

Engenharia de Computação

Fundamentos de Programação

Aula 11 – Ponteiros

Prof. Fernando Barreto

- **Revisão de Variáveis em geral:**
 - Possui um nome e tipo
 - USO: Armazenar dados (numéricos/caracteres em geral)
 - É associado a um endereço de memória
 - Ao utilizar uma variável, acessa-se o seu conteúdo/valor através do endereço de memória associado a essa variável
 - Ao atribuir um valor para uma variável, modifica-se o seu conteúdo/valor
 - O **&** serve para obter o endereço de memória associado a uma variável
- **Variável Ponteiro**
 - USO: Armazenar endereço de memória
 - Ferramenta poderosa para acessar qualquer região de memória do programa

- Declaração de Variável Ponteiro
`<type> *nome_variavel;`
- O `<type>` indica como o dado apontado pelo endereço de memória armazenado em `nome_variavel` será interpretado
 - Exemplos: `unsigned int *teste=NULL;`
- Uma variável ponteiro pode receber outro ponteiro ou endereço memória se for do mesmo tipo (exceção se aplicar *casting*)
- Pode-se inicializar com NULL , significa "nenhum lugar"...

- Informações sobre o uso:
 - Variável ponteiro sem * significa acessar o conteúdo da variável ponteiro, que é um endereço de memória
 - Ex: nome_ponteiro = &variavel_comum;
 - Variável ponteiro com * significa acessar um dado na área de memória apontada pelo conteúdo da variável ponteiro
 - Ex: *nome_ponteiro = variavel_comum;
 - Variável ponteiro com & significa acessar o endereço associado a essa variável ponteiro (não o seu conteúdo!!!)

```
int vnumero = 10;
int *exemplo = NULL; //declara uma variável ponteiro exemplo que apontará para uma região
                      //de memória onde terá um número inteiro

printf("Valor de vnumero: %i \n", vnumero); //10
printf("Endereço memória associado à variável vnumero: %p \n", &vnumero); //7ffef13bb010

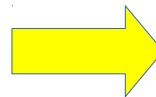
exemplo = &vnumero; //variável ponteiro recebe o endereço associado a vnumero

printf("Endereço memória assoiado à variável ponteiro: %p \n", &exemplo); //0x7ffef13bbc70
printf("Conteúdo da variável ponteiro exemplo: %p \n", exemplo); //0x7ffef13bb010

*exemplo = *exemplo + 15; //modifica conteúdo ponteiro (área de memória apontada)
(*exemplo)++; //incrementa o conteúdo do ponteiro

printf("Valor apontado pelo conteúdo de exemplo: %i \n", *exemplo); //26
```

Endereço	Var	Conteúdo
0x7ffef13bb010	vnumero	10
0x7ffef13bbc70	exemplo	NULL



Endereço	Var	Conteúdo
0x7ffef13bb010	vnumero	26
0x7ffef13bbc70	exemplo	0x7ffef13bb010

- Operações de Adição ou Subtração
 - O conteúdo de um ponteiro (endereço de memória) pode aumentar ou diminuir de x em x bytes, onde x depende do tipo declarado do ponteiro
 - Ex: `int` (tem *sizeof* de 4 bytes), então x será 4
 - OBS: Ponteiro constante (Array) não pode ser alterado

```
int v[5] = {1,2,3,4,5};
int *exemplo = NULL;

exemplo = v;

for(int i=0;i<5;i++){
    printf("%i é igual a %i \n", *exemplo, V[i]);
    exemplo++;
}
```

A cada iteração, o endereço armazenado pelo ponteiro *exemplo* irá aumentar 4 unidades (tipo `int`) apontando para o próximo índice do vetor. Isso é possível pois o um array é um conjunto sequencial de elementos

- **Arrays** declarados são ponteiros constantes !!
 - Um array é um conjunto de elementos adjacentes na memória
 - Ponteiro constante: Não se pode alterar o endereço de memória armazenado
 - Um ponteiro pode apontar para um array do mesmo tipo
 - `[i]` significa deslocamento de `i` itens...

```
int v[5] = {1,2,3,4,5};
int *exemplo = NULL;
```

```
printf("Endereço apontado por v: %p \n", v); //0x1100
printf("Endereço memória de v: %p \n", &v); //0x1100
//nem sempre valores de v e &v são iguais !
//ex: parâmetro de função, ou alocação dinâmica
exemplo = v;
```

[i] indica o deslocamento a partir de 0x1100

```
for(int i=0;i<5;i++){
    printf("Dados: %i é igual a %i e %i \n", exemplo[i], *(exemplo+i), v[i]);
    printf("End. Mem.: %p é igual a %p e %p \n", &exemplo[i], exemplo+i, &v[i]);
}
```

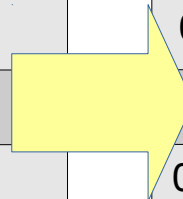
Endereço	Var	Conteúdo
0x1100	int v[0]	1
0x1104	int v[1]	2
0x1108	int v[2]	3
0x110C	int v[3]	4
0x1110	int v[4]	5
...
0x1234	int *exemplo	0x1100

• Passagem de parâmetros (por Referência)

```
void inverter(int *X, int *Y) { //os 1º e 2º parâmetros receberão endereço de memória ou ponteiro
    int tmp=0;
    tmp=*X;
    *X=*Y; //modifica a variável a
    *Y=tmp; //modifica a variável b
}

int main(int argc, char **argv ) {
    int a=14,b=6; //variáveis main(){ }
    inverter(&a,&b); //envia o endereço de memória associado de a e b para a função inverter
    printf("a = %i e b = %i\n", a, b); //mostra-se a e b alterados na função inverter
    return (EXIT_SUCCESS);
}
```

Endereço	Var	Conteúdo
0x7ffef13bb010	a	14
0x7ffef13bbc70	b	6
...
0x8234ffa80023	X	???
0x8234ffa800ab	Y	???
0x8234ffa800d0	tmp	0



Endereço	Var	Conteúdo
0x7ffef13bb010	a	6
0x7ffef13bbc70	b	14
...
0x8234ffa80023	X	0x7ffef13bb010
0x8234ffa800ab	Y	0x7ffef13bbc70
0x8234ffa800d0	tmp	14

- Passagem por referência: structs

```
typedef struct {
    char nome[31];
    int idade;
}tipo_cad;
void zerarUmCad(tipo_cad *Cad) {
    strcpy(Cad->nome, " "); //Marcador -> em struct somente se ponteiro
    Cad->idade = -1; //Marcador -> em struct somente se ponteiro
}
void zerarTodosCad(int tam, tipo_cad *Cad) {
    //Poderia escrever assim: void zerarTodosCad(int tam, tipo_cad Cad[tam]) {
    for(int i=0; i<tam; i++){
        zerarUmCad(&Cad[i]); //poderia usar zerarUmCad(Cad+i);
    }
}
int main(int argc, char **argv ) {
    tipo_cad cad[5];
    zerarUmCad(&cad[3]); //é o mesmo que zerarUmCad(cad+3);
    zerarTodosCad(5, cad);
    return (EXIT_SUCCESS);
}
```

- 1) Faça uma função do tipo void que receba 3 parâmetros reais (a, b, c), sendo o parâmetro c por referência. A função deve fazer o cálculo de $a*b$ e retornar o valor pela variável c. Mostrar o valor da variável c no main().

```
void teste(int tam, V[tam]) {
    printf("Endereco associado a V: %p\n", &V);
    printf("Endereco armazenado em V: %p\n", V);
    for(int i=0;i<tam;i++) {
        printf("TESTE: Endereço memória referente a V[%i]: %p, é o mesmo que %p\n", i, V+i, &V[i]);
        printf("TESTE: Conteúdo de V[%i]: %i, é o mesmo que %i \n", i, V[i], *(V+i));
    }
}

int main(){
    int X[2] = {10,15};
    printf("Endereco associado a X: %p\n", &X);
    printf("Endereco armazenado em X: %p\n", X);
    for(int i=0;i<2;i++) {
        printf("MAIN: Endereço memória referente a X[%i]: %p , é o mesmo que %p\n", i, X+i, &X[i]);
        printf("MAIN: Conteúdo de X[%i]: %i, é o mesmo que %i \n", i, X[i], *(X+i));
    }
    teste(2,X);

    return(EXIT_SUCCESS);
}
```

0x8234ffa80023

0x7ffa425eef0

0x7ffa425eef0

0x7ffa425eef0

Endereço	Var	Conteúdo
0x7ffa425eef0	X[0]	10
0x7ffa425eef4	X[1] ou *(X+1)	15
...
0x8234ffa80023	V ou &V[0]	0x7ffa425eef0
0x8234ffa80027	(V+1) ou &V[1]	0x7ffa425eef4

- A partir da definição que qualquer array unidimensional é um ponteiro, então pode-se definir um ponteiro comum para representar um vetor

```
void zerar(int tam, int *V) { //seria a mesma ideia de: void teste(int tam, int V[tam]) {  
    for(int i=0;i<tam;i++)  
        V[i]=0;  
}  
int main(){  
    int X[2] = {10,15};  
    teste(2, X);  
    return(EXIT_SUCCESS);  
}
```

- E um array multidimensional ?
 - Conceito de ponteiro de ponteiro e como os dados são organizados na memória!!!

- **Array unidimensional (vetor)**

- Elementos de dados em sequência na memória
 - Ex: `int v[5]={1,2,3,4,5};`

<code>*v ou v[0]</code>	1
<code>*(v+1) ou v[1]</code>	2
<code>*(v+2) ou v[2]</code>	3
<code>*(v+3) ou v[3]</code>	4
<code>*(v+4) ou v[4]</code>	5

- **Array multidimensional (matriz)**

- Cada linha é uma espécie de ponteiro para um vetor, que representa as colunas. Deslocamentos de linha ocorre em múltiplos do número de colunas
 - Ex: `int V[3][3]={{1,2,3},{4,5,6},{7,8,9}};`

	Endereço	Var	Conteúdo
	0x1112	<code>int V[][]</code>	0x2256
<code>V ou V[0]</code> →	0x2256	<code>V[0][0] ou *(V[0]+0) ou *(*V+0)+0)</code>	1
	0x225A	<code>V[0][1] ou *(V[0]+1) ou *(*V+0)+1)</code>	2
	0x225E	<code>V[0][2] ou *(V[0]+2) ou *(*V+0)+2)</code>	3
<code>V+1 ou V[1]</code> →	0x2263	<code>V[1][0] ou *(V[1]+0) ou *(*V+1)+0)</code>	4
	0x2267	<code>V[1][1] ou *(V[1]+1) ou *(*V+1)+1)</code>	5
	0x226B	<code>V[1][2] ou *(V[1]+2) ou *(*V+1)+2)</code>	6
<code>V+2 ou V[2]</code> →	0x226F	<code>V[2][0] ou *(V[2]+0) ou *(*V+2)+0)</code>	7
	0x2274	<code>V[2][1] ou *(V[2]+1) ou *(*V+2)+1)</code>	8
	0x2278	<code>V[2][2] ou *(V[2]+2) ou *(*V+2)+2)</code>	9

- Ao declarar uma variável Array Multidimensional, com as dimensões explícitas (ex: `int X[3][5]`), ela é alocada sequencialmente na memória (cada bloco de linha fica em sequência, um bloco após o outro...)
- Com Array Multidimensional dinâmico isso pode não ocorrer

- Variáveis locais e parâmetros de uma função ficam em uma área temporária de memória (*stack*)
 - São liberadas ao término da função
- Uma variável dinâmica é alocada em uma área de memória específica para isso denominada *heap*
 - Permanecem após o término de uma função
 - Otimizar o uso de memória, evitando o uso indiscriminado de variáveis globais ou variáveis com grandes dimensões desnecessariamente
 - Uso racional da memória
 - Alocar arrays dinamicamente
 - Permite que uma variável alocada dentro de uma função possa ser utilizada em outras funções se retornada adequadamente (return)

- Aloca/Desaloca blocos de memória para variáveis em tempo de execução.
 - Faz-se uma requisição de espaço de memória (nº de bytes) ao sistema operacional o qual retorna um ponteiro genérico para a região alocada
 - Deve-se desalocar se não for mais utilizar essa área !!!!!
- Funções <stdlib.h>: malloc(), calloc(), realloc(), free()
 - Exemplo de alocação, supondo que o S.O. forneça as posições de 0x8234ffa80027 até 0x8234ffa80033b

Endereço	Var	Conteúdo
0x7ffa425eef0	int *novo	NULL
...
0x8234ffa80027	livre	
0x8234ffa8002b	livre	
0x8234ffa8002f	livre	
0x8234ffa80033	livre	
0x8234ffa80037	livre	

Alocando
5 posições
de memória



Endereço	Var	Conteúdo
0x7ffa425eef0	int *novo	0x8234ffa80027
...
0x8234ffa80027	novo[0]	
0x8234ffa8002b	novo[1]	
0x8234ffa8002f	novo[2]	
0x8234ffa80033	novo[3]	
0x8234ffa80037	novo[4]	

void *malloc(unsigned int quant)

- Aloca bloco de **quant** bytes e retorna um ponteiro p/ início desse bloco
- Retorna NULL em caso de erro

void *calloc(unsigned int num, unsigned int size)

- Aloca um bloco de (**num * size**) bytes e retorna um ponteiro p/ o início desse bloco. Todo o bloco já vem com os bits em 0
- Retorna NULL em caso de erro

void *realloc(void *ponteiro, unsigned int quant)

- Realoca um bloco previamente alocado para um bloco maior ou menor
 - Ponte retornar o mesmo ***ponteiro** se houver espaço, porém se não houver espaço sequencial, aloca-se uma nova região e copia-se o bloco antigo para o novo bloco, retornando um novo ponteiro...
- Retorna NULL em caso de erro

void free(void *ponteiro)

- Recebe o ponteiro para o início do bloco, desalocando da memória

- Exemplo: Criar Vetor de 5 elementos dinamicamente...

```
int main() {
    int *novo = NULL;

    novo = (int *) malloc(5 * sizeof(int));
    if (novo == NULL) {
        printf("Erro de alocação de mem.");
        exit(1);
    }

    for(int i=0 ; i<5 ; i++) {
        scanf("%i", &novo[i] );
    }
    for(int i=0 ; i<5 ; i++) {
        printf("%i ", novo[i] );
    }

    free(novo); //liberar memória
    novo=NULL; //evita lixo de memória
    return 0;
}
```

- **sizeof(tipo ou variável)** retorna o tamanho em bytes necessário para um tipo (int, char, float, struct) ou variável.

- **void *malloc (tamanho a ser alocado);**

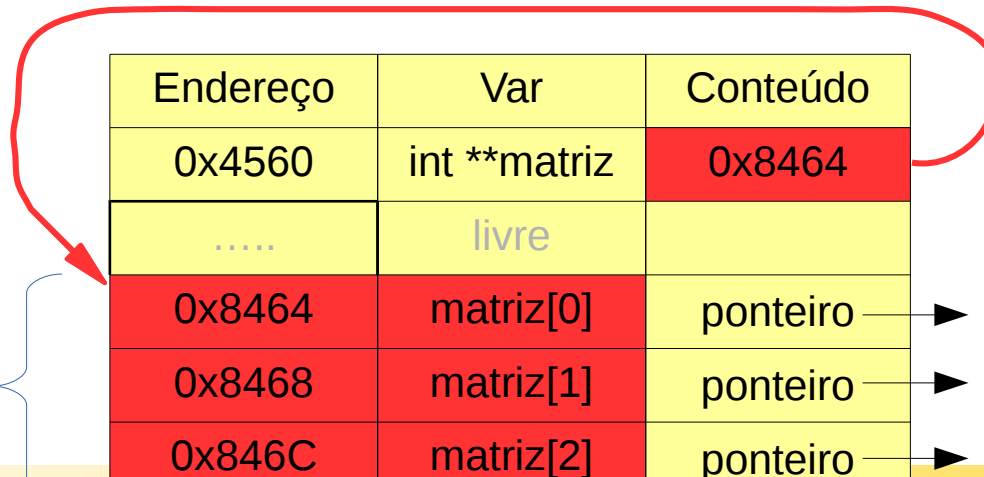
- Retorna um ponteiro para o início da região de memória alocada. No exemplo, alocou-se 20 bytes sendo suficiente para vetor de 5 **int**
- É necessário fazer *casting* devido ao **void ***
- Retorna NULL em caso de erro.

- **novo** pode ser interpretado como um vetor, pois são blocos sequenciais de **int** na memória

- **novo** aponta para uma região de memória, e caso não utilize mais, deve-se liberar essa região utilizando a função **free(ponteiro)** e em seguida, atribua **NULL** ao ponteiro para evitar uso de lixo de memória...

- Array multidimensional (matriz)
 - Alocar dinamicamente um array unidimensional de ponteiros, onde cada item representará um ponteiro de uma linha da matriz. Cada linha terá suas colunas alocadas dinamicamente através de outro array unidimensional. Conceito de "ponteiro de ponteiros" !!!!
 - Exemplo: alocar uma matriz 3x3 de inteiros dinamicamente
 - 1º PASSO: Alocar um vetor de ponteiros, onde cada item representará uma linha da matriz

```
int **matriz = NULL;
int NL=3;
matriz= (int**) malloc(NL*sizeof(int *));
```



Endereço	Var	Conteúdo
0x4560	int **matriz	0x8464
.....	livre	
0x8464	matriz[0]	ponteiro
0x8468	matriz[1]	ponteiro
0x846C	matriz[2]	ponteiro

matriz será um ponteiro de ponteiros (**), no caso, um ponteiro para um vetor de 3 ponteiros, onde cada item do vetor (matriz[0], matriz[1], matriz[2]) será um ponteiro para uma linha de **matriz**.

matriz recebe o ponteiro (0x8464), que conterá outros ponteiros para as linhas da matriz (matriz[0], matriz[1] e matriz[2]) que conterão vetores de inteiros...

O endereço 0x8464 é o início do bloco alocado pelo malloc().

- 2º Passo: alocar as colunas para cada linha da matriz

Cada linha da matriz é alocada dinamicamente com um vetor de tamanho 3 para acomodar as colunas de cada linha

Cada linha terá um bloco diferente na memória, não sendo obrigatoriamente sequencial (quando a dimensão era fixa na declaração da matriz)

```
int **matriz = NULL;
int NL=3, NC=3;
matriz = (int**) malloc(NL*sizeof(int *));
```

```
for(int i=0;i<NL;i++) { //para cada linha
    matriz[i]= (int *) malloc(NC*sizeof(int));
}
```

```
for(int i=0;i<NL;i++)
    for(int j=0;j<NC;j++)
        matriz[i][j] = i*NC + j + 1;
```

Endereço	Var	Conteúdo
0x4560	int **matriz	0x8464
.....	livre	
0x8464	matriz[0]	0x8500
0x8468	matriz[1]	0x87AC
0x846C	matriz[2]	0x8D88
.....	livre	
0x8500	matriz[0][0]	1
0x8504	matriz[0][1]	2
0x8508	matriz[0][2]	3
.....	livre	
0x87AC	matriz[1][0]	4
0x87B0	matriz[1][1]	5
0x87B3	matriz[1][2]	6
.....	livre	
0x8D88	novo[2][0]	7
0x8D8B	novo[2][1]	8
0x8D8F	novo[2][2]	9

```
int main() {
    int **matriz = NULL;
    int NL=3, NC=3;

    matriz = (int **) malloc(NL * sizeof(int *));
    for(int i=0;i<NL;i++)
        matriz[i]= (int *) malloc(NC*sizeof(int));
```

Aloca a matriz na memória

```
for(int i=0;i<NL;i++) {
    for(int j=0;j<NC;j++) {
        matriz[i][j] = i*NC + j + 1; //i*numero_colunas (armazenar de 1....9)
        printf("%i\t", matriz[i][j]);
    }
}
```

Para desalocar uma matriz,
deve-se fazer o processo inverso:
desalocar as colunas para depois
desalocar as linhas

```
for(int i=0;i<NL;i++)
    free(matriz[i]); //liberar bloco de memória das colunas de cada linha

free(matriz); //liberar bloco de memória dos ponteiros das linhas
matriz=NULL; //evita lixo de memória
return 0;
}
```

- Alocação de array de structs

```
typedef struct {
    float notas[4];
    int idade;
} tipo_cad;

int main( ) {
    tipo_cad *cad = NULL;
    cad = (tipo_cad *) malloc(5 * sizeof(tipo_cad)); //exemplo alocar 5 cadastros do tipo_cad
    for(int i=0 ; i<5 ; i++) {
        for(int n=0 ; n<4 ; n++){
            printf("Digite cad[%i].notas[%i] ", i , n );
            scanf("%f", &cad[i].notas[n] );
        }
        printf("Digite cad[%i].idade ", i );
        scanf("%i", &cad[i].idade );
    }
    for(i=0 ; i<5 ; i++) {
        printf("cad[%i].notas { ", i);
        for(int n=0 ; n<4 ; n++)
            printf("%f ", cad[i].notas[n] );
        printf("} cad[%i].idade { %i }\\n ", cad[i].idade );
    }
    free(cad); //liberar memória
    cad=NULL; //evita lixo de memória
    return 0;
}
```

- 1) Faça uma função que alogue dinamicamente um vetor de inteiros, onde a dimensão do vetor é informado por parâmetro. O vetor deve ter os conteúdos em 0. Retorne para o main() o vetor alocado.
- 2) Faça uma função que alogue dinamicamente uma matriz de reais, onde as dimensões da matriz (nº de linhas e colunas) são passados por parâmetro. A matriz gerada deve ser preenchida com números 1. A função deve retornar essa matriz alocada para a função main().
- 3) Faça uma função que receba uma matriz por parâmetro bem como suas dimensões e desaloque ela da memória.
- 4) Escreva uma função que receba como parâmetro duas matrizes, A e B, e seus tamanhos. A função deve retornar o ponteiro para uma matriz C, em que C é o produto da multiplicação da matriz A pela matriz B. Se a multiplicação das matrizes não for possível, retorne um ponteiro nulo.

5) Faça um programa que gerencie cadastros de clientes. Cada cliente terá nome, idade, endereço e cidade. Deve-se organizar as cidades em outro cadastro, sendo que o cadastro de cliente deve aproveitar o cadastro de cidades (usar ponteiro).

- O programa conterá apenas as cidades abaixo em um vetor:

Apucarana, PR
Arapongas, PR
Sao Paulo, SP
Joinville, SC
Porto Alegre, RS
Rio de Janeiro, RJ

- O programa ainda deve solicitar ao usuário quantos cadastros existirão para alocar um vetor.
- Procure criar um menu para gerenciar o cadastro

```
typedef struct {  
    char cidade[21];  
    char estado[4];  
}tipo_cidade;
```

```
typedef struct {  
    char nome[31];  
    int idade;  
    char end[31];  
    tipo_cidade *city;  
}tipo_cad;
```