

Linguagem C: Ponteiros - parte 2

Delano Beder

*Baseado no livro: Linguagem C Completa e Descomplicada, de André Backes

Array de ponteiros

- Assim como os arrays de outros tipos, podemos definir array de ponteiros

```
tipo_dado *nome_array[tamanho];
```

- Exemplo:

```
int *vp[10]; // cria um array de ponteiros para inteiros
```

```
// Cada posição de um array de ponteiros pode armazenar o endereço de  
// uma variável ou o endereço da posição inicial de um outro array.
```

```
int main() {  
  
    int *pvet[2];  
    int x = 10, y[2] = {20, 30};  
    pvet[0] = &x;  
    pvet[1] = y;  
  
    // Imprime os endereços das variáveis  
  
    printf("Endereco pvet[0]: %p\n", pvet[0]);  
    printf("Endereco pvet[1]: %p\n", pvet[1]);  
  
    // Imprime o conteúdo de uma variável  
    printf("Conteúdo apontado por pvet[0]: %d\n", *pvet[0]);  
  
    // Imprime uma posição do vetor  
    printf("Conteúdo em pvet[1][1]: %d\n", pvet[1][1]);  
  
    return 0;  
}
```

Ponteiro para ponteiro

- 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.

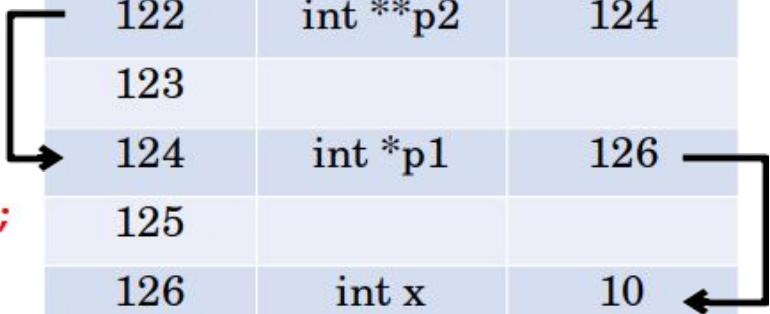
Ponteiro para ponteiro

- *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.

Ponteiro 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("Conteúdo em *p2: %p\n", *p2);
//Conteúdo do endereço do endereço
printf("Conteúdo 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		



Ponteiro 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;
```

Ponteiro 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		

```
graph TD
    p3[120: char ***p3] --> p2[122: char **p2]
    p2 --> p1[124: char *p1]
    p1 --> letra[126: char letra]
```


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 vai **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;  
}
```

Linguagem C: Alocação Dinâmica

Definição

- Sempre que escrevemos um programa, é preciso reservar espaço para as informações que serão processadas.
- Para isso utilizamos as variáveis
 - Uma variável é uma posição de memória que armazena uma informação que pode ser modificada pelo programa.
 - Ela deve ser definida antes de ser usada.

Definição

- Infelizmente, nem sempre é possível saber, em tempo de execução, o quanto de memória um programa irá precisar.
- Exemplo
 - Faça um programa para cadastrar o preço de **N** produtos, em que **N** é um valor informado pelo usuário

```
int N, i;  
double produtos[N];
```

Errado! Não sabemos o valor de **N**

```
int N, i;  
  
scanf("%d", &N)  
  
double produtos[N];
```

Funciona, mas não é o mais indicado

Arrays - criação (alocação de espaço)

Tamanho Fixo (tempo compilação)	Tamanho Variado (tempo execução)	Tamanho Variado (alocação dinâmica - tempo execução)
<code>int vetor[10]</code>	<code>int vetor[N]</code> (versão C99)	<code>int *vetor = (int*) malloc(N * sizeof(int))</code>
Tamanho não pode ser alterado depois de criado	Tamanho não pode ser alterado depois de criado	Tamanho pode ser alterado depois de criado
Se o tamanho for grande, a compilação pode apresentar <i>warnings</i> .	Se não houver memória suficiente para a alocação, o programa encerra com <i>segmentation fault</i> .	É possível verificar se a alocação foi realizada com sucesso e tomar providências no caso de falha na alocação.
A desalocação (liberação) da memória é automática	A desalocação (liberação) da memória é automática	A desalocação não é automática. Fica a critério do programador (função free) liberar a memória de forma explícita.

Definição

- A *alocação dinâmica* permite ao programador criar “variáveis” em tempo de execução, ou seja, alocar memória para novas variáveis quando o programa está sendo executado, e não apenas quando se está escrevendo o programa.
 - Quantidade de memória é alocada sob demanda
 - Menos desperdício de memória
 - Espaço é reservado até liberação explícita
 - Depois de liberado, estará disponível para outros usos
 - **Obs:** espaço alocado e não liberado explicitamente não é automaticamente liberado ao final da execução



A alocação dinâmica consiste em requisitar um espaço de memória ao computador, em tempo de execução, o qual devolve para o programa o endereço do início desse espaço alocado usando um ponteiro.

Memória		
#	var	conteúdo
119		
120		
121	int *n	NULL
122		
123		
124		
125		
126		
127		
128		
129		

Alocando 5
posições de
memória em int * n



Memória		
#	var	conteúdo
119		
120		
121	int *n	#123
122		
123	n[0]	11
124	n[1]	25
125	n[2]	32
126	n[3]	44
127	n[4]	52
128		
129		

Alocação Dinâmica

- A linguagem C ANSI usa apenas 4 funções para o sistema de alocação dinâmica, disponíveis na **stdlib.h**:
 - malloc
 - calloc
 - realloc
 - free

Alocação Dinâmica - malloc

- **malloc**

- A função malloc() serve para alocar memória e tem o seguinte protótipo:

```
void *malloc (unsigned int num);
```

- Funcionalidade

- Dado o número de bytes que queremos alocar (**num**), ela aloca na memória e retorna um ponteiro **void*** para o primeiro byte alocado.

Alocação Dinâmica - malloc

- O ponteiro **void*** pode ser atribuído a qualquer tipo de ponteiro via *type cast*. Se não houver memória suficiente para alocar a memória requisitada a função malloc() retorna um ponteiro nulo.

```
void *malloc (unsigned int num);
```

Alocação Dinâmica - malloc

- Alocar 1000 bytes de memória livre.

```
char *p;  
p = (char *) malloc(1000);
```

- Alocar espaço para 50 inteiros:

```
int *p;  
p = (int *) malloc(50*sizeof(int));
```

Alocação Dinâmica - malloc

- Operador **sizeof()**

- Retorna o número de **bytes** de um dado tipo de dado. Ex.: int, float, char, struct...

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

```
int main(){  
  
    printf("char: %d\n", sizeof(char)); // 1  
    printf("int: %d\n", sizeof(int)); // 4  
    printf("float: %d\n", sizeof(float)); // 4  
    printf("ponto: %d\n", sizeof(struct ponto)); // 8  
  
    return 0;  
}
```

Alocação Dinâmica - malloc

- Operador **sizeof()**

- No exemplo anterior,

```
p = (int *) malloc(50*sizeof(int));
```

- **sizeof(int)** retorna 4

- número de bytes necessário para armazenar um valor do tipo **int** na memória

- Portanto, são alocados 200 bytes ($50 * 4$)

- 200 bytes = 50 posições do tipo **int** na memória

Alocação Dinâmica - malloc

- Se não houver memória suficiente para alocar a memória requisitada, a função **malloc()** retorna um ponteiro nulo

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

int main() {

    int i;
    int *v = (int *) malloc(5 * sizeof(int));

    if (v == NULL) {
        printf("Erro: Memoria Insuficiente\n");
        exit(1);
    }

    for (i = 0; i < 5; i++) {
        scanf("%d", &v[i]);
    }

    for (i = 0; i < 5; i++) {
        printf("%d ", v[i]);
    }
    printf("\n");

    free(v);

    return 0;
}
```


Alocação Dinâmica - calloc

- **calloc**

- A função `calloc()` também serve para alocar memória, mas possui um protótipo um pouco diferente:

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

- **Funcionalidade**

- A função `calloc()` faz o mesmo que a função `malloc()`. A diferença é que agora passamos a quantidade de posições a serem alocadas e o tamanho do tipo de dado alocado como parâmetros distintos da função.

Alocação Dinâmica - calloc

- Exemplo da função **calloc**

```
int main() {  
    //alocação com malloc  
    int *p;  
    p = (int *) malloc(50*sizeof(int));  
    if(p == NULL) {  
        printf("Erro: Memoria Insuficiente!\n");  
    }  
    //alocação com calloc  
    int *p1;  
    p1 = (int *) calloc(50, sizeof(int));  
    if(p1 == NULL) {  
        printf("Erro: Memoria Insuficiente!\n");  
    }  
  
    return 0;  
}
```

Alocação Dinâmica - realloc

- **realloc**

- A função `realloc()` serve para realocar memória e tem o seguinte protótipo:

```
void *realloc (void *ptr, unsigned int num);
```

- Funcionalidade

- A função modifica o tamanho da memória previamente alocada e apontada por ***ptr** para aquele especificado por **num**.
- O valor de **num** pode ser maior ou menor que o original.

Alocação Dinâmica - realloc

- **realloc**

- Um ponteiro para o bloco é devolvido porque `realloc()` pode precisar mover o bloco para aumentar seu tamanho.
- Se isso ocorrer, o conteúdo do bloco antigo é copiado para o novo bloco, e nenhuma informação é perdida.

```
int main() {  
  
    int i;  
    int *v = (int *) malloc(5 * sizeof(int));  
    if (v == NULL) {  
        printf("Erro: Memoria Insuficiente\n");  
        exit(1);  
    }  
    for (i = 0; i < 5; i++) {  
        v[i] = i + 1;  
    }  
    for (i = 0; i < 5; i++) {  
        printf("%d\n", v[i]);  
    }  
    // Diminui o tamanho do vetor  
    v = (int *) realloc(v, 3 * sizeof(int));  
    for (i = 0; i < 3; i++) {  
        printf("%d\n", v[i]);  
    }  
    // Aumenta o tamanho do vetor  
    v = (int *) realloc(v, 10 * sizeof(int));  
    for (i = 0; i < 10; i++) {  
        printf("%d\n", v[i]);  
    }  
    free(v);  
    return 0;  
}
```

Alocação Dinâmica - realloc

- Observações sobre realloc()
 - Se ***ptr** for nulo, aloca **num** bytes e devolve um ponteiro (igual malloc);
 - se **num** é zero, a memória apontada por ***ptr** é liberada (igual free).
 - Se não houver memória suficiente para a alocação, um ponteiro nulo é devolvido e o bloco original é deixado inalterado.

Alocação Dinâmica - free

- **free**

- Diferente das variáveis definidas durante a escrita do programa, as variáveis alocadas dinamicamente não são liberadas automaticamente pelo programa.
- Quando alocamos memória dinamicamente é necessário que nós a liberemos quando ela não for mais necessária.
- Para isto existe a função **free()** cujo protótipo é:

```
void free(void *p);
```

Alocação Dinâmica - free

- Assim, para liberar a memória, basta passar como parâmetro para a função `free()` o ponteiro que aponta para o início da memória a ser desalocada.
- Como o programa sabe quantos bytes devem ser liberados?
 - Quando se aloca a memória, o programa guarda o número de bytes alocados numa "tabela de alocação" interna.

Alocação Dinâmica

- Exemplo da função **free()**

```
int main() {  
    int *p, i;  
    p = (int *) malloc(50*sizeof(int));  
    if(p == NULL) {  
        printf("Erro: Memoria Insuficiente!\n");  
  
        exit(1);  
    }  
    for (i = 0; i < 50; i++) {  
        p[i] = i+1;  
    }  
    for (i = 0; i < 50; i++) {  
        printf("%d\n", p[i]);  
    }  
    //libera a memória alocada  
    free(p);  
  
    return 0;  
}
```


Alocação de arrays

- Para armazenar um array o compilador C calcula o tamanho, em bytes, necessário e reserva posições sequenciais na memória
 - Note que isso é muito parecido com alocação dinâmica
- Como já vimos, existe uma ligação muito forte entre ponteiros e arrays.
 - O nome do array é apenas um ponteiro que aponta para o primeiro elemento do array.

Alocação de arrays

- Ao alocarmos memória estamos, na verdade, alocando um array.

```
int *p;  
int i, N = 100;  
  
p = (int *) malloc(N*sizeof(int));  
  
for (i = 0; i < N; i++)  
    scanf("%d", &p[i]);
```



Alocação de arrays

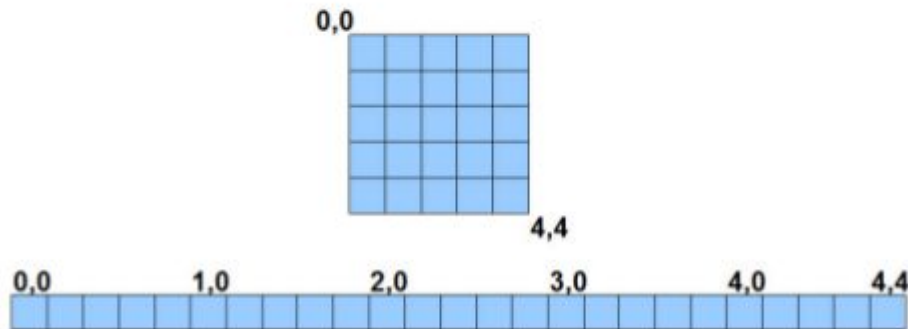
- Note, no entanto, que o array alocado possui apenas uma dimensão
- Para liberá-lo da memória, basta chamar a função `free()` ao final do programa:

```
int *p;  
int i, N = 100;  
  
p = (int *) malloc(N*sizeof(int));  
  
for (i = 0; i < N; i++)  
    scanf("%d", &p[i]);  
  
free(p);
```

Alocação de matrizes

- Para alocarmos arrays com mais de uma dimensão, temos duas soluções:
- **Solução 1: array unidimensional**

```
int mat[5][5];
```



Solução 1: Array unidimensional

— — —

Podemos alocar um array de uma única dimensão e tratá-lo como se fosse uma matriz (2 dimensões)

```
int main() {
    int *M;
    int i, j, Nlinhas = 2, Ncolunas = 2;
    M = (int *) malloc(Nlinhas * Ncolunas * sizeof(int));
    for (i = 0; i < Nlinhas; i++) {
        for (j = 0; j < Ncolunas; j++) {
            M[i * Ncolunas + j] = i + j;
        }
    }
    for (i = 0; i < Nlinhas; i++) {
        for (j = 0; j < Ncolunas; j++) {
            printf("%d ", M[i * Ncolunas + j]);
        }
        printf("\n");
    }
    free(M);
    return 0;
}
```

Desvantagem:

- uso de apenas um colchete para acessar todos os elementos (array de uma dimensão)
- Deve-se calcular o deslocamento no array para simular a segunda dimensão

`M[i * Ncolunas + j]`

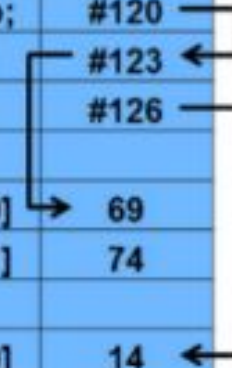
Alocação de matrizes

- Solução 2: “ponteiro para ponteiro”.
 - Em um ponteiro para ponteiro, cada nível do ponteiro permite criar uma nova dimensão no array.

Alocação de matrizes

```
int main() {
    int **p;
    int i, j, Nlinhas = 2, Ncolunas = 2;
    p = (int **) malloc(Nlinhas * sizeof(int*));
    for (i = 0; i < Nlinhas; i++) {
        p[i] = (int *) malloc(Ncolunas * sizeof(int));
        for (j = 0; j < Ncolunas; j++) {
            scanf("%d", &p[i][j]);
        }
    }
    for (i = 0; i < Nlinhas; i++) {
        for (j = 0; j < Ncolunas; j++) {
            printf("%d ", p[i][j]);
        }
        printf("\n");
    }
    for (i = 0; i < Nlinhas; i++) {
        free(p[i]);
    }
    free(p);
    return 0;
}
```

Memória		
#	var	conteúdo
119	int **p;	#120
120	p[0]	#123
121	p[1]	#126
122		
123	p[0][0]	69
124	p[0][1]	74
125		
126	p[1][0]	14
127	p[1][1]	31
128		

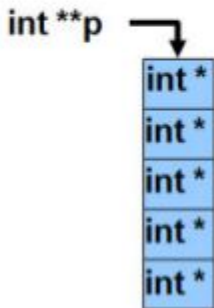


Alocação de matrizes

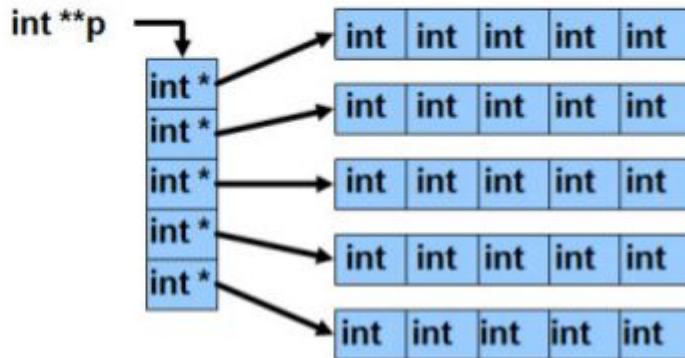
- Em um ponteiro para ponteiro, cada nível do ponteiro permite criar uma nova dimensão no array.

```
p = (int **) malloc(Nlinhas * sizeof(int*));  
for (i = 0; i < Nlinhas; i++) {  
    p[i] = (int *) malloc(Ncolunas * sizeof(int));  
}
```

1º malloc
Cria as linhas da matriz



2º malloc
Cria as colunas da matriz



Alocação de matrizes

- Diferente dos arrays de uma dimensão, para liberar um array com mais de uma dimensão da memória, é preciso liberar a memória alocada em cada uma de suas dimensões, na ordem inversa da que foi alocada

Alocação de arrays

```
int main() {  
    int **p;  
    int i, j, Nlinhas = 2, Ncolunas = 2;  
    p = (int **) malloc(Nlinhas * sizeof(int*));  
    for (i = 0; i < Nlinhas; i++) {  
        p[i] = (int *) malloc(Ncolunas * sizeof(int));  
        for (j = 0; j < Ncolunas; j++) {  
            scanf("%d", &p[i][j]);  
        }  
    }  
    for (i = 0; i < Nlinhas; i++) {  
        for (j = 0; j < Ncolunas; j++) {  
            printf("%d ", p[i][j]);  
        }  
        printf("\n");  
    }  
    for (i = 0; i < Nlinhas; i++) {  
        free(p[i]);  
    }  
    free(p);  
    return 0;  
}
```

Alocação de struct

- Assim como os tipos básicos, também é possível fazer a alocação dinâmica de estruturas.
- As regras são exatamente as mesmas para a alocação de uma **struct**.
- Podemos fazer a alocação de
 - uma única **struct**
 - um array de **structs**

Alocação de struct

- Para alocar uma única **struct**
 - Um ponteiro para **struct** receberá o **malloc()**
 - Utilizamos o **operador seta** para acessar o conteúdo
 - Usamos **free()** para liberar a memória alocada

```
struct cadastro{
    char nome[50];
    int idade;
};

int main() {
    struct cadastro *cad = (struct cadastro*) malloc(sizeof(struct cadastro));
    strcpy(cad->nome, "Maria");
    cad->idade = 30;

    free(cad);

    return 0;
}
```

Alocação de struct

- Para alocar um array de **struct**
 - Um ponteiro para **struct** receberá o **malloc()**
 - Utilizamos os **colchetes []** para acessar o conteúdo
 - Usamos **free()** para liberar a memória alocada

Alocação de struct

```
struct cadastro{
    char nome[50];
    int idade;
};

int main(){
    struct cadastro *vcad = (struct cadastro*) malloc(10*sizeof(struct cadastro));

    strcpy(vcad[0].nome, "Maria");
    vcad[0].idade = 30;

    strcpy(vcad[1].nome, "Cecilia");
    vcad[1].idade = 10;

    strcpy(vcad[2].nome, "Ana");
    vcad[2].idade = 10;

    free(vcad);

    return 0;
}
```

Exercícios

Lista no AVA