

Alocação dinâmica

Aprenda ponteiros antes de continuar

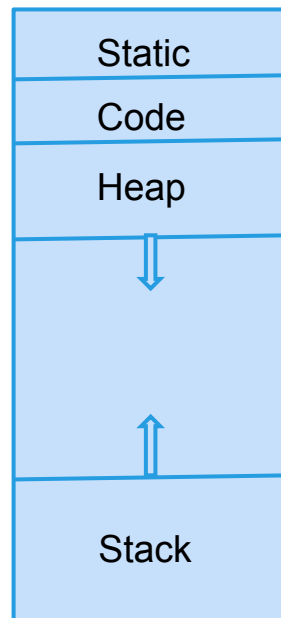
Layout de um programa em memória

Static: Tamanho fixo durante toda a execução. Espaço em que ficam as variáveis globais. Manutenção feita pelo sistema.

Code: Tamanho fixo durante toda a execução. Espaço em que ficam as instruções do programa.

Endereços:

0x00
0x01
0x02
...



Heap: Tamanho varia durante a execução. Espaço em que são alocadas as variáveis dinâmicas. Cresce de endereços **pequenos** para endereços **grandes**. Manutenção feita pelo **programador!**

Stack: Tamanho varia durante a execução. Espaço em que são alocadas as variáveis estáticas. Cresce de endereços **grandes** para endereços **pequenos**. Manutenção feita pelo compilador.

Funções de alocação

As seguintes funções são utilizadas para gerenciar memória dinamicamente:

```
#include <stdlib.h>
void* malloc (size_t size);           // aloca memoria nao inicializada
void* calloc (size_t num, size_t size); // aloca memoria inicializada com zeros
void* realloc (void* ptr, size_t size); // realoca um espaco ja alocado com um novo tamanho
void free (void* ptr);                // libera memoria
```

Mais informações sobre as funções:

<http://www.cplusplus.com/>

Alocando uma variável

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

int main() {
    int* ptr;
    ptr = (int*) malloc(sizeof(int));
    return 0;
}
```

Alocando uma variável

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

int main() {
    int* ptr;
    ptr = (int*) malloc(sizeof(int));
    return 0;
}
```

Operador sizeof: Retorna o tamanho de um **tipo** em **bytes**. Não é função! O próprio compilador transforma `sizeof(int)` em 4 (por exemplo); e compila uma chamada à função `malloc()` passando diretamente este número como argumento.

Utiliza-se um operador para manter portabilidade em diferentes arquiteturas.

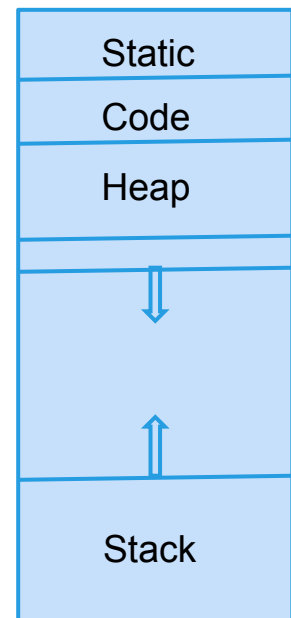
Tipos ponteiro, por exemplo, têm tamanho 4 bytes em arquiteturas de 32 bits e 8 bytes em arquiteturas de 64 bits!

Alocando uma variável

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

```
int main() {
    int* ptr;
    ptr = (int*) malloc(sizeof(int));
    return 0;
}
```

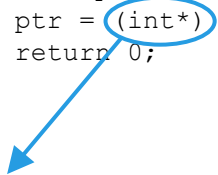
Endereços:
0x00
0x01
0x02
...



Alocando uma variável

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

int main() {
    int* ptr;
    ptr = (int*) malloc(sizeof(int));
    return 0;
}
```



Coerção (casting) necessária para atribuir à variável `ptr`, do tipo `int*`, o valor de uma expressão do tipo `void*`.

As funções de alocação utilizam o tipo “ponteiro para vazio”, porque são utilizadas para alocar variáveis de qualquer tipo.

Um ponteiro é um endereço, ou seja, um número contador de bytes na memória. Portanto, tipos-ponteiro diferentes são na verdade tipos basicamente equivalentes.

Liberando uma variável


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

int main() {
    int* ptr;
    ptr = (int*) malloc(sizeof(int));
    /* codigo do programa vem aqui */
    free(ptr);
    return 0;
}
```


Liberando uma variável

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

int main() {
    int* ptr;
    ptr = (int*) malloc(sizeof(int));
    /* codigo do programa vem aqui */
    free(ptr);
    return 0;
}
```



Memória alocada dinamicamente deve ser gerenciada completamente pelo próprio programador! **É preciso manter em mente que variáveis assim são utilizadas através de seus ponteiros!** Se um ponteiro é sobrescrito com outro ponteiro, acontece **vazamento de memória**. Por isso, mantém-se a prática de liberar memória que não será mais utilizada, se realmente for necessário sobrescrever um ponteiro de memória dinâmica. É preciso liberar memória dinâmica até mesmo imediatamente antes de um programa ser encerrado e retornar o controle para o sistema operacional.

Matrizes estáticas

Matrizes estáticas são alocadas de forma **contígua**. Particularmente, a linguagem C aloca matrizes estáticas como uma concatenação de linhas - e não uma concatenação de colunas. Observe o programa e sua saída abaixo:

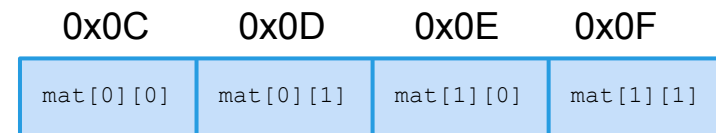
```
#include <stdio.h>
```

```
int main() {  
    char mat[2][2];  
    printf("%p %p\n", &mat[0][0], &mat[0][1]);  
    printf("%p %p\n", &mat[1][0], &mat[1][1]);  
    return 0;  
}
```

Saída:

```
0028FF0C 0028FF0D
```

```
0028FF0E 0028FF0F
```



Note que os endereços dos caracteres tanto da primeira quanto da segunda linha são adjacentes. Além disso, o endereço do último caractere da primeira linha é adjacente ao endereço do primeiro caractere da segunda linha.

Matrizes estáticas

Utilizando alocação dinâmica de memória, não é possível alocar uma matriz contígua. Isto se deve ao tratamento diferenciado que o compilador dá a diferentes declarações de matrizes de um mesmo tipo de dados, ao gerar instruções de endereçamento. Uma declaração estática exige que o programador informe as **dimensões** da matriz **diretamente no código-fonte**. Ao declarar uma matriz colocando suas dimensões diretamente no código-fonte de um programa, o compilador gera instruções de endereçamento a partir de **um número, um endereço e dois índices**: o número de colunas da matriz, o endereço do primeiro elemento da matriz; e os índices de linha e coluna do elemento desejado.

```
#include <stdio.h>

int main() {
    char mat[2][3];
    int i = 1, j = 2;
    mat[i][j] = "F";
    return 0;
}
```

No exemplo ao lado, o número de colunas é 3 (três), o endereço do primeiro elemento é o símbolo `mat` e os índices de linha e coluna são respectivamente os símbolos `i` e `j`.

Matrizes estáticas

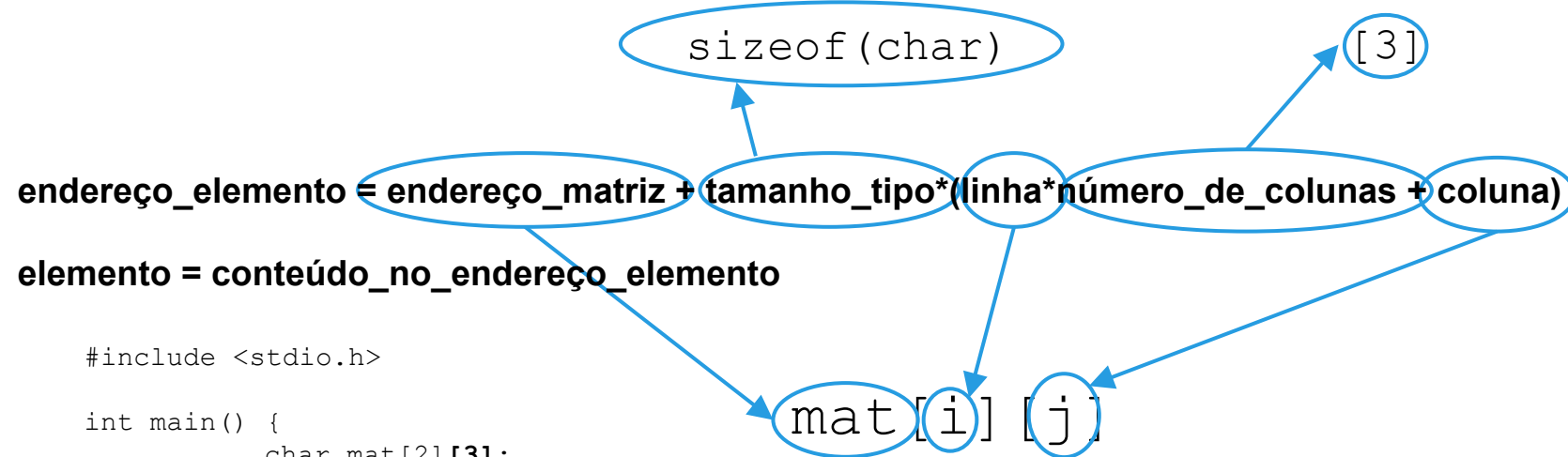
Fórmulas utilizadas pelo compilador para calcular o endereço de um elemento de uma matriz estática:

endereço_elemento = endereço_matriz + tamanho_tipo*(linha*número_de_colunas + coluna)

elemento = conteúdo_no_endereço_elemento

Matrizes estáticas

Fórmulas utilizadas pelo compilador para calcular o endereço de um elemento de uma matriz estática:



Matrizes estáticas

Fórmulas utilizadas pelo compilador para calcular o endereço de um elemento de uma matriz estática:

$$\text{endereço_elemento} = \text{endereço_matriz} + \text{sizeof(char)} * (\text{linha} * \text{número_de_colunas} + \text{coluna})$$

elemento = conteúdo_no_endereço_elemento

The diagram shows the formula for calculating the address of an element in a static matrix. It includes annotations: 'sizeof(char)' is circled and has an arrow pointing to it from below. '[3]' is circled and has an arrow pointing to 'número_de_colunas' in the formula. The terms 'tamanho_tipo' and 'número_de_colunas' in the formula are also circled.

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

```
int main() {  
    char mat[2][3];  
    int i = 1, j = 2;  
    mat[i][j] = "F";  
    return 0;  
}
```

Se você utilizar o símbolo `mat` no programa, o compilador saberá que deve utilizar `sizeof(char)` como o tamanho do tipo e 3 como o número de colunas.

Matrizes estáticas

Fórmulas utilizadas pelo compilador para calcular o endereço de um elemento de uma matriz estática:

$\text{endereço_elemento} = \text{endereço_matriz} + \text{tamanho_tipo} * (\text{linha} * \text{número_de_colunas} + \text{coluna})$

$\text{elemento} = \text{conteúdo_no_endereço_elemento}$

Observação: apenas **um** acesso à memória é realizado ao utilizar esta fórmula!

Matrizes estáticas

Você deve estar pensando: “Legal! Existe uma fórmula para calcular o endereço de um elemento de uma matriz. E por acaso eu posso utilizar a fórmula eu mesmo para calcular o endereço e acessar o elemento?!” A resposta é **sim**! Experimente executar o seguinte programa:

```
#include <stdio.h>

int main() {
    int mat[2][3]; /* o numero de colunas eh 3 */
    int elemento;

    /* vamos calcular o endereco em bytes do elemento [1][2] */
    long unsigned int endereco_elemento;
    /* endereco em bytes da matriz */
    long unsigned int endereco_matriz = (long unsigned int)mat;

    int linha = 1, coluna = 2;
    mat[linha][coluna] = 16433355;
    /* formula */
    endereco_elemento = endereco_matriz + sizeof(int)*(linha*3 + coluna);
    /* realizamos o casting do endereco para o tipo ponteiro de inteiro e
       acessamos o inteiro apontado por este ponteiro */
    elemento = *( (int*)endereco_elemento );
    printf("elemento: %x\n", elemento); /* imprime o elemento na notacao hexadecimal */
    return 0;
}
```

Matheus Pimenta - matheuscscp@gmail.com

Matrizes dinâmicas

Para se declarar uma matriz sem informar suas dimensões, utiliza-se um tipo **ponteiro-de-ponteiro**. Desta forma, o compilador gera instruções de endereçamento a partir de **dois endereços** e **dois índices**:

- 1- o endereço do primeiro elemento de um vetor de **vetores**, ou seja, de um vetor de **ponteiros**;
- 2- o índice de um elemento deste vetor de ponteiros;
- 3- o endereço que é o próprio conteúdo do elemento referenciado no vetor de ponteiros - e é o endereço do primeiro elemento de um vetor comum;
- 4- e o índice de um elemento deste vetor comum.

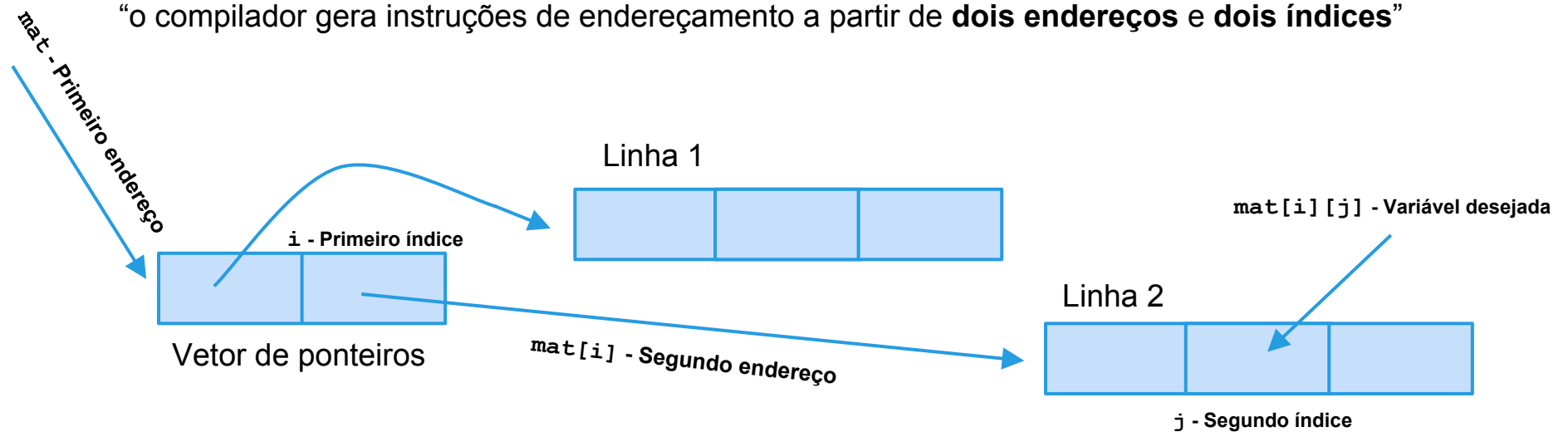
```
#include <stdio.h>

int main() {
    char** mat;
    int i = 1, j = 2;
    mat[i][j] = "F";
    return 0;
}
```

No exemplo ao lado, o endereço do primeiro elemento do vetor de ponteiros é o símbolo `mat`; o índice de um elemento neste vetor de ponteiros é o símbolo `i`; o endereço do primeiro elemento de um vetor comum é a expressão `mat[i]`; e o índice de um elemento neste vetor comum é o símbolo `j`.

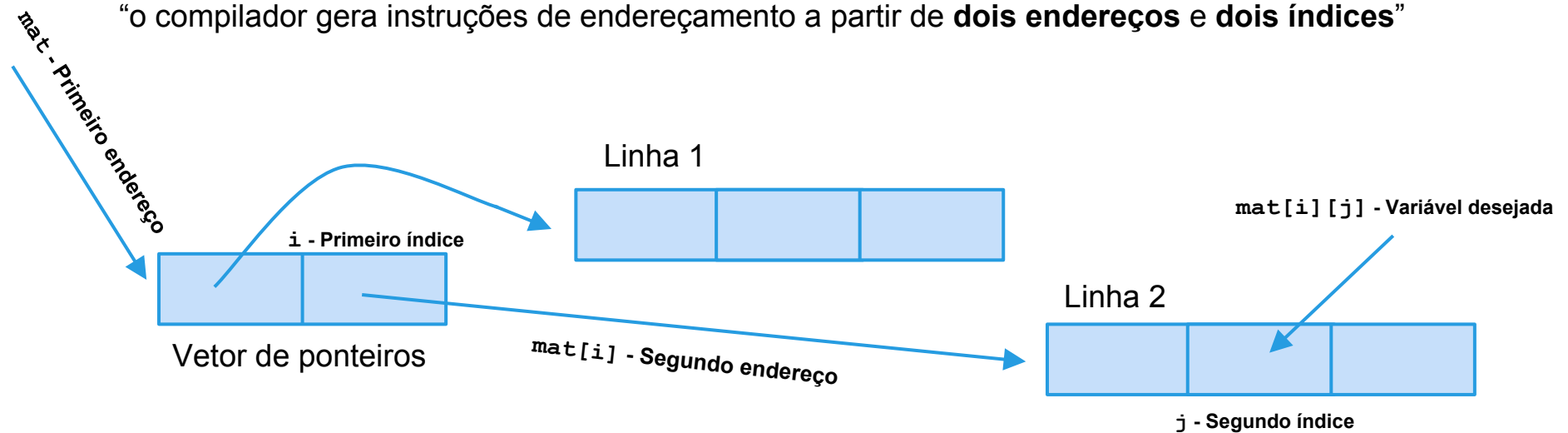
Matrizes dinâmicas

“o compilador gera instruções de endereçamento a partir de **dois endereços** e **dois índices**”



Matrizes dinâmicas

“o compilador gera instruções de endereçamento a partir de **dois endereços** e **dois índices**”



Observação: **dois** acessos à memória são realizados!

Matrizes dinâmicas

Fórmulas utilizadas pelo compilador para calcular o endereço de um elemento de uma matriz dinâmica:

$\text{endereço_linha} = \text{endereço_matriz} + \text{tamanho_ponteiro} * \text{linha}$
 $\text{endereço_elemento} = \text{conteúdo_no_endereço_linha} + \text{tamanho_tipo} * \text{coluna}$
 $\text{elemento} = \text{conteúdo_no_endereço_elemento}$

endereço_matriz:	mat
tamanho_ponteiro:	sizeof(char*)
linha:	i
conteúdo_no_endereço_linha:	mat[i]
tamanho_tipo:	sizeof(char)
coluna:	j
conteúdo_no_endereço_elemento:	mat[i][j]

Matrizes dinâmicas

Note que a **forma**, ou sintaxe, para acessar elementos de matrizes alocadas tanto estática quanto dinamicamente é a mesma: `mat[i][j]`. Por quê?

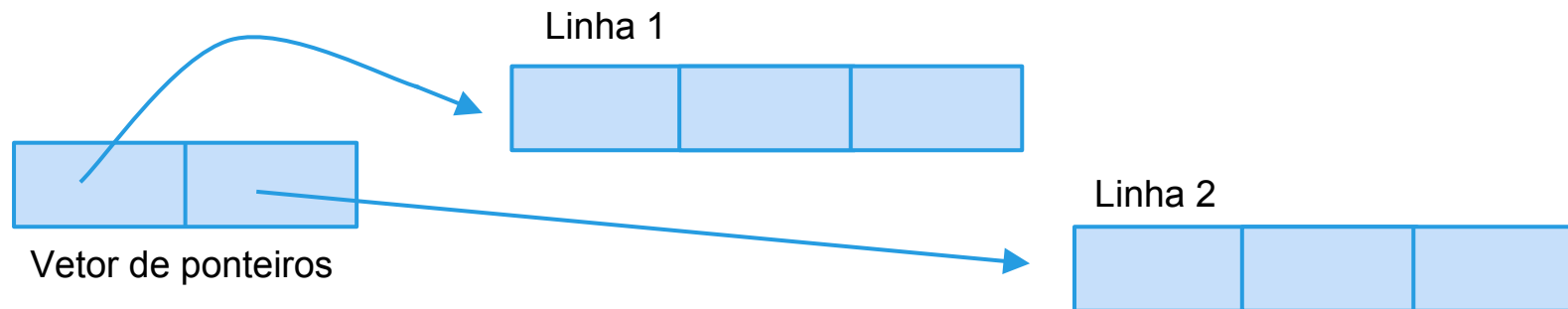
Se `mat` é do tipo `char[][3]`, o compilador gera instruções utilizando as fórmulas do slide 12, realizando apenas **um** acesso à memória. Se `mat` é do tipo `char**`, o compilador gera instruções utilizando as fórmulas do slide 20, realizando **dois** acessos à memória.

Alocando uma matriz

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main() {
5     char** mat;
6     int m = 2, n = 3, i;
7     mat = (char**) malloc(sizeof(char*)*m);
8     for (i = 0; i < m; i++)
9         mat[i] = (char*) malloc(sizeof(char)*n);
10    return 0;
11 }
```

Suponha que queremos alocar uma matriz de m linhas por n colunas. A linha 7 aloca o vetor de ponteiros e os `malloc()` subsequentes alocam os vetores comuns.

Conclusão: alocamos $m + 1$ regiões contíguas de memória, onde a primeira região possui m elementos e as outras m regiões possuem n elementos.



Alocando uma matriz

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main() {
5      char** mat;
6      int m = 2, n = 3, i;
7      mat = (char**) malloc(sizeof(char*)*m);
8      for (i = 0; i < m; i++)
9          mat[i] = (char*) malloc(sizeof(char)*n);
10     /* codigo do programa vem aqui */
11     for (i = 0; i < m; i++)
12         free(mat[i]);
13     free(mat);
14     return 0;
15 }
```

Ao final do programa, é necessário liberar a memória utilizada pela matriz. Repare que a liberação foi feita na ordem contrária. Se o vetor de ponteiros fosse liberado antes, perderíamos as referências dos outros vetores, provocando um vazamento de memória.

Alocando uma matriz

Exercício:

Utilizando como exemplos os slides 16 e 23, escreva um programa que faça a alocação dinâmica de uma matriz de inteiros 2x3, inicialize o elemento `mat[1][2]` com o valor 16433355, acesse este mesmo elemento através das fórmulas do slide 20, imprima o elemento acessado em notação hexadecimal e então libere a matriz antes de retornar o controle para o sistema operacional (`return 0;`).