AED1 - Aula 10 - Alocação dinâmica de memória

Alocação dinâmica de memória é necessária quando

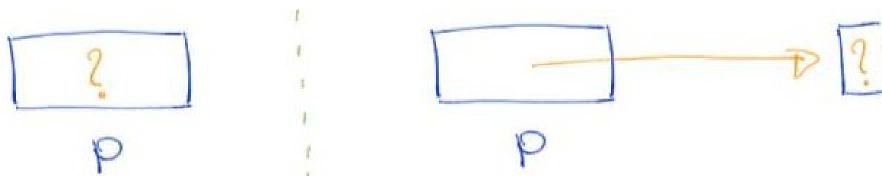
- a quantidade de memória que o programa necessita não é conhecida a priori,
    - mas apenas durante a execução do mesmo

Para manipular memória dinamicamente é necessária a biblioteca stdlib

- que possui as funções
    - malloc(int k)
        - recebe como parâmetro um inteiro k
        - aloca um bloco de memória com k bytes
        - devolve um apontador para o primeiro byte deste bloco
            - em geral este endereço é armazenado num apontador
    - free(void *p)
        - recebe como parâmetro um apontador p
        - 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;
}
```



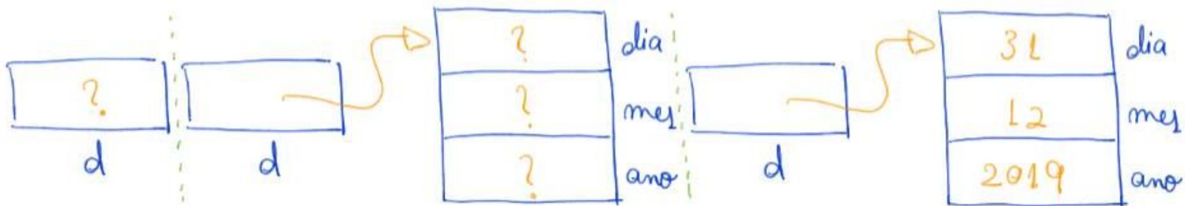
- stdlib também possui outras funções úteis como rand e atoi

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 = 1000000000;
int *v;
v = mallocSafe(n * sizeof(int));
```

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;
int i, n;

v = mallocSafe(n * sizeof(int));

for (i = 0; i < n; i++)
{
    v[i] = i;
    // *(v + i) = i;
}

for (i = 0; i < n; i++)
{
    printf("endereco de v[%d] = %p e conteudo de v[%d] = %d\n", i, (v + i), i, v[i]);
  // printf("endereco de v[%d] = %p e conteudo de v[%d] = %d\n", i, &v[i], i, *(v + i));
}

free(v);
v = NULL;
```

Matrizes dinâmicas

- pensamos em matrizes bidimensionais como um vetor de vetores

Alocação de matrizes

- primeiro alocamos um vetor de apontadores
  - cada posição desse corresponde a uma linha da matriz
- depois alocamos um vetor de elementos para cada linha

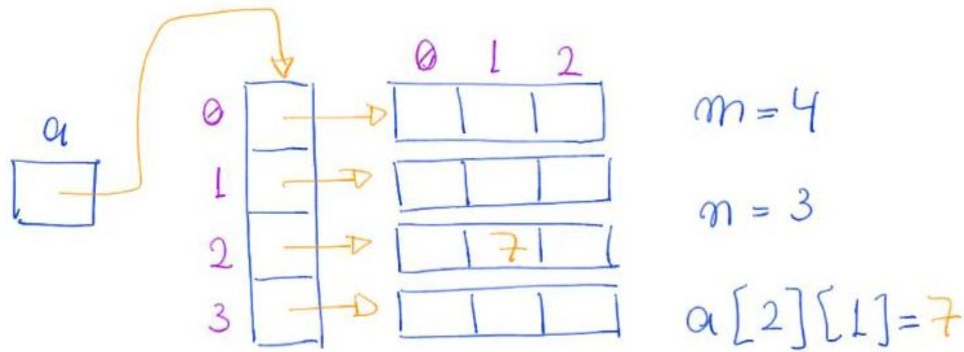
```
int **a;
int i, n, m;

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;
```



- preencher e imprimir matriz

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

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

Liberação de matrizes

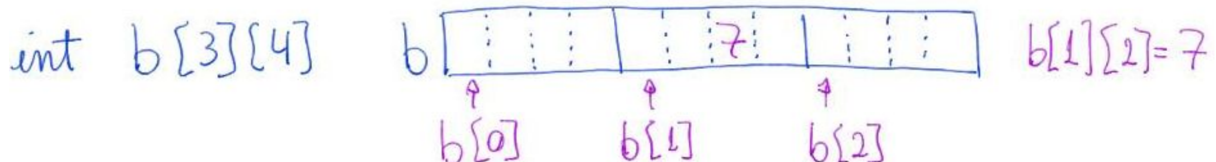
- para liberar a memória alocada é importante ir na ordem inversa da alocação
- i.e., primeiro liberamos o vetor de cada linha
    - e depois liberamos o vetor de apontadores

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

Matrizes automáticas

- matrizes alocadas estaticamente como um bloco único de memória

```
int b[3][4];
b[1][2] = 7;
printf("%d\n", b[1][2]);
```



Bônus:

- alocação dinâmica, preenchimento, impressão e liberação para matriz de três dimensões

```
int ***h;

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 *));
    }
}

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;

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");
    }
}

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;
```