

# Algoritmos e Estruturas de Dados I

1º Período Engenharia da Computação

Prof. Edwaldo Soares Rodrigues

Email: [edwaldo.rodrigues@uemg.br](mailto:edwaldo.rodrigues@uemg.br)

Material adaptado do prof. André Backes

# Alocação Dinâmica de Memória

# 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;
- Outra diferença, é que a função calloc() inicializa as posições alocadas com o valor zero;

# 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: Memória Insuficiente!\n");  
    }  
    //alocação com calloc  
    int *p1;  
    p1 = (int *) calloc(50, sizeof(int));  
    if(p1 == NULL) {  
        printf("Erro: Memória 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 = (int*)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 da array
    p = (int*) realloc(p, 3*sizeof(int));
    for(i = 0; i < 3; i++){
        printf("%d\n", p[i]);
    }
    printf("\n");
    //Aumentando o tamanho da array
    p = (int *) 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: Memória 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;  
}
```

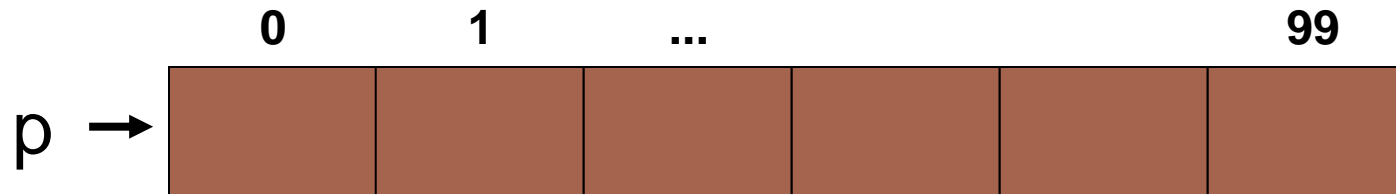
# Alocação de arrays

- 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;

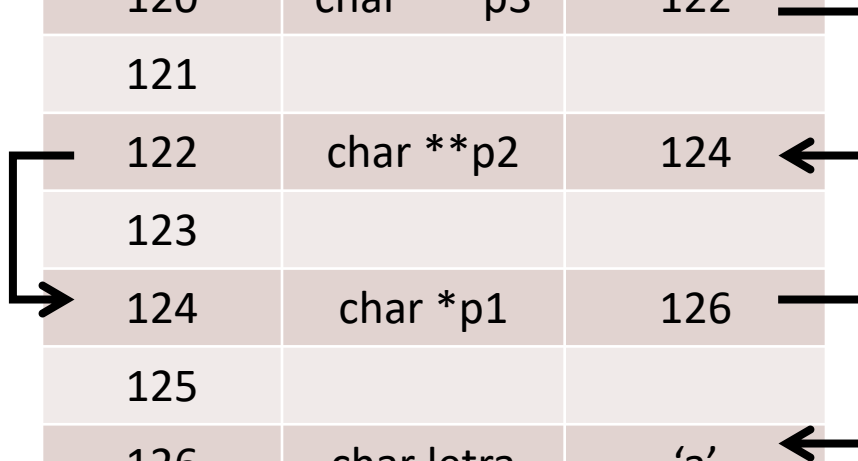
# Alocação de arrays

- 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		



The diagram illustrates the memory layout and pointer relationships. Arrows indicate the following: p3 (at address 120) points to p2 (at address 122); p2 (at address 122) points to p1 (at address 124); and p1 (at address 124) points to letra (at address 126). The variable letra at address 126 contains the character 'a'.

# 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]);
}
```

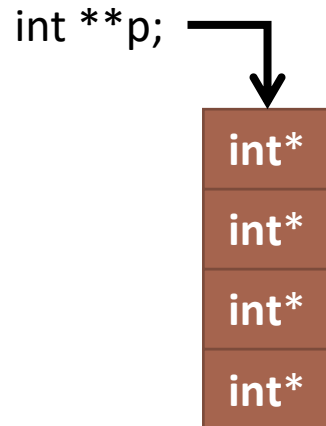
Memória		
posição	variável	conteúdo
119	int **p	120
120	p[0]	123
121	p[1]	126
122		
123	p[0][0]	69
124	p[0][1]	74
125		
126	p[1][0]	14
127	p[1][1]	31
128		

# Alocação de arrays

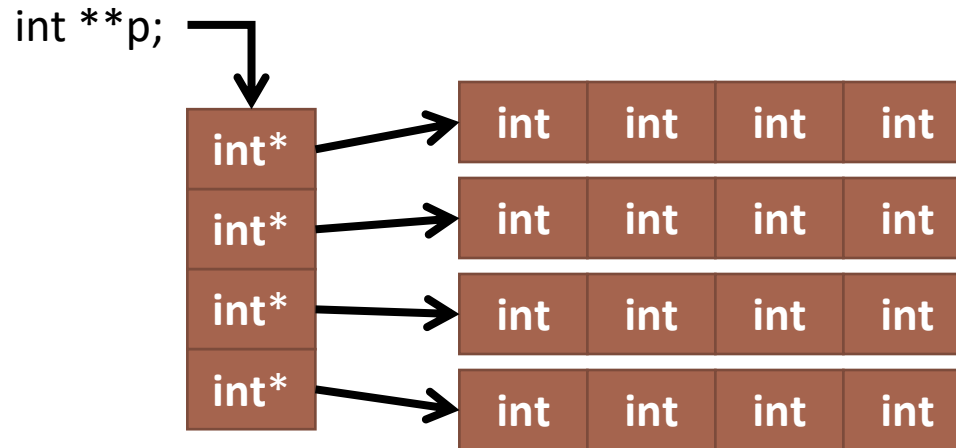
- Em um ponteiro para ponteiro, cada nível do ponteiro permite criar uma nova dimensão no array;

```
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;
}
```

# Constantes

- Frequentemente, utilizamos um determinado valor diversas vezes durante um programa. Para ilustrar, vamos imaginar um programa que trabalhe com um vetor ou uma matriz;
  - Na declaração de um vetor limitamos o seu tamanho a um determinado tamanho máximo;

# Constantes

- Na leitura e na escrita é comum lermos todos os elementos do vetor;
- Em geral percorremos o vetor uma ou mais vezes durante o nosso programa;
- Em todos esses casos, utilizamos um mesmo valor para limitar o laço for que percorrerá todo o vetor (ou matriz) e para a declaração;

# Constantes

- Uma forma de utilizarmos uma única representação para esse valor é declarar uma variável e atribuir um valor constante a ela, mas isso só resolve o problema dos laços, não o da declaração do vetor;

# Constantes

```
#include <stdio.h>

int linhas=3, colunas=3;
int matriz[3][3];
//int matriz[linhas][colunas]; // NAO FUNCIONA

int main(){

    int i,j;

    for(i=0; i < linhas; i++){
        for(j=0; j < colunas; j++){
            scanf("%d",&matriz[i][j]);
        }
    }

    return 0;
}
```

# Constantes

- Para resolver isso, utilizamos o comando `#define`, que permite definir constantes dentro do programa que podem ser utilizadas em qualquer lugar onde uma constante seria utilizada (inclusive na declaração de vetores);
- Ex:
  - `#define MAX_ELEMENTOS 10`
  - define a constante `MAX_ELEMENTOS` com o valor 10;

# Constantes

```
#include <stdio.h>

#define linhas 3
#define colunas 3

int matriz[linhas][colunas];

int main(){
    int i,j;

    for(i=0; i < linhas; i++){
        for(j=0; j < colunas; j++){
            scanf("%d",&matriz[i][j]);
        }
    }

    return 0;
}
```

# Constantes

- O formato padrão do comando `#define` é:
  - `#define NOME_DA_CONSTANTE <valor_da_constante>`
- Qualquer tipo de constante pode ser colocada no lugar da constante do `#define`, como pontos flutuantes, cadeias de caracteres, entre outros...;



# Constantes

- O comando `#define` deve ser utilizado SEMPRE abaixo do comando `#include` inicial ou de outro comando `define`, nunca dentro do `main ()`;
- Tipicamente utilizamos letras maiúsculas para o nome das constantes e letras minúsculas para o resto do programa (nome de variáveis, comandos, entre outros...);

# Constantes

- Uma constante não tem tipo. Na verdade, o compilador verifica todos os lugares onde você usou a constante e substitui pelo valor à direita ANTES de realizar a compilação. Esse processo é chamado de pré-compilação;
- Uma boa forma de organizar o seu programa é colocar constantes no começo dele, com nomes claros. Logo ao abrir o seu código será possível identificar os limites de seu programa;

# Exemplo de uso do #define

```
#include <stdio.h>
#define NOTA 10
#define MENSAGEM "Parabens, nota %d\n"
main () {
    printf ("Parabens|, nota %d\n", 10);
    printf ("Parabens, nota %d\n", NOTA);
    printf (MENSAGEM, NOTA);
}
```

# Exercícios

- Crie um programa que:
  - (a) Aloque dinamicamente um array de 5 números inteiros;
  - (b) Peça para o usuário digitar os 5 números no espaço alocado;
  - (c) Mostre na tela os 5 números;
  - (d) Libere a memória alocada;
- Faça um programa que leia do usuário o tamanho de um vetor a ser lido e faça a alocação dinâmica de memória. Em seguida, leia do usuário seus valores e imprima o vetor lido.
- Faça um programa que leia do usuário o tamanho de um vetor a ser lido e faça a alocação dinâmica de memória. Em seguida, leia do usuário seus valores e mostre quantos dos números são pares e quantos são ímpares.

# Algoritmos e Estruturas de Dados I

- Bibliografia:

- Básica:

- CORMEN, Thomas; RIVEST, Ronald, STEIN, Clifford, LEISERSON, Charles. Algoritmos. Rio de Janeiro: Elsevier, 2002.
    - DEITEL, Paul; DEITEL, Harvey. C++ como programar. 5. ed. São Paulo: Pearson, 2006.
    - MELO, Ana Cristina Vieira de; SILVA, Flávio Soares Corrêa da. Princípios de linguagens de programação. São Paulo: Edgard Blücher, 2003.

- Complementar:

- ASCENCIO, A. F. G. & CAMPOS, E. A. V. Fundamentos da programação de computadores. 2. ed. São Paulo: Pearson Education, 2007.
    - MEDINA, Marcelo, FERTIG, Cristina. Algoritmos e programação: teoria e prática. Novatec. 2005.
    - MIZRAHI, V. V.. Treinamento em linguagem C: módulo 1. São Paulo: Makron Books, 2008.
    - PUGA, S. & RISSETTI, G. Lógica de programação e estruturas de dados com aplicações em java. São Paulo: Prentice Hall, 2004.
    - ZIVIANI, Nívio. Projeto de Algoritmos com Implementação em Pascal e C. Cengage Learning. 2010.

# Algoritmos e Estruturas de Dados I

