

# Algoritmos e Estruturas de Dados 1 (AED1)

## Alocação dinâmica de memória

Alocação dinâmica de memória é necessária quando

- a quantidade de memória que o programa necessita não é conhecida a priori,
  - mas apenas durante a execução do mesmo.
- Também pode ser usada para alocar grandes quantidades de memória,
  - pois os limites para alocação estática costumam ser menores.

Para manipular memória dinamicamente é necessária

- a biblioteca `stdlib.h` que possui as funções:
  - `malloc(unsigned int k)`, que
    - recebe como parâmetro um inteiro não negativo `k`,
    - aloca um bloco de memória com `k` bytes e
    - devolve um apontador para o primeiro byte deste bloco.
      - Em geral, este endereço é armazenado num apontador.
  - `free(void *p)`, que
    - recebe como parâmetro um apontador `p` e
    - libera o bloco de memória apontado por `p`.
- Adicionar `#include <stdlib.h>` no início do código para usar a biblioteca

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

int main(int argc, char *argv[]) {
    char *p;
    p = malloc(1);
    scanf("%c", p);
    printf("%c\n", *p);
    return 0;
}
```



- `stdlib.h` também possui outras funções úteis como `rand` e `atoi`.

`Malloc` é frequentemente usado com o operador `sizeof`

- que recebe um tipo,
  - ou variável do tipo desejado,
- e devolve o número de bytes usado por aquele tipo.

```
typedef struct {
    int dia, mes, ano;
} Data;
```

```

Data *d;
// d = malloc(sizeof(Data));
d = malloc(sizeof(*d));
d->dia = 31;
(*d).mes = 12;
d->ano = 2019;
printf("dia = %d, mes = %d, ano = %d\n", (*d).dia, d->mes,
(*d).ano);

```



Nem sempre malloc() devolve o que promete,

- pois a memória é finita
  - e o sistema operacional pode impor outros limites.
- Caso a alocação falhe
  - malloc devolve NULL.
- Por isso, convém verificar se o endereço devolvido é igual a NULL,
  - e reportar erro neste caso.

```

void *mallocSafe(unsigned nbytes) {
    void *p;
    p = malloc(nbytes);
    if (p == NULL) {
        printf("Deu ruim! malloc devolveu NULL!\n");
        exit(EXIT_FAILURE);
    }
    return p;
}

```

- Note que, esta função recebe um inteiro sem sinal (unsigned)
  - já que alocar um número negativo de bytes não faz sentido.
- Um caso em que mallocSafe pode detectar erro
  - é na tentativa de alocar um vetor muito grande.

```

unsigned n;
int *v;
scanf("%u", &n); // testar n = 1000000000
v = mallocSafe(n * sizeof(int));

```

- Quiz1: Por que  $n = 10^9$  deve causar falha?
  - Dica: tem a ver com o número de bytes
    - que conseguimos endereçar com 32 bits.

free(p) libera o bloco de memória apontado por p,

- mas não muda o valor de p.
- Por questão de segurança, convém atribuir NULL para apontadores
  - que apontavam para blocos de memória liberados.

```
free(d);  
d = NULL;
```

- Caso contrário, a informação pode continuar acessível.

```
printf("dia = %d, mes = %d, ano = %d\n", (*d).dia, d->mes,  
(*d).ano);
```

## Vetores alocados dinamicamente

Código:

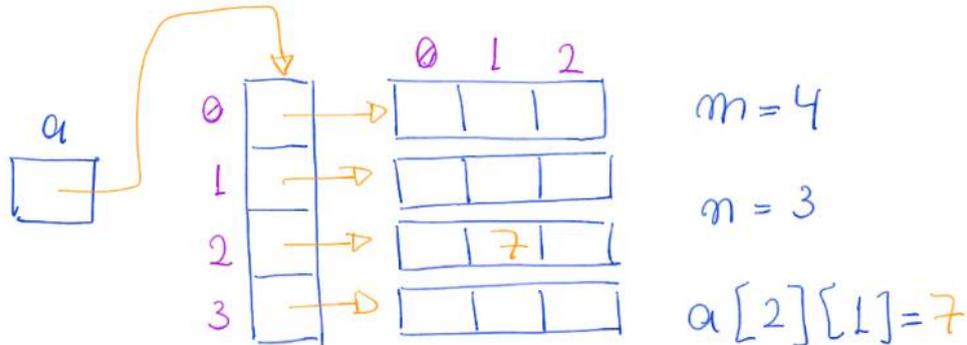
```
int *v, i, n;  
  
n = atoi(argv[1]);  
v = mallocSafe(n * sizeof(int));  
  
// preenchendo o vetor em ordem crescente  
for (i = 0; i < n; i++) {  
    v[i] = i;  
    // *(v + i) = i;  
}  
  
for (i = 0; i < n; i++) {  
    printf("endereço de v[%d] = %p e conteúdo de v[%d] = %d\n",  
i, (v + i), i, v[i]);  
    // printf("endereço de v[%d] = %p e conteúdo de v[%d] =  
%d\n", i, &v[i], i, *(v + i));  
}  
  
free(v);  
v = NULL;
```

## Matrizes alocadas dinamicamente

Pensamos em matrizes bidimensionais como um vetor de vetores.

Alocação dinâmica de matrizes:

- Primeiro alocamos um vetor de apontadores.
  - Cada posição deste corresponde a uma linha da matriz.
- Depois alocamos um vetor de elementos para cada linha.



```
int **a; i, j, m, n;  
  
a = mallocSafe(m * sizeof(int *));  
for (i = 0; i < m; i++)  
    a[i] = mallocSafe(n * sizeof(int));
```

Atribuição e acesso:

- O elemento da linha `i` e coluna `j` de `a` está em `a[i][j]`.
- Exemplo:
  - Atribuição do valor -1 para a célula da linha 4 coluna 2.

```
a[4][2] = -1;
```

- Diferentes maneiras de fazer a leitura deste valor.

```
printf("%d\n", a[4][2]);  
printf("%d\n", (*(a + 4) + 2));  
printf("%d\n", (*(a + 4))[2]);  
printf("%d\n", *(a[4] + 2));
```

- Preencher a matriz

```
for (i = 0; i < m; i++)  
    for (j = 0; j < n; j++)  
        a[i][j] = 1000 * i + j;
```

- Imprimir a matriz

```
for (i = 0; i < m; i++) {  
    for (j = 0; j < n; j++) {  
        printf("%4d ", a[i][j]);  
    }  
    printf("\n");  
}
```

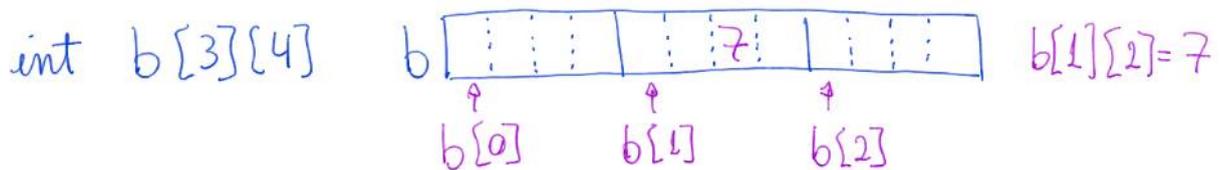
## Liberação de matrizes:

- Para liberar a memória alocada é importante ir na ordem inversa da alocação,
  - i.e., primeiro liberamos o vetor de cada linha
    - e depois liberamos o vetor de apontadores.

```
for (i = 0; i < m; i++) {  
    free(a[i]);  
    a[i] = NULL;  
}  
free(a);  
a = NULL;
```

## Matrizes alocadas estaticamente

- como um bloco único de memória:



- Exemplo de atribuição:

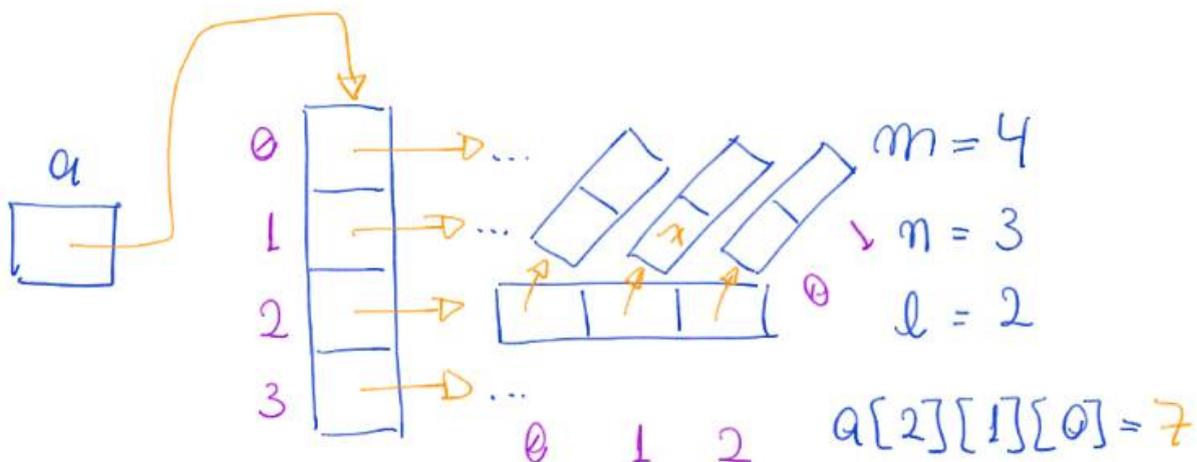
```
int b[3][4];  
b[1][2] = 7;
```

- e diferentes maneiras de fazer a leitura.

```
printf("%d\n", b[1][2]);  
printf("%d\n", *((int *)b + 4 * 1 + 2));  
printf("%d\n", *(int *)((void *)b + 4 * sizeof(int) * 1 +  
sizeof(int) * 2));
```

## Bônus:

- Alocação dinâmica, preenchimento, impressão e liberação de
  - matriz de três dimensões.



```

int ***h;

// alocação dinâmica
h = mallocSafe(m * sizeof(int **));
for (i = 0; i < m; i++) {
    h[i] = mallocSafe(n * sizeof(int *));
    for (j = 0; j < n; j++) {
        h[i][j] = mallocSafe(l * sizeof(int));
    }
}

// preenchimento
for (i = 0; i < m; i++)
    for (j = 0; j < n; j++)
        for (k = 0; k < l; k++)
            h[i][j][k] = 1000000 * i + 1000 * j + k;

// impressão por camadas
for (k = 0; k < l; k++) {
    printf("camada %d\n", k);
    for (i = 0; i < m; i++) {
        for (j = 0; j < n; j++) {
            printf("%7d ", h[i][j][k]);
        }
        printf("\n");
    }
}

// liberação
for (i = 0; i < m; i++) {
    for (j = 0; j < n; j++) {
        free(h[i][j]);
        h[i][j] = NULL;
    }
    free(h[i]);
    h[i] = NULL;
}
free(h);
h = NULL;

```