MCTA028-15: Programação Estruturada

Aula 9: Alocação Dinâmica (Primeira Parte)

Wagner Tanaka Botelho
wagner.tanaka@ufabc.edu.br / wagtanaka@gmail.com
Universidade Federal do ABC (UFABC)
Centro de Matemática, Computação e Cognição (CMCC)

Introdução

Introdução

- Alocação estática:
  - NÃO é possível alterar o tamanho do espaço durante a execução do programa;
  - É utilizada quando o programador sabe a quantidade de memória necessária;
  - Até agora, a quantidade de memória declarada e reservada é FIXA.

```
char a;
```
→ Espaço reservado para um elemento do tipo `char` (ocupa 1 *byte* na memória.

```
int vetor[10];
```
→ Espaço reservado para dez valores do tipo `int` (ocupa 4 *bytes* na memória). Portanto, 4x10 = 40 *bytes*.

```
double matriz[3][3];
```
→ Espaço reservado para 6 valores do tipo `double` (ocupa 8 *bytes* na memória). Portanto, 3x3x8 = 72 *bytes*.

Introdução

- Alocação dinâmica:
  - A Linguagem C permite reservar dinamicamente, e em tempo de execução, blocos de memórias utilizando ponteiros, evitando, assim, desperdício de memória;
  - Utilizada quando NÃO se sabe ao certo quanto de memória será necessário para armazenar os dados com que se quer trabalhar.

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

Introdução

- **EXEMPLO!!** Desenvolver um algoritmo para processar os valores dos salários dos funcionários de uma pequena empresa:
  - Uma solução é declarar um *array* do tipo **`float`** bem GRANDE, por exemplo, 1000 posições:

```
float salarios[1000];
```

- A declaração de um *array* muito GRANDE possui **DOIS** problemas. Se a empresa tiver:
  - **MENOS** de 1000 funcionários, o algoritmo está desperdiçando memória;
  - **MAIS** de 1000 funcionários, o tamanho do *array* será insuficiente para lidar com os dados de todos os funcionários.

Primeira Etapa

Ponteiro p apontando para NULL

MEMÓRIA

#	Var	Conteúdo
119		
121	`int *p`	NULL
122		
123		
124		
125		
126		
127		
128		

Segunda Etapa

Requisitando para o computador CINCO posições inteiras de memória.

Terceira Etapa

Computador alocando 5 posições de memória (de #123 até #127) em int *p

Quarta Etapa

O ponteiro p passa então a se comportar como se fosse um *array* de tamanho 5, ou seja, `int p[5]`.

MEMÓRIA

#	Var	Conteúdo
119		
120		
121	`int *p`	#123
122		
123	`p[0]`	12
124	`p[1]`	22
125	`p[2]`	33
126	`p[3]`	44
127	`p[4]`	55
128		

Funções para Alocação de Memória

Funções para Alocação de Memória

- A Linguagem C usa apenas QUATRO FUNÇÕES para o sistema de alocação dinâmica disponíveis na biblioteca **`stdlib.h`**:
  - **`malloc`**;
  - **`calloc`**;
  - **`realloc`**;
  - **`free`**.
- Além dessas funções, existe a função **`sizeof`**:
  - Auxilia as demais funções no processo de alocação de memória.

Função `sizeof()`

Função `sizeof()`

- Alocar memória para um elemento do tipo `int` (4 *bytes*) é **DIFERENTE** de alocar para o tipo `float` (8 *bytes*);
- Função `sizeof()`:
  - Usada para saber o NÚMERO de *bytes* necessários para alocar um único elemento de determinado tipo de dado;
  - Pode ser usada de DUAS formas:

```
sizeof nome_da_variável
```

```
sizeof (nome_do_tipo)
```

Ex_01.c

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

struct ponto{
    int x,y;
};

void main(){
    int x=0;
    double y=0;

    printf("Tamanho char: %d\n",sizeof(char));
    printf("Tamanho int: %d\n",sizeof(int));
    printf("Tamanho float: %d\n",sizeof(float));
    printf("Tamanho double: %d\n",sizeof(double));
    printf("Tamanho struct ponto: %d\n",sizeof(struct ponto));
    printf("Tamanho da variavel x: %d\n",sizeof x);
    printf("Tamanho da variavel y: %d\n",sizeof y);
}
```

D:\UFABC\Disciplinas\2021-2025\Q1\PE\

```
Tamanho char: 1
Tamanho int: 4
Tamanho float: 4
Tamanho double: 8
Tamanho struct ponto: 8
Tamanho da variavel x: 4
Tamanho da variavel y: 8
```

Função `malloc()`

Função `malloc()`

- Função **`malloc()`**:
  - Utilizada para ALOCAR MEMÓRIA durante a EXECUÇÃO do algoritmo;
  - Faz o PEDIDO de memória ao computador e RETORNA um **PONTEIRO** com o endereço do INÍCIO do espaço de memória alocado;
  - O seu protótipo é:

Recebe um PARÂMETRO de entrada

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

`num:` tamanho do espaço de memória a ser alocado

Retorna NULL no caso de ERRO! OU o PONTEIRO para a PRIMEIRA POSIÇÃO do *array* ALOCADO.

O **PONTEIRO** retornado é **GENÉRICO** (**`void*`**). Esse ponteiro pode ser atribuído a **QUALQUER** tipo de ponteiro via *type cast*.

No momento da alocação da memória, deve-se levar em conta o TAMANHO do dado alocado.

Ex_04.c

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   void main(){
4       char*p;
5
6       p = (char *) malloc(1000);
7
8       int *p;
9
10      p = (int *) malloc(1000);
11  }
```

➡ Alocando espaço para 1000 `chars`.

➡ Alocando espaço para 250 `ints`.

Bytes para **char**: um *array* de 1.000 posições de caracteres.

Bytes para **int**: um *array* de 250 posições de inteiros.

*Ex_02.c

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

void main(){
    int *p;

    p = (int *) malloc(5*sizeof(int));

    for (int i=0; i<5; i++){
        printf("Valor da posicao %d: ",i);
        scanf("%d",&p[i]);
    }
}
```

Alocando um *array* com **cinco** posições de inteiros: `5*sizeof(int)`.

O ponteiro `p` passa a ser tratado como um *array* `p[i]`

A função **`sizeof(int)`** retorna 4 *bytes* (quantidade de *bytes* do tipo `int`). Portanto, são alocados 20 *bytes* (5 * 4).

A função **`malloc()`** retorna um PONTEIRO GENÉRICO, o qual é convertido no tipo de ponteiro via *typecast*: (**`int*`**).

*Ex_03.c

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

void main(){
    int *p;

    p = (int *) malloc(5*sizeof(int));

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

    for (int i=0; i<5; i++){
        printf("Valor da posicao %d: ",i);
        scanf("%d",&p[i]);
    }
}
```

É importante TESTAR se foi possível fazer a alocação de memória. A função **`malloc()`** retorna um ponteiro **NULL** para indicar que **NÃO** há memória disponível no computador ou que ocorreu algum outro **ERRO** que impediu a memória de ser alocada.

Função `calloc()`

Função `calloc()`

- Função **`calloc()`**:
  - Assim como a função `malloc()`, a função `calloc()` também serve para **ALOCAR MEMÓRIA** durante a execução do algoritmo;
  - Faz o PEDIDO de memória ao computador e RETORNA um **PONTEIRO** com o endereço do início do espaço de memória alocado;
  - O seu protótipo é:

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

  - Recebe **DOIS** parâmetros de entrada:
    - **num**: quantidade de elementos no *array* a ser alocado;
    - **size**: tamanho de cada elemento do *array*.

Função `calloc()`

- Função **`calloc()`**:
  - Retorna:
    - **NULL**: caso de ERRO;
    - O **PONTEIRO** para a primeira posição do *array* alocado.
- Basicamente, a **`calloc()`** é **IGUAL** a função **`malloc()`**:
  - A diferença é que agora passamos os valores da quantidade de elementos alocados e do tipo de dado alocado como parâmetros distintos da função.

malloc() *versus* calloc()

Ex_05.c

```c
#include <stdio.h>
#include <stdlib.h>
void main(){
    int *p;
    p = (int *) malloc(50*sizeof(int));

    if(p == NULL){
        printf("Erro: Memoria Insuficiente!\n");
    }

    int *p1;
    p1 = (int *) calloc(50,sizeof(int));

    if(p1 == NULL){
        printf("Erro: Memoria Insuficiente!\n");
    }
}
```

➡ Alocando com `malloc()`.

➡ Alocando com `calloc()`.

+ A função **`malloc()`** multiplica o total de elementos do *array* pelo tamanho de cada elemento;
+ A função **`calloc()`** recebe os DOIS valores como parâmetros distintos.

malloc() *versus* calloc()

Ex_06.c

```c
#include <stdio.h>
#include <stdlib.h>
void main(){
    int *p, *p1;

    p = (int *) malloc(5*sizeof(int));

    p1 = (int *) calloc(5,sizeof(int));

    printf("calloc \t\t malloc\n");

    for (int i=0; i<5; i++){
        printf("p1[%d]=%d \t p[%d] = %d\n",i,p1[i],i,p[i]);
    }
}
```

D:\UFABC\Disciplinas\2021-2025\Q1\PE\Aula

```
calloc          malloc
p1[0]=0         p[0] = 11098416
p1[1]=0         p[1] = 0
p1[2]=0         p[2] = 11075920
p1[3]=0         p[3] = 0
p1[4]=0         p[4] = 1308640050
```

A função **`calloc()`** inicializa todos os **BITS** do espaço alocado com 0s.

Função `realloc()`

Função `realloc()`

- Função **`realloc()`**:
  - Serve para **ALOCAR MEMÓRIA** ou **REALOCAR BLOCOS** de memória previamente alocados pelas funções **`malloc()`**, **`calloc()`** ou **`realloc()`** ;
  - O seu protótipo é:

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

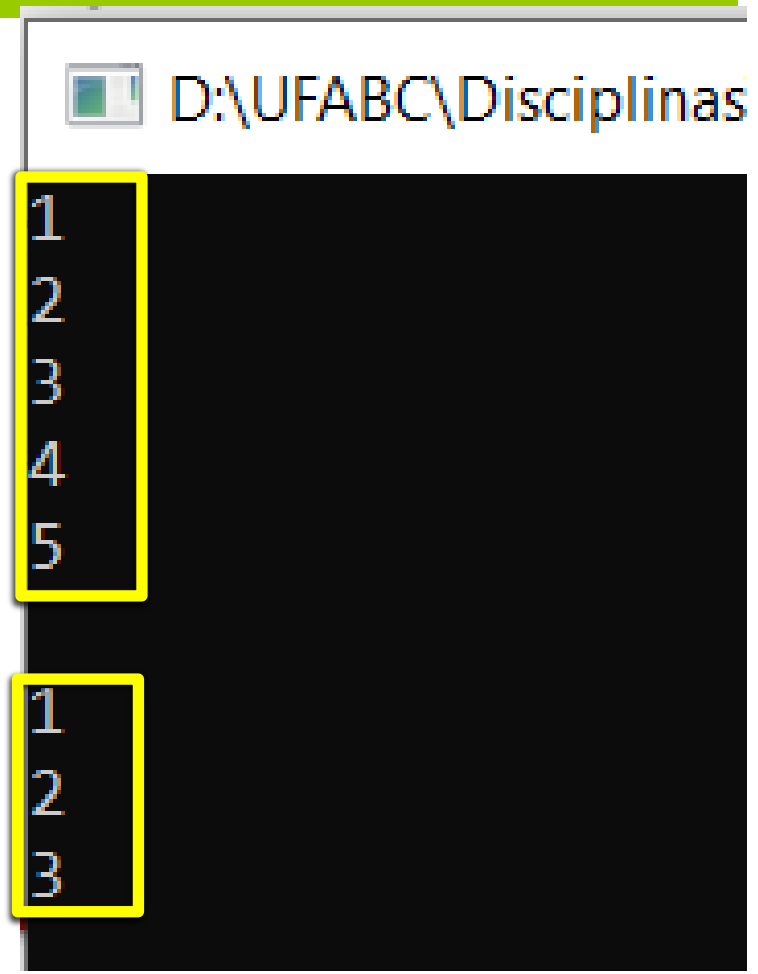
  - Recebe **DOIS** parâmetros de **ENTRADA**:
    - Um **PONTEIRO** para um bloco de memória PREVIAMENTE alocado;
    - num: tamanho em *bytes* do espaço de memória a ser alocado.
  - **Retorna**:
    - **NULL**: caso de **ERRO**;
    - O **PONTEIRO** para a primeira posição do *array* alocado/realocado.

Ex_07.c

```c
#include <stdio.h>
#include <stdlib.h>
void main(){
    int *p = NULL;

    p = malloc(5*sizeof(int));
    for (int i = 0; i < 5; i++){
        p[i] = i+1;
    }

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

    p = realloc(p,3*sizeof(int));

    if(p==NULL){
        printf("Memoria Insuficiente!\n");
    }
    else{
        for (int i = 0; i < 3; i++){
            printf("%d\n",p[i]);
        }
    }
}
```

Alocando com `malloc()`.

Diminuindo o tamanho do *array*.



p: ponteiro para o bloco de memória previamente alocado;

3*sizeof(int): tamanho em *bytes* do espaço de memória a ser alocado.

Ex_08.c

```c
#include <stdio.h>
#include <stdlib.h>
void main(){
    int *p = NULL;

    p = malloc(5*sizeof(int));
    for (int i = 0; i < 5; i++){
        p[i] = i+1;
    }

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

    p = realloc(p,10*sizeof(int));

    if(p==NULL){
        printf("Memoria Insuficiente!\n");
    }
    else{
        for (int i = 0; i < 10; i++){
            p[i] = i+1;
        }

        for (int i = 0; i < 10; i++){
            printf("%d\n",p[i]);
        }
    }
}
```

Alocando com `malloc()`.

Aumentando o tamanho do *array*.

D:\UFABC\Disciplinas\202

```
1
2
3
4
5

1
2
3
4
5
6
7
8
9
10
```

Função `free()`

Função `free()`

- **SEMPRE** que alocamos memória de forma dinâmica (`malloc()`, `calloc()` ou `realloc()`), é necessário **LIBERAR** essa memória quando ela **NÃO** for mais utilizada;
- **DESALOCAR**, ou **LIBERAR**, a memória previamente alocada faz com que ela se torne novamente disponível para futuras alocações;
- Para liberar um bloco de memória previamente alocada utiliza-se a função **`free()`**, cujo protótipo é:

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

- A função recebe apenas UM parâmetro de entrada:
  - O ponteiro (**`*p`**) para o **INÍCIO** do bloco de memória alocado.

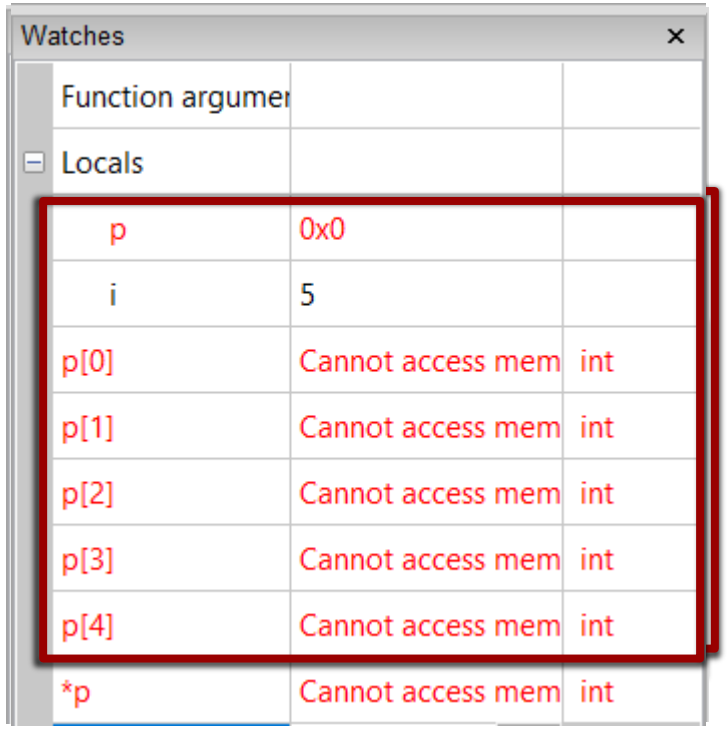
*Ex_09.c

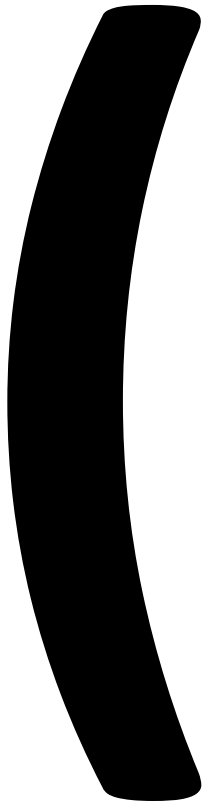
```c
#include <stdio.h>
#include <stdlib.h>
void main(){
    int *p,i;

    p = (int *) malloc(5*sizeof(int));

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

    for (i = 0; i < 5; i++){
        p[i] = i+1;
    }

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

    free(p);

    p=NULL;
}
```

Alocando um *array* com cinco posições de inteiros: `5*sizeof(int)`.

Liberando a memória.

Depois que liberar, não é recomendado deixar ponteiros soltos.



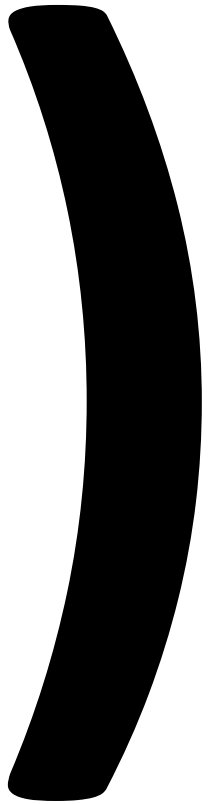


```c
void main(){
    int tam = 0;

    printf("Tamanho do vetor:");
    scanf("%i", &tam);

    int vetCol[tam];
}
```

Não é ALOCAÇÃO DINÂMICA! O ESTÁTICO é **IMUTÁVEL** desde a INICIALIZAÇÃO da aplicação. O `vetCol[]` **NÃO** pode ser alterado, depois que criado.

```c
void main(){
    int *p;

    p = (int *) malloc(5*sizeof(int));

    for (int i=0; i<5; i++){
        printf("Valor da posicao %d: ",i);
        scanf("%d",&p[i]);
    }
}
```

Você pode estar pensando que também **NÃO** é ALOCAÇÃO DINÂMICA, pois o programador está definindo o tamanho do vetor. Entretanto, o 5 **NÃO** é **IMUTÁVEL**, ou seja, pode-se **AUMENTAR/DIMINUIR** e até **LIBERAR** o espaço alocado. Portanto, é um exemplo de **ALOCAÇÃO DINÂMICA!!!**



Referências

- Slides do Prof. Luiz Rozante;
- SALES, André Barros de; AMVAME-NZE, Georges. Linguagem C: roteiro de experimentos para aulas práticas. 2016;
- BACKES, André. Linguagem C Completa e Descomplicada. Editora Campus. 2013;
- SCHILDT, Herbert. C Completo e Total. Makron Books. 1996;
- DAMAS, Luís. Linguagem C. LTC Editora. 1999;
- DEITEL, Paul e DEITEL, Harvey. C Como Programar. Pearson. 2011.