



UNIVERSIDADE
FEDERAL DO CEARÁ



CK0179 – Programação Computacional para Engenharia: Ponteiros

Prof. Maurício Moreira Neto

Objetivo

- Aprender sobre a teoria por trás da utilização de ponteiros
- Aprender a utilizar ponteiros na linguagem C

Definição (relembrando...)

- Variável
 - E um espaço reservado de memória usado para guardar um valor que pode ser modificado pelo programa
- Ponteiro
 - **E um espaço reservado de memória usado para guardar o *endereço de memória* de uma outra variável**
 - Um ponteiro é uma variável como qualquer outra do programa: sua diferença é que esta não armazena um valor inteiro, real, caractere ou logico (booleano)
 - Serve para armazenar endereços de memória (são valores inteiros sem sinal)

Declaração

- Assim como um variável, um ponteiro também possui um tipo

```
//declaração de variável  
tipo_variável nome_variável;
```

```
//declaração de ponteiro  
tipo_ponteiro *nome_ponteiro;
```

- O asterisco(*) informa ao compilador que aquela variável não vai guardar um valor mas sim um endereço para o tipo específico

```
int x;  
float y;  
struct ponto p;  
  
int *x;  
float *y;  
struct ponto *p;
```

Declaração

- Exemplos de declaração de variáveis e ponteiros

```
int main() {  
    //Declara um ponteiro para int  
    int *p;  
    //Declara um ponteiro para float  
    float *x;  
    //Declara um ponteiro para char  
    char *y;  
    //Declara um ponteiro para struct ponto  
    struct ponto *p;  
    //Declara uma variável do tipo int e um ponteiro para int  
    int soma, *p2, ;  
  
    return 0;  
}
```

Declaração

- Na linguagem C, quando declaramos um ponteiro precisamos informar ao compilador para que tipo de variável vamos apontá-la
 - Um ponteiro **int*** aponta para um inteiro
 - Esse ponteiro guarda o endereço de memória onde se encontra armazenada uma variável do tipo **int**

Inicialização

- Ponteiros apontam para uma posição de memória
 - **Cuidado:** ponteiros não inicializados apontam para um lugar indefinido!!

- Exemplo:
 - `int *p;`

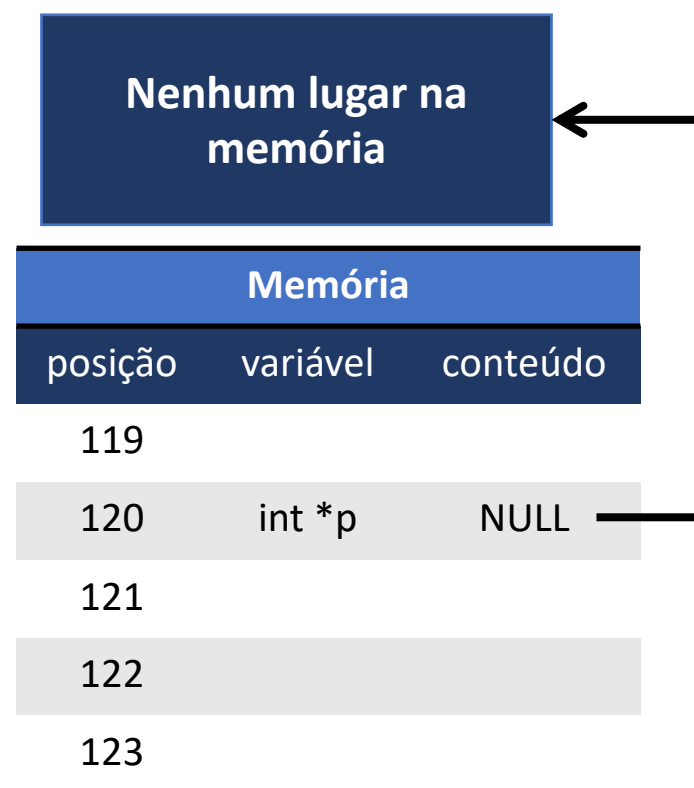
Memória		
posição	variável	conteúdo
119		
120	int *p	????
121		
122		
123		

Inicialização

- Um ponteiro pode ter o valor especial NULL que é o endereço de nenhum lugar

- Exemplo:


- `int *p = NULL;`



Inicialização

- Os ponteiros devem ser inicializados antes de serem usados
- Assim devemos apontar um ponteiro para um lugar conhecido
 - Podemos apontá-lo para uma variável que já exista no programa

Memória		
posição	variável	conteúdo
119		
120	int *p	122
121		
122	int c	10
123		

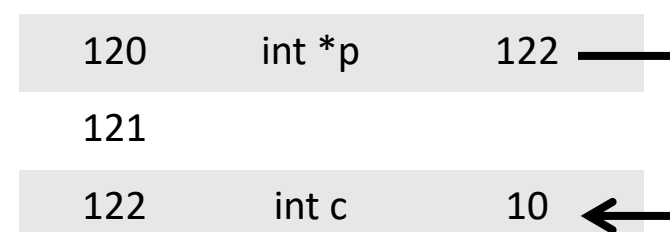


Inicialização

- O ponteiro armazena o endereço da variável para onde ela aponta
 - Para saber o endereço de memória de uma variável do nosso programa, usamos o operador **&**
 - Ao armazenar o endereço, o ponteiro estará apontando para aquela variável

```
int main(){  
    //Declara uma variável int contendo o valor 10  
    int c = 10;  
    //Declara um ponteiro para int  
    int *p;  
    //Atribui ao ponteiro o endereço da variável int  
    p = &c ;  
  
    return 0;  
}
```

Memória		
posição	variável	conteúdo
119		
120	int *p	122
121		
122	int c	10
123		



Utilização

- Tendo um ponteiro armazenado um endereço de memória, como saber o valor guardado dentro dessa posição?

Utilização

- Para acessar o **valor** guardado dentro de uma posição na memória apontada por um ponteiro, basta usar o operador ***asterisco*** `*` na frente do nome do ponteiro

```
int main() {  
    //Declara uma variável int contendo o valor 10  
    int c = 10;  
    //Declara um ponteiro para int  
    int *p;  
    //Atribui ao ponteiro o endereço da variável int  
    p = &c;  
    printf("Conteudo apontado por p: %d \n", *p); // 10  
    //Atribui um novo valor à posição de memória apontada por p  
    *p = 12;  
    printf("Conteudo apontado por p: %d \n", *p); // 12  
    printf("Conteudo de count: %d \n", c); // 10  
  
    return 0;  
}
```

Utilização

- ***p** : conteúdo da posição de memória apontado por **p**
- **&c** : o endereço na memória onde está armazenada a variável **c**

```
int main(){
    //Declara uma variável int contendo o valor 10
    int c = 10;
    //Declara um ponteiro para int
    int *p;
    //Atribui ao ponteiro o endereço da variável int
    p = &c;
    printf("Conteúdo apontado por p: %d \n", *p); // 10
    //Atribui um novo valor à posição de memória apontada por p
    *p = 12;
    printf("Conteúdo apontado por p: %d \n", *p); // 12
    printf("Conteúdo de count: %d \n", c); // 12

    return 0;
}
```

Utilização

- De modo geral, um ponteiro só pode receber o endereço de memória de uma variável do mesmo tipo do ponteiro
 - Isso ocorre porque diferentes tipos de variáveis ocupam espaços de memória de tamanhos diferentes
 - É possível atribuir a um ponteiro de inteiro (**int ***) o endereço de uma variável do tipo **float**. No entanto, o compilador assume qualquer endereço que esse ponteiro armazene obrigatoriamente apontará para uma variável do tipo **int**
 - Isso pode gerar problemas na interpretação dos valores

Utilização

```
int main() {  
    int *p, *p1, x = 10;  
    float y = 20.0;  
    p = &x;  
    printf("Conteudo apontado por p: %d \n", *p);  
  
    p1 = p;  
    printf("Conteudo apontado por p1: %d \n", *p1);  
  
    p = &y;  
    printf("Conteudo apontado por p: %d \n", *p);  
    printf("Conteudo apontado por p: %f \n", *p);  
    printf("Conteudo apontado por p: %f \n", *((float*)p));  
  
    return 0;  
}
```

```
Conteudo apontado por p: 10  
Conteudo apontado por p1: 10  
Conteudo apontado por p: 1101004800  
Conteudo apontado por p: 0.000000  
Conteudo apontado por p: 20.000000
```

Operadores com Ponteiros

- Atribuição

- P1 aponta para o mesmo lugar que p2

```
int *p, *p1;  
int c = 10;  
p = &c;  
p1 = p; //equivale a p1 = &c;
```

- A variável apontada por p1 recebe o mesmo conteúdo da variável apontada por p2

```
int *p, *p1;  
int c = 10, d = 20;  
p = &c;  
p1 = &d;  
  
*p1 = *p; //equivale a d = c;
```


Operadores com Ponteiros

- Apenas duas operações aritméticas podem ser utilizadas no endereço armazenado pelo ponteiro: **adição** e **subtração**
- Podemos apenas somar e subtrair valores INTEIROS!
 - $p++$
 - Soma +1 no endereço armazenado no ponteiro
 - $p--$
 - Subtrai 1 no endereço armazenado no ponteiro
 - $p = p+30$
 - Soma +30 no endereço armazenado no ponteiro
 - ...

Operadores com Ponteiros

- As operações de adição e subtração no endereço dependem do tipo de dado que o ponteiro aponta
 - Considere um ponteiro para inteiro, **int ***
 - O tipo **int** ocupa um espaço de 4 bytes na memória
 - Assim, nas operações de adição e subtração são adicionados/subtraídos 4 bytes por incremento/decremento, pois esse é o tamanho de um inteiro na memória e, portanto, é também o valor mínimo necessário para sair dessa posição reservada de memória

Memória		
posição	variável	conteúdo
119		
120	int a	10
121		
122		
123		
124	int b	20
125		
126		
127		
128	char c	'k'
129	char d	's'
130		

Operadores com Ponteiros

- Operações ilegais com ponteiros
 - Dividir ou multiplicar ponteiros
 - Somar o endereço de dois ponteiros
 - Não se pode adicionar ou subtrair valores dos tipos float ou double de ponteiros

Operadores com Ponteiros

- Já sobre seu conteúdo apontado, valem todas as operações
 - `(*p)++;`
 - incrementar o conteúdo da variável apontada pelo ponteiro `p`;
 - `*p = (*p) * 10;`
 - multiplica o conteúdo da variável apontada pelo ponteiro `p` por 10;

```
int *p;  
int c = 10;  
  
p = &c;  
  
(*p)++;  
*p = (*p) * 10;
```

Operadores com Ponteiros

- **Operações relacionais**

- == e != para saber se dois ponteiros são iguais ou diferentes
- >, <, >= e <= para saber qual ponteiro aponta para uma posição mais alta na memória

```
int main() {  
    int *p, *p1, x, y;  
    p = &x;  
    p1 = &y;  
    if (p == p1)  
        printf("Ponteiros iguais\n");  
    else  
        printf("Ponteiros diferentes\n");  
  
    return 0;  
}
```

Ponteiros Genéricos

- Normalmente, um ponteiro aponta para um tipo específico de dado.
 - Um ponteiro genérico é um ponteiro que pode apontar para qualquer tipo de dado.
- Declaração

```
void *nome_ponteiro;
```

Ponteiros Genéricos

- Exemplos:

```
int main() {  
    void *pp;  
    int *p1, p2 = 10;  
    p1 = &p2;  
    //recebe o endereço de um inteiro  
    pp = &p2;  
    printf("Endereco em pp: %p \n", pp);  
    //recebe o endereço de um ponteiro para inteiro  
    pp = &p1;  
    printf("Endereco em pp: %p \n", pp);  
    //recebe o endereço guardado em p1 (endereço de p2)  
    pp = p1;  
    printf("Endereco em pp: %p \n", pp);  
  
    return 0;  
}
```

Ponteiros Genéricos

- Para acessar o **conteúdo** de um ponteiro genérico é preciso antes convertê-lo para o tipo de ponteiro com o qual se deseja trabalhar
 - Isso é feito via *type cast*

```
int main(){
    void *pp;
    int p2 = 10;
    // ponteiro genérico recebe o endereço de um
    // inteiro
    pp = &p2;
    //enta acessar o conteúdo do ponteiro genérico
    printf("Conteudo: %d\n", *pp); //ERRO
    // converte o ponteiro genérico pp para (int *)
    // antes de acessar seu conteúdo.
    printf("Conteudo: %d\n", *(int*)pp); //CORRETO

    return 0;
}
```


Ponteiros e Arrays

- Ponteiros e arrays possuem uma ligação muito forte
 - Arrays são agrupamentos de dados do mesmo tipo na memória
 - Quando declaramos um array, informamos ao computador para reservar uma certa quantidade de memória a fim de armazenar os elementos do array de forma sequencial
 - Como resultado dessa operação, o computador nos devolve um ponteiro que aponta para o começo dessa sequência de bytes na memória

Ponteiros e Arrays

- O nome do array (sem índice) é apenas um ponteiro que aponta para o primeiro elemento do array

```
int vet[5] = {1,2,3,4,5};  
int *p;  
  
p = vet;
```

Memória		
posição	variável	conteúdo
119		
120		
121	int *p	123
122		
123	int vet[5]	1
124		2
125		3
126		4
127		5
128		



Ponteiros e Arrays

- Os colchetes [] substituem o uso conjunto de operações aritméticas e de acesso ao conteúdo (operador “*”) no acesso ao conteúdo de uma posição de um array ou ponteiro.
 - O valor entre colchetes é o deslocamento a partir da posição inicial do array.
 - Nesse caso, **p[2]** equivale a ***(p+2)**.

```
int main () {  
    int vet[5] = {1,2,3,4,5};  
    int *p;  
    p = vet;  
  
    printf("Terceiro elemento: %d ou %d", p[2], *(p+2));  
  
    return 0;  
}
```

Ponteiros e Arrays

- Nesse exemplo

```
int vet[5] = {1, 2, 3, 4, 5};  
int *p;  
  
p = vet;
```

- Temos que:
 - *p é equivalente a vet[0]
 - vet[índice] é equivalente a *(índice)
 - vet é equivalente a &vet[0]
 - &vet[índice] é equivalente a (vet + índice)

Ponteiros e Arrays

Usando array

```
int main() {  
    int vet[5] = {1, 2, 3, 4, 5};  
    int *p = vet;  
    int i;  
    for (i = 0; i < 5; i++)  
        printf("%d\n", p[i]);  
  
    return 0;  
}
```

Usando ponteiro

```
int main() {  
    int vet[5] = {1, 2, 3, 4, 5};  
    int *p = vet;  
    int i;  
    for (i = 0; i < 5; i++)  
        printf("%d\n", *(p+i));  
  
    return 0;  
}
```

Ponteiros e Arrays

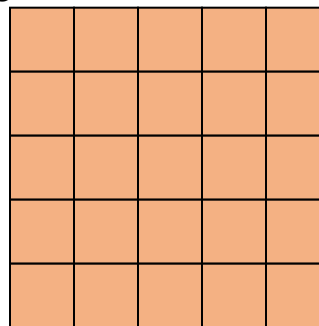
- Arrays Multidimensionais

- Apesar de terem mais de uma dimensão, na memória os dados são armazenados linearmente

- Exemplo:

- `int mat[5][5]`

0,0



4,4

0,0

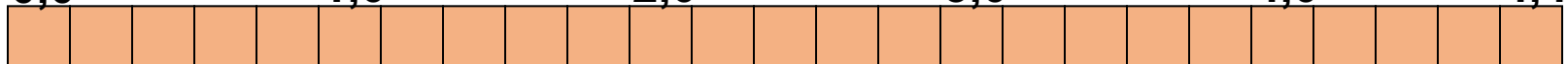
1,0

2,0

3,0

4,0

4,4



Ponteiros e Arrays

- Pode-se então percorrer as várias dimensões do array como se existisse apenas uma dimensão. As dimensões mais a direita mudam mais rápido

Usando array

```
int main() {  
    int mat[2][2] = {{1,2},{3,4}};  
    int i,j;  
    for(i=0;i<2;i++)  
        for(j=0;j<2;j++)  
            printf("%d\n", mat[i][j]);  
  
    return 0;  
}
```

Usando ponteiro

```
int main() {  
    int mat[2][2] = {{1,2},{3,4}};  
    int *p = &mat[0][0];  
    int i;  
    for(i=0;i<4;i++)  
        printf("%d\n", *(p+i));  
  
    return 0;  
}
```

Ponteiros para Struct

- Existem duas abordagens para acessar o conteúdo de um ponteiro para uma struct:

- Abordagem 1**

- Devemos acessar o conteúdo do ponteiro para struct para somente depois acessar os seus campos e modificá-los.

- Abordagem 2**

- Podemos usar o *operador seta* “->”
- ponteiro->nome_campo**

```
struct ponto {  
    int x, y;  
};
```

```
struct ponto q;  
struct ponto *p;
```

```
p = &q;
```

```
(*p).x = 10;  
p->y = 20;
```


Ponteiros para Struct

- A linguagem C permite criar ponteiros com diferentes níveis de apontamento
 - É possível criar um ponteiro que aponte para outro ponteiro, criando assim vários níveis de apontamento
 - Assim, um ponteiro poderá apontar para outro ponteiro, que, por sua vez, aponta para outro ponteiro, que aponta para um terceiro ponteiro e assim por diante

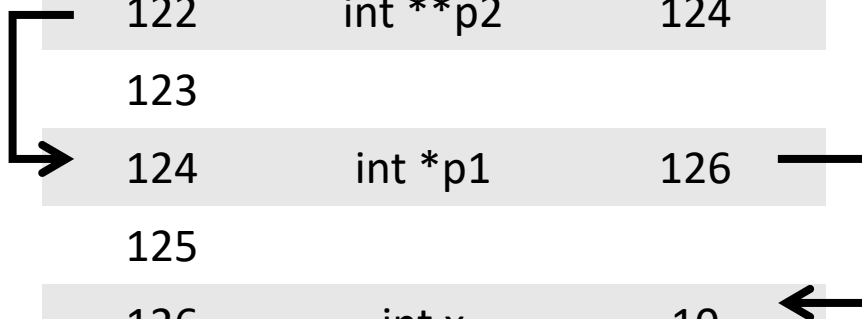
Ponteiros para Struct

- 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_ponteiro **nome_ponteiro;`
- Acesso ao conteúdo
 - `**nome_ponteiro` é o conteúdo final da variável apontada;
 - `*nome_ponteiro` é o conteúdo do ponteiro intermediário

Ponteiros para Ponteiro

```
int x = 10;
int *p1 = &x;
int **p2 = &p1;
//Endereço em p2
printf("Endereco em p2: %p\n", p2);
//Conteúdo do endereço
printf("Conteudo em *p2: %p\n", *p2);
//Conteúdo do endereço do endereço
printf("Conteudo em **p2: %d\n", **p2);
```

Memória		
posição	variável	conteúdo
119		
120		
121		
122	int **p2	124
123		
124	int *p1	126
125		
126	int x	10
127		



Ponteiros para Ponteiro

- É a quantidade de asteriscos (*) na declaração do ponteiro que indica o número de níveis de apontamento que ele possui

```
//variável inteira  
int x;  
//ponteiro para um inteiro (1 nível)  
int *p1;  
//ponteiro para ponteiro de inteiro (2 níveis)  
int **p2;  
//ponteiro para ponteiro para ponteiro de inteiro (3 níveis)  
int ***p3;
```

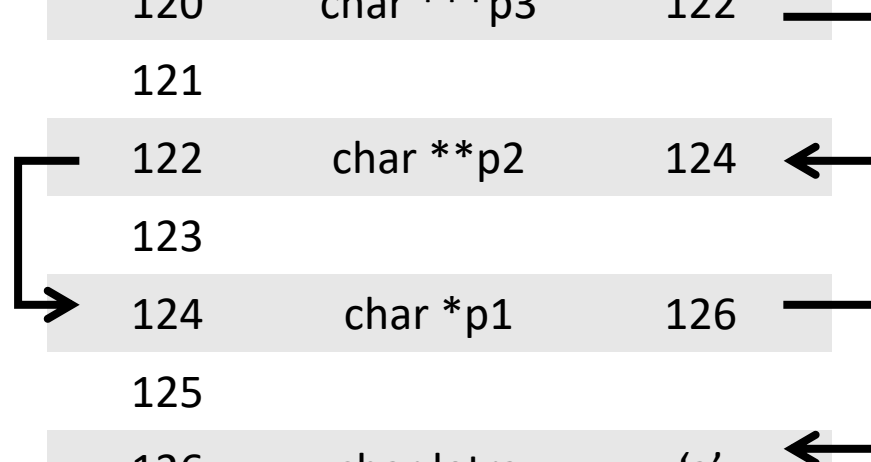
Ponteiros para Ponteiro

- Conceito de “ponteiro para ponteiro”:

```
char letra = 'a';
char *p1;
char **p2;
char ***p3;
```

```
p1 = &letra;
p2 = &p1;
p3 = &p2;
```

Memória		
posição	variável	conteúdo
119		
120	char ***p3	122
121		
122	char **p2	124
123		
124	char *p1	126
125		
126	char letra	'a'
127		



Dúvidas?



Referências

- André Luiz Villar Forbellone, Henri Frederico Eberspächer, Lógica de programação (terceira edição), Pearson, 2005, ISBN 9788576050247.
- Ulysses de Oliveira, Programando em C - Volume I - Fundamentos, editora Ciência Moderna, 2008, ISBN 9788573936599.
- **Slides baseados no material do site "Linguagem C Descomplicado"**
 - <https://programacaodescomplicada.wordpress.com/complementar/>