



LINGUAGEM C: ALOCAÇÃO DINÂMICA

Leandro Henrique Furtado Pinto Silva

CRÉDITOS

O material dessa aula foi gentilmente cedido pelo Professor André Ricardo Backes e, por esse motivo, o crédito é dele.



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



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, ou seja, quando o programa precisa
 - Menos desperdício de memória
 - Espaço é reservado até liberação explícita
 - Depois de liberado, estará disponibilizado para outros usos e não pode mais ser acessado
 - Espaço alocado e não liberado explicitamente é automaticamente liberado ao final da execução



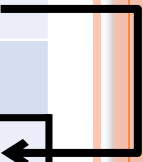
ALOCANDO MEMÓRIA

| Memória | | |
|---------|----------|----------|
| posição | variável | conteúdo |
| 119 | | |
| 120 | | |
| 121 | int *p | NULL |
| 122 | | |
| 123 | | |
| 124 | | |
| 125 | | |
| 126 | | |
| 127 | | |
| 128 | | |

**Alocando 5
posições de
memória em int *p**



| Memória | | |
|---------|----------|----------|
| posição | variável | conteúdo |
| 119 | | |
| 120 | | |
| 121 | int *p | 123 |
| 122 | | |
| 123 | p[0] | 11 |
| 124 | p[1] | 25 |
| 125 | p[2] | 32 |
| 126 | p[3] | 44 |
| 127 | p[4] | 52 |
| 128 | | |



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 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

```
int main() {
    int *p;
    p = (int *) malloc(5*sizeof(int));
    if(p == NULL) {
        printf("Erro: Memoria Insuficiente!\n");
        system("pause");
        exit(1);
    }
    int i;
    for (i=0; i<5; i++){
        printf("Digite o valor da posicao %d: ",i);
        scanf("%d",&p[i]);
    }

    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

- Basicamente, 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 *p = malloc(5*sizeof(int));
    for (i = 0; i < 5; i++) {
        p[i] = i+1;
    }
    for (i = 0; i < 5; i++) {
        printf("%d\n", p[i]);
    }
    printf("\n");
    //Diminui o tamanho do array
    p = realloc(p, 3*sizeof(int));
    for (i = 0; i < 3; i++) {
        printf("%d\n", p[i]);
    }
    printf("\n");
    //Aumenta o tamanho do array
    p = realloc(p, 10*sizeof(int));
    for (i = 0; i < 10; i++) {
        printf("%d\n", p[i]);
    }

    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 - FREE

- Exemplo da função **free()**

```
int main() {
    int *p, i;
    p = (int *) malloc(50*sizeof(int));
    if(p == NULL) {
        printf("Erro: Memoria Insuficiente!\n");
        system("pause");
        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

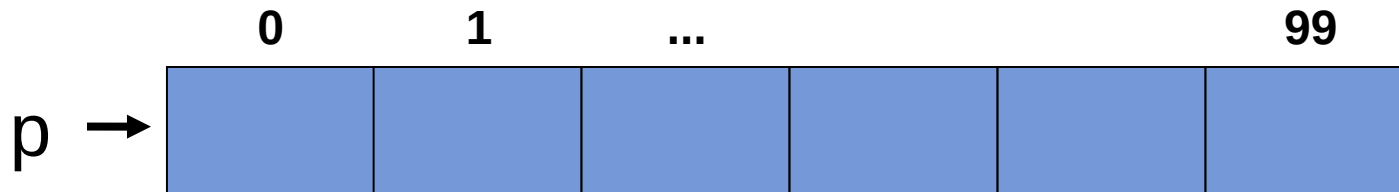
- Para armazenar um array o compilador C calcula o tamanho, em bytes, necessário e reserva posições sequenciais na memória
 - Note que isso é muito parecido com alocação dinâmica
- Existe uma ligação muito forte entre ponteiros e arrays.
 - O nome do array é apenas um ponteiro que aponta para o primeiro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

- Para alocarmos arrays com mais de uma dimensão, utilizamos o conceito de “ponteiro para ponteiro”.
 - Ex.: `char ***p3;`
- Para cada nível do ponteiro, fazemos a alocação de uma dimensão do array.



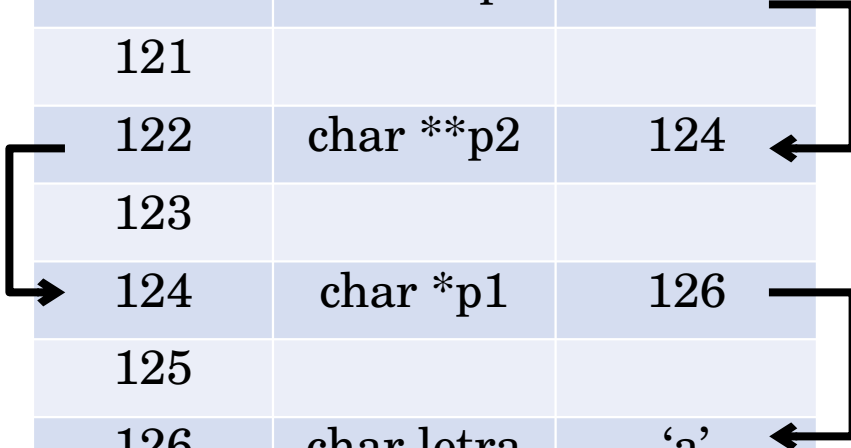
ALOCAÇÃO DE ARRAYS

- 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 | | |



ALOCAÇÃO DE ARRAYS

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

```
int **p; //2 "*" = 2 níveis = 2 dimensões
int i, j, N = 2;
p = (int**) malloc(N*sizeof(int*));

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

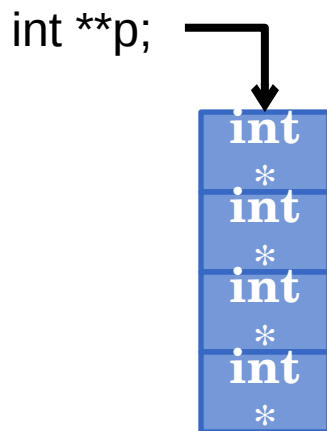
| Memória | | |
|---------|----------|----------|
| posição | variável | conteúdo |
| 119 | int **p | 120 |
| 120 | p[0] | 123 |
| 121 | p[1] | 126 |
| 122 | | |
| 123 | p[0][0] | 69 |
| 124 | p[0][1] | 74 |
| 125 | | |
| 126 | p[1][0] | 14 |
| 127 | p[1][1] | 31 |
| 128 | | |

ALOCAÇÃO DE ARRAYS

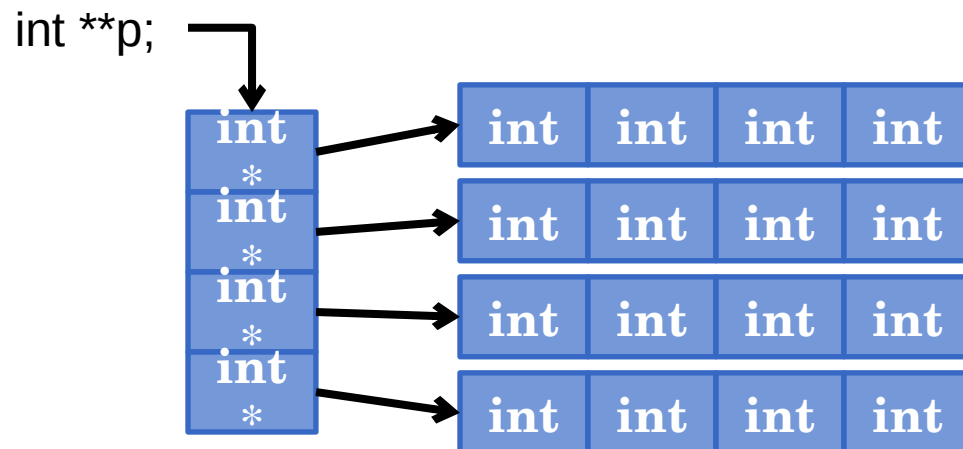
- Em um ponteiro para ponteiro, cada nível do ponteiro permite criar uma nova dimensão no array.

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

1º malloc:
cria as linhas



2º malloc:
cria as colunas



ALOCAÇÃO DE ARRAYS

- 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 **p; //2 "*" = 2 níveis = 2 dimensões
int i, j, N = 2;
p = (int**) malloc(N*sizeof(int*));

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

```
for (i = 0; i < N; i++)
    free(p[i]);
free(p);
```



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 uma única **struct**
 - Um ponteiro para **struct** receberá o **malloc()**
 - Utilizamos os **colchetes []** para acessar o conteúdo
 - Usamos **free()** para liberar a memória alocada

```
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;
}
```



MATERIAL COMPLEMENTAR

○ Vídeo Aulas

- Aula 60: Alocação Dinâmica pt.1 – Introdução:
● youtu.be/ErOmueylikM
- Aula 61: Alocação Dinâmica pt.2 – Sizeof:
● youtu.be/p2ihD9uDZs4
- Aula 62: Alocação Dinâmica pt.3 – Malloc:
● youtu.be/iU9CL5d-P5U
- Aula 63: Alocação Dinâmica pt.4 – Calloc:
● youtu.be/34uZMXVQd08
- Aula 64: Alocação Dinâmica pt.5 – Realloc:
● youtu.be/vEMTGkANXM4
- Aula 65: Alocação Dinâmica pt.6 – Alocação de Matrizes:
● youtu.be/W4vbwEJn11U

