boaventura@unb.br](mailto: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





# Ponteiro

- 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;`





# Ponteiro

```
#include <stdio.h>

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

    return 0;
}
```





# Ponteiro

- 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





# Ponteiro

- 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)



# Ponteiro

Endereço de Memória

Valor

Variável

...

...

...

0x7ffc09f6ed68

0

i

0x7ffc09f6ed60

0x7ffc09f6ed5c

0x7ffc09f6ed68

p

...

...

...







# Ponteiro

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





# Ponteiro

```
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





# Ponteiro

- 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)





# Ponteiro

```
#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;
}
```





# Ponteiro

- 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 \*'





# Ponteiro

- 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;
}
```

Saída: 10



```
#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:

Endereço  
de  
Memória

Valor

Variável

0x7f..6 8
0x7f..6 0
0x7f...5 c

LIXO
LIXO
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;
```

```
}
```

Saída:

Endereço  
de  
Memória

Valor


Variável

0x7f..6 8	10	x
0x7f..6 0	LIXO	
0x7f...5 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;  
}
```

Saída:

Endereço de Memória	Valor	Variável
0x7f..68	10	x
0x7f..60	20	y
0x7f...5c	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;
```

```
}
```

Saída:

Endereço  
de  
Memória

Valor

Variável

0x7f..6  
8

10

x

0x7f..6  
0

20

y

0x7f...5  
c

LIXO

a



```
#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  
de  
Memória

Valor

Variável

0x7f..6 8	10	x
0x7f..6 0	20	y
0x7f...5 c	LIXO	a



```
#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

0x7f..6  
8

10

x

0x7f..6  
0

20

y

0x7f...5  
c

0x7f..6  
8

a

Saída:

Valor antes 10 20



```
#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

0x7f..6  
8

5

x

0x7f..6  
0

20

y

0x7f...5  
c

0x7f..6  
8

a

Saída:

Valor antes 10 20



```
#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

0x7f..6  
8

5

x

0x7f..6  
0

20

y

0x7f...5  
c

0x7f..6  
0

a

Saída:

Valor antes 10 20



```
#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

0x7f..6  
8

5

x

0x7f..6  
0

6

y

0x7f...5  
c

0x7f..6  
0

a

Saída:

Valor antes 10 20





```
#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

0x7f..6  
8

5

x

0x7f..6  
0

6

y

0x7f...5  
c

0

a

Saída:

Valor antes 10 20



```
#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

0x7f..6  
8

5

x

0x7f..6  
0

6

y

0x7f...5  
c

0

a

Saída:

Valor antes 10 20

Valor depois 5 6





# Funções

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





# Funções

```
int func(int arg)
{
    arg = arg + 1;
    printf("Func: %d\n", arg);
}
```

```
int main()
{
    int i = 0;
    func(i);
    printf("Main: %d\n", i);
    return 0;
}
```

Saída: Func: 1  
Main: 0

```
int func(int *arg)
{
    *arg = *arg + 1;
    printf("Func: %d\n", *arg);
}
```

```
int main()
{
    int i = 0;
    func(&i);
    printf("Main: %d\n", i);
    return 0;
}
```

Saída: Func: 1  
Main: 1





# Funções

- Quando se deseja passar uma variável 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**





# Funções

- 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





# Funções

- 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





# Funções

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







# Ponteiro e Array

- 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 área de memória (ou seja, um ponteiro)





# Ponteiro e Array

- 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]);`





# Ponteiro e Array

```
int main()  
{  
    int i = 5;  
    int *p = &i;  
  
    printf("%d\n", *p);  
    printf("%d\n", p[0]);  
    return 0;  
}
```

Saída do programa:

5

5





# Ponteiro e Array

- 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





# Ponteiro e Array

```
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: 2  
2  
8





# Ponteiro e Array

- 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!





# Ponteiro e Array

```
int main()
{
    int i = 1;
    int arr[3] = {2, 4, 8};
    int *p;

    p = &i;
    printf("%d\n", *p);

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

Saída do  
programa: 1  
2





# Ponteiro e Array

```
int main()
{
    int i = 1;
    int arr[3] = {2, 4, 8};
    int *p;

    ↓
    p = &i;
    printf("%d\n", *p);

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

Saída do  
programa: 1  
2







# Ponteiro e Array

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

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

    arr = p; // ERRO
    return 0;
}
```

error: assignment  
to expression with  
array type





# Ponteiro e Array

- 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





# Ponteiro e Array

- 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](mailto:lucas.boaventura@unb.br)

