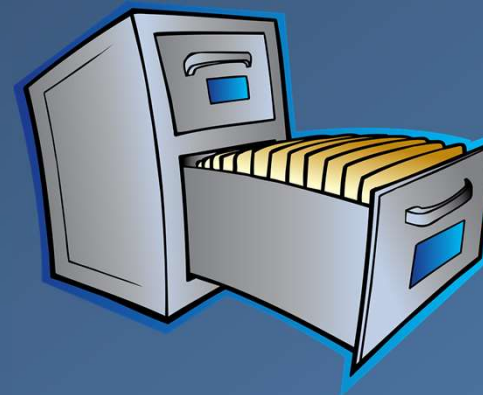


Conceitos Elementares (cont)



Manipulação de Arquivos

Desafio:

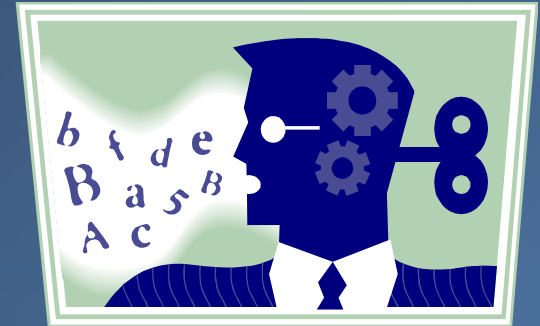
- **Conteúdo abordado:**

- Manipulação de arquivos

- **Material:** <https://www.guru99.com/c-file-input-output.html>

- Refaça o Exercício 4, de modo que toda a saída do programa (impressão das matrizes) seja feita TAMBÉM em um arquivo de texto

- O programa deve pedir ao usuário para fornecer o nome do arquivo de saída. Se uma *string* vazia for fornecida, essa opção deve ser desativada



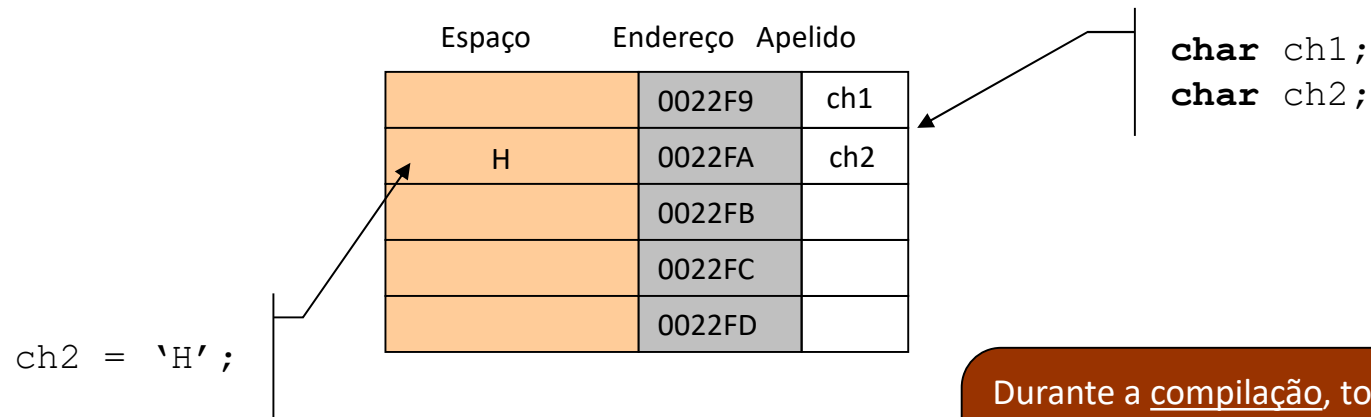
Ponteiros

Variáveis e Endereços

- Toda variável tem um endereço de memória associado a ela.
- Esse endereço é o local onde essa variável é armazenada no sistema.
- Normalmente, o endereço das variáveis não são conhecidos quando o programa é escrito.
- O endereço de uma variável é dependente do sistema computacional e também da implementação do compilador de linguagem que está sendo usado.
- O endereço de uma mesma variável pode mudar entre diferentes execuções de um mesmo programa usando uma mesma máquina.

Alocação de Memória

- Uma variável provê identificação unívoca para uma peça de informação.
- Uma declaração de variável provoca a reserva de uma área de memória, cujo tamanho está relacionado ao tipo da variável.
- O nome da variável permite que essa área de memória seja referenciada pelo “apelido” e não pelo seu endereço.

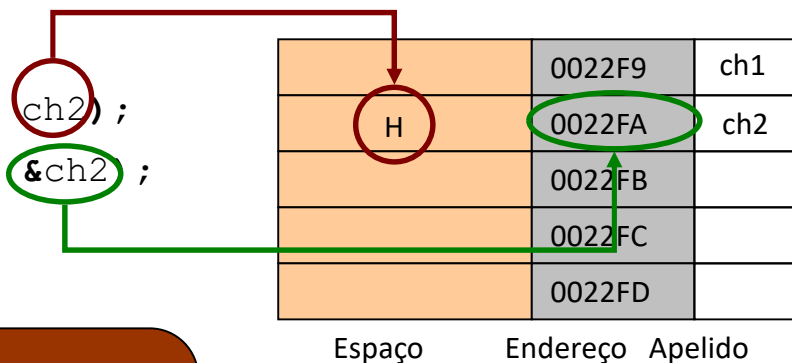


Durante a compilação, toda referência a nomes de variáveis é substituída por um endereço relativo de memória.

Operador de Endereço

- Na linguagem C é possível saber o endereço de uma variável através do operador **&**.
- **Exemplo:**

```
main() {  
    char ch2 = 'H';  
    printf("Conteudo de ch = %c", ch2);  
    printf("Endereco de ch = %p", &ch2);  
}
```



- Função **scanf**:

- Recebe como parâmetro uma referência de endereço (&ch2).
- Lê dados do teclado e armazena no endereço fornecido.

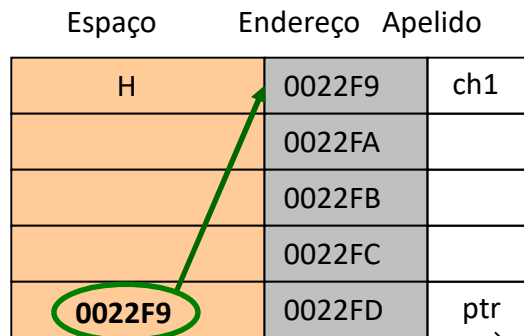
Ponteiros

- Variáveis que armazenam endereços de memória
- Para declarar uma variável do tipo ponteiro utilizamos o operador unário *
 - `int *ap_int;`
 - `char *ap_char;`
 - `float *ap_float;`
 - `double *ap_double;`
- Para cada tipo de dados, existe um tipo de ponteiro para guardar o seu endereço
- Obs: o operador * deve preceder o nome da variável
 - `int *ap1, *ap_2, *ap_3;`
- Ao atribuir o endereço de uma variável a um apontador, dizemos que o mesmo “aponta” para a variável

Apontadores (Ponteiros)

- Exemplo:

Espaço	Endereço	Apelido
H	0022F9	ch1
	0022FA	
	0022FB	
	0022FC	
0022F9	0022FD	ptr



```
char ch1 = 'H';  
char *ptr; /* apontador para char */  
ptr = &ch1; /* ap_x aponta para ch1 */
```

Importante: o espaço ocupado por um apontador depende do espaço de endereçamento de memória do sistema.
Neste exemplo: 3 bytes.

Apontadores (Ponteiros)

- Importante: para acessarmos o valor de uma variável apontada por um endereço, também usamos o operador *

Espaço	Endereço	Apelido
M	0022F9	ch1
	0022FA	
	0022FB	
	0022FC	
0022F9	0022FD	ptr

```
char ch1 = 'H';  
char *ptr; /* apontador para char */  
ptr = &ch1; /* ap_x aponta para ch1 */  
  
*ptr = 'X';
```

Importante: `*ptr` pode ser usado em qualquer contexto que a variável `ch1` seria.

Apontadores para Registros

- Para acessar os elementos de um registro através de um apontador, devemos primeiro acessar o registro e depois acessar o campo desejado

```
typedef struct ponto {  
    double x;  
    double y;  
} Ponto;  
Ponto *ap_p, p;  
ap_p = &p;  
(*ap_p).x = 4.0;  
(*ap_p).y = 5.0;
```

- Os parênteses são necessários pois o operador * tem prioridade menor que o operador "."

Apontadores para Registros

- Para simplificar o acesso aos campos de um registro através de apontadores, foi criado o operador “->”.
- Usando este operador acessamos os campos de um registro diretamente através do apontador

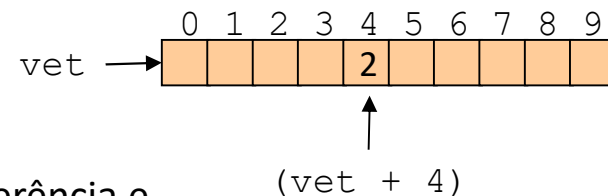
```
typedef struct ponto {  
    double x;  
    double y;  
} Ponto;  
Ponto *ap_p = NULL, p;  
ap_p = &p;  
ap_p->x = 4.0;  
ap_p->y = 5.0;
```

Vetores e Matrizes

- Vetores e Matrizes sempre foram apontadores, mas a sintaxe de vetores “esconde” esse fato.
 - Uma variável que representa um vetor é implementada por um apontador constante para o primeiro elemento do vetor.
 - A operação de indexação corresponde a deslocar este apontador ao longo dos elementos alocados ao vetor (sem perder a referência inicial).

```
int vet[10];
```

```
vet[4] = 2;
```



- Portanto,
 - Vetores são sempre passados por referência e
 - A linguagem C não pode detectar acessos fora dos limites do vetor

Vetores de Ponteiros

- Não existe diferença entre vetores de apontadores e vetores de tipos simples.
 - basta observar que o operador `*` tem precedência menor que o operador de indexação `[]`

Ex:

```
int main() {  
    char *cores[] = {"amarelo", "verde", "vermelho", "laranja", "preto"};  
  
    int a=3, b=5, c=78, d=23;  
    int *numeros[] = {&a, &b, &c, &d};  
  
    printf("%d    %s", *numeros[0], cores[0]);  
    getch();  
    return 0;  
}
```

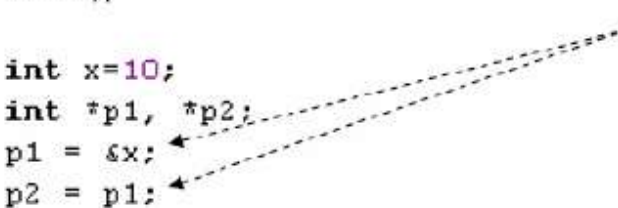
Expressões com Ponteiros

- Alguns aspectos especiais sobre ponteiros:
 - Atribuição de Ponteiros
 - Aritmética de Ponteiros
 - Comparação de Ponteiros

Atribuição de Ponteiros

Como é o caso com qualquer variável, um ponteiro pode ser usado no lado direito de um comando de atribuição para passar seu valor para outro ponteiro:

```
#include<stdio.h>
int main()
{
    int x=10;
    int *p1, *p2;
    p1 = &x;
    p2 = p1;
    printf("O endereço de p2 eh: %p\n\n", p2);
    printf("O conteudo do endereco apontado pelo ponteiro p1 eh: %d\n\n", *p1);
    printf("O conteudo do endereco apontado pelo ponteiro p2 eh: %d\n\n", *p2);
    system("pause");
}
```



Tanto p1 quanto p2 apontam para o endereço de memória da variável x

Aritmética de Ponteiros

- Existem apenas duas operações aritméticas que podem ser usadas com ponteiros:
 - Adição (incremento) e
 - Subtração (decremento).

- Quando incrementamos um ponteiro ele passa a apontar para o próximo valor do mesmo tipo para o qual o ponteiro aponta;
 - Se tivermos um ponteiro para um inteiro e o incrementamos ele passa a apontar para o próximo inteiro;

Aritmética de Ponteiros

- A aritmética de ponteiros não se limita apenas ao incremento e decremento, podemos somar ou subtrair inteiros de ponteiros:

```
#include<stdio.h>
int main()
{
    float *p1;
    printf("O endereço de p1 eh: %p",p1);
    p1 = p1+20;      //equivale à p1+=20;
    printf("\n\nO novo endereço de p1 eh: %p\n\n",p1);
    system("pause");
}
```

p1 apontará para o 20º elemento do tipo float adiante (20 x 4 bytes adiante)

Aritmética de Ponteiros

- Além de adição e subtração entre um ponteiro e um inteiro, nenhuma outra operação aritmética pode ser efetuada com ponteiros;
- Não podemos multiplicar ou dividir ponteiros;
- Não podemos adicionar ou subtrair o tipo float ou o tipo double a ponteiros;
- Quando uma variável precisa ser acessada de diferentes partes do programa.

Aritmética de Ponteiros

- Exemplo:

```
main() {  
    int v[10];  
    int *el;  
    int i;  
  
    el = &v[0];  
  
    /*inicializa conteudo de v via ponteiro */  
    for (i=0; i<10; ++i)  
        *(el + i) = 0;  
}
```

Ou seja: a expressão `el+i` aponta para `v[i]`.

Comparação de Ponteiros

- É possível comparar dois ponteiros em uma expressão relacional (<, <=, > e >=) ou se eles são iguais ou diferentes (== e !=);
- A comparação entre dois ponteiros se escreve como a comparação entre outras duas variáveis quaisquer:

```
#include<stdio.h>
int main()
{
    int x=10, y=10;
    int *p1, *p2;
    p1 = &x;
    p2 = &y;
    if(p1>p2)
        printf("A variavel x esta armazenada em um endereco de memoria acima da variavel y");
    else
        printf("\n\nA variavel y esta armazenada em um endereco de memoria acima da variavel x");
    printf("\n\nCertificando...\n\t\tEndereco de x: %p \n\t\tEndereco de y: %p\n\n");
    system("pause");
}
```

Expressões com Ponteiros

- **Exercícios:**

Considere o trecho de programa abaixo. Depois de executado, quais são os valores associados aos itens de (a) a (g). Suponha que os endereços das variáveis *u* e *v* são 1000 e 1004 respectivamente.

```
int v, u;  
int *pv, *pu;  
v = 45;  
pv = &v;  
*pv = v + 1;  
u = *pv + 1;  
pu = &u;
```

- (a) &*v*
- (b) *pv*
- (c) **pv*
- (d) *u*
- (e) &*u*
- (f) *pu*
- (g) **pu*

Práticas:

6. Crie um programa que declare um vetor de inteiros de 20 posições. Sem utilizar índices, faça o seguinte:
 - Preenchê-lo com valores aleatórios de 1 – 100
 - Ler um inteiro “aux” (entre 0-19)
 - Imprimir o valor contido na posição “aux” do vetor
 - Imprimir o vetor inteiro, na ordem inversa.

- OBS: lembre-se de NÃO usar índices.

Práticas:

7. Crie programa que defina duas estruturas: Pessoa e Endereco. Pessoa deve ter nome, idade, RG e *referência* (ponteiro) para Endereco. Endereco deve ter rua, numero, complemento, cidade. O programa deve declarar e inicializar uma variável do tipo Pessoa e outra do tipo Endereco (ligando uma à outra).

Implemente a função

```
void alteraEndereco (Pessoa *p, Endereco *e)
```

que deve substituir o endereco da pessoa “p” por “e”

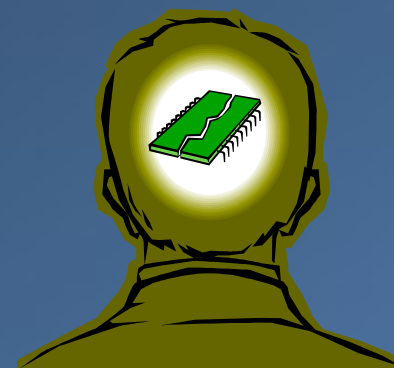
Implemente a função

```
void atualizaEndereco (Pessoa *p, char m, char *valor)
```

atualiza um componente do endereço (“m” indica qual: r, n, c, C)

Implemente a função

```
void imprimePessoa (Pessoa *p)
```

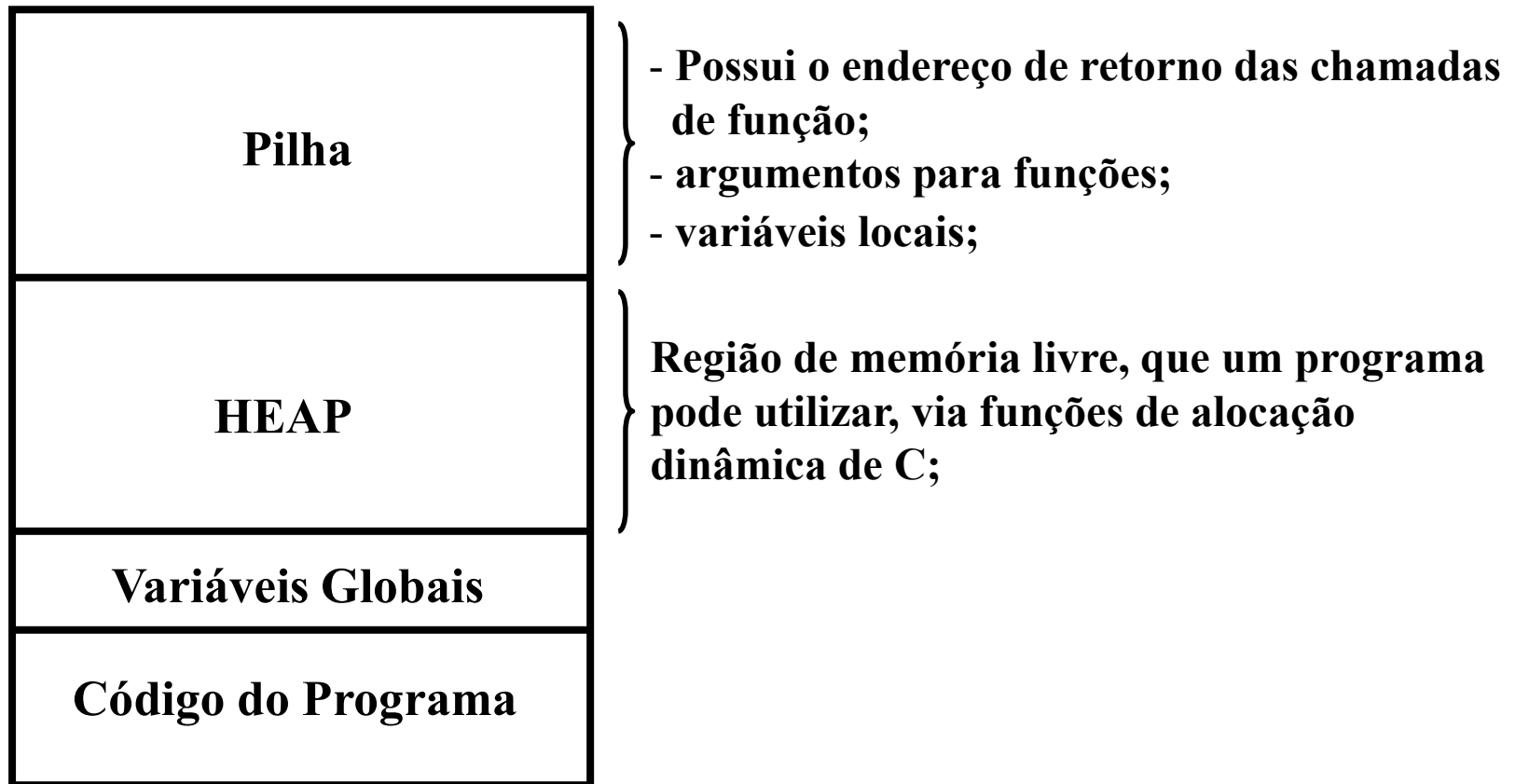



Alocação Dinâmica de Memória

Alocação de Memória

- Duas maneiras (mais comuns) de reservar memória:
 - a. Reserva estática de espaços de memória com tamanho fixo, na forma de variáveis locais :
`int a; char nome[64];`
 - b. Reserva dinâmica de espaços de memória de tamanho arbitrário, com o auxílio de ponteiros:
`int *a; char *nome;`
- Variáveis não podem ser acrescentadas em tempo de execução
 - Porém, um programa pode precisar de quantidade variável de memória
- A reserva só ocorre durante a execução do programa, através de requisições ao SO

Mapa de Memória



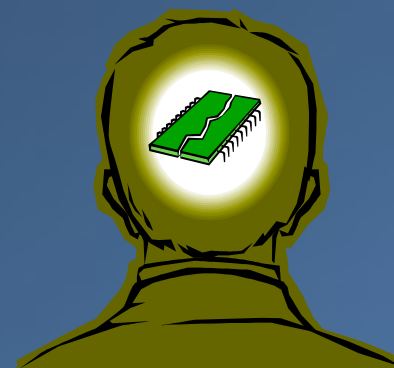
Mapa conceitual de memória de um programa em C

Alocação de Memória

- Vantagens da alocação dinâmica:
 - Flexibilidade: muitas vezes não temos como prever, antecipadamente, as necessidades de uso de memória.
 - Economia: podemos reservar memória de acordo com a necessidade imediata. Evitamos super-dimensionamentos.
- Desvantagem principal:
 - Gerenciamento de Memória:
 - A alocação dinâmica atribui parte da responsabilidade de gerenciar memória ao programador. Essa tarefa, geralmente, é propensa a erros de difícil detecção e correção.
 - Gerenciar a memória envolve: reserva de memória, acesso correto às regiões alocadas, liberação de regiões não mais utilizadas

Funções de Alocação em C

- A alocação e liberação de espaços de memória é feito por funções da biblioteca “stdlib.h” (em alguns sistemas “malloc.h”)
- As principais:
 - [malloc\(size\)](#): “*memory allocation*” Aloca espaço em memória
 - [free\(ref\)](#): Libera espaço em memória, alocado dinamicamente
- Alternativas:
 - [calloc\(n, size\)](#): “*count allocation (?)*” Aloca espaço em memória para um *array* de *n* elementos de tamanho *size*
 - [realloc\(ref, size\)](#): Modifica o tamanho de um bloco de memória previamente alocado.



Funções de Alocação Dinâmica: referência

Função malloc()

- Aloca um bloco consecutivo de bytes na memória e retorna o endereço deste bloco.
- Devemos informar o tamanho do bloco, por parâmetro, em número de bytes
 - Frequentemente, devemos usar a função “*sizeof()*”
- O espaço alocado pode ser usado para armazenar qualquer tipo de dado (void *).
 - Devemos converter o tipo genérico retornado (void *) para o tipo desejado (*cast*)

Ex:

```
Aluno *a;  
a = (Aluno *)malloc(sizeof(Aluno));
```

Função free()

- Libera o uso de um bloco de memória, permitindo que este espaço seja reaproveitado.
- Deve ser passado para a função free() exatamente o mesmo endereço retornado por uma chamada da função malloc()
- A determinação do tamanho do bloco a ser liberado é feita automaticamente.

Ex:

```
int *p;  
p = (int*) malloc(100 * sizeof(int));  
free(p);
```


Função calloc()

- Função equivalente a malloc(), usada para alocar espaço para um vetor de elementos

`calloc(10, sizeof(int)) ≡ malloc(10 * sizeof(int))`

- Devemos informar, por parâmetro, o tamanho do vetor e o tamanho (em bytes) de cada elemento desse vetor.
 - O espaço alocado é iniciado com bits 0
- Esta função também retorna um ponteiro para void (void*)
 - Devemos converter o tipo genérico retornado para o tipo desejado

Ex:

```
Aluno *a;  
a = (Aluno *) calloc(10, sizeof(Aluno));
```

Função realloc()

- Função utilizada para modificar o tamanho ocupado por uma área de memória já alocada
- Devemos informar, por parâmetro, a referência da área a ser redimensionada, e o novo tamanho (em bytes).
 - Se a área original não puder ser redimensionada, uma nova área é criada, e a antiga é liberada.
- Esta função também retorna um ponteiro para void (void*)

Ex: `int *a, *b;`

```
    a = (int *)malloc(sizeof(int));
```

```
    b = (int *)realloc(a, sizeof(int)*4);
```

OBS: `realloc(NULL, size) ≡ malloc(size)`

Exemplo: Alocação de Matrizes

- No caso de vetores multidimensionais (e.g. matrizes), devemos alocar um vetor de apontadores, i.e. um apontador por linha, e depois um vetor de elementos para cada linha

Ex:

```
float **mat; /* matriz de elementos do tipo float */
int nlin = 10, ncol=10; /* numeros de linhas e colunas */
mat = (float **)calloc(nlin, sizeof(float *));
        /* aloca vetor de ponteiros para variaveis float */
if (mat != NULL) {
    int i;
    for (i=0; i < nlin; i++) {
        mat[i] = (float *)calloc(ncol, sizeof(float));
                /* aloca vetor de variaveis float */
    }
}
```

Práticas:

8. Implemente um programa que:

- a. Crie uma função que receba “tam” como parâmetro um número inteiro entre 10 e 100. Então, deve criar um vetor de inteiros com números entre 0 e 50, cujo tamanho é definido por “tam”.
- b. Crie uma função que receba dois vetores de inteiros (e seus respectivos tamanhos), como parâmetro, e retorne a concatenação dos dois como um terceiro vetor.
- c. Crie uma função que receba como parâmetro um vetor de inteiros (e seu tamanho), e imprima seus elementos na tela.
- d. O programa principal deve usar a função em (a.) para criar dois vetores de inteiros de tamanhos distintos (definidos pelo usuário). Deve, então, usar a função em (b.) para concatenar os dois vetores. Finalmente, deve usar a função em (c.) para imprimir os dois vetores originais e o vetor retornado.

Práticas:

9. Altere o programa do Exercício 7, de forma que no lugar de variáveis do tipo Pessoa e Endereço, ponteiros para esses tipos sejam declarados.

Implemente a função

```
Pessoa criaPessoa ()
```

que deve instanciar Pessoa (e Endereço) e deve pedir ao usuário todas as informações (não usual, mas funciona como exercício)

Altere a função abaixo para que receba os dados do endereço e crie um novo objeto Endereco, para associar a “p”

```
void alteraEndereco (Pessoa *p, <dados de endereço>)
```

Faça as alterações que achar necessárias às outras funções, para que continuem funcionando.