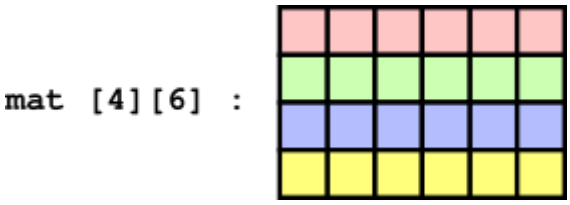
Alocação dinâmica de matrizes

As matrizes **estáticas** podem ser definidas e acessadas de forma bastante simples:

```
int matriz[100][100] ;

for (i = 0; i < 100; i++)
  for (j = 0; j < 100; j++)
    matriz[i][j] = i+j ;
```

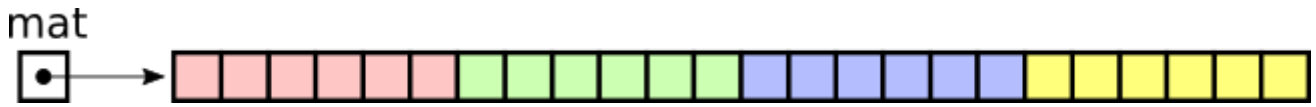
Por outro lado, matrizes **dinâmicas** não são tão simples de alocar e usar. Esta página mostra alguns métodos de como fazê-lo.

Nas figuras a seguir, cada faixa de blocos de uma **mesma cor** representa uma linha da matriz:



Método 1: alocação única com aritmética de ponteiros

Neste método, os elementos da matriz são alocados em um único vetor, linearmente. Os elementos da matriz são acessados explicitamente através de aritmética de ponteiros.



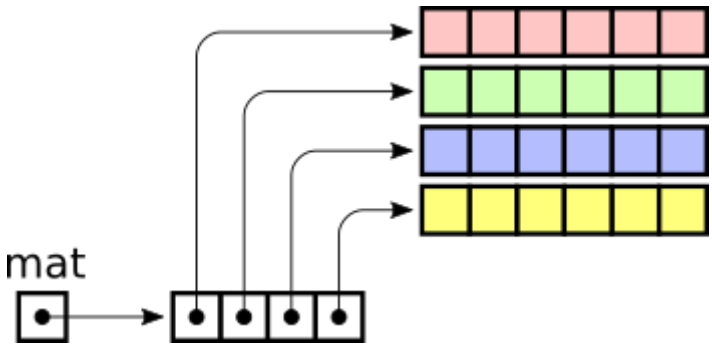

```
#define LIN 4
#define COL 6

int *mat ;
int i, j ;

// aloca um vetor com todos os elementos da matriz
mat = malloc (LIN * COL * sizeof (int)) ;

// percorre a matriz
for (i = 0; i < LIN; i++)
  for (j = 0; j < COL; j++)
    mat[(i*COL) + j] = 0 ; // calcula a posição de cada elemento

// libera a memória da matriz
free (mat) ;
```

Apesar do acesso às posições da matriz ser menos intuitivo, esta abordagem é muito rápida no acesso à memória, pois todos os elementos da matriz estarão alocados em posições contíguas de memória, devido à sua boa localidade de referências.

Método 2: vetor de ponteiros de linhas separadas

Neste método, a matriz é vista e alocada como um vetor de ponteiros para linhas, que são vetores de elementos. O vetor de ponteiros de linhas e os vetores de cada linha são alocados separadamente.

A vantagem desta técnica é que o acesso aos elementos da matriz usa a mesma sintaxe do acesso a uma matriz estática, o que torna a programação mais simples. Entretanto, sua localidade de referências é pior que na abordagem anterior.

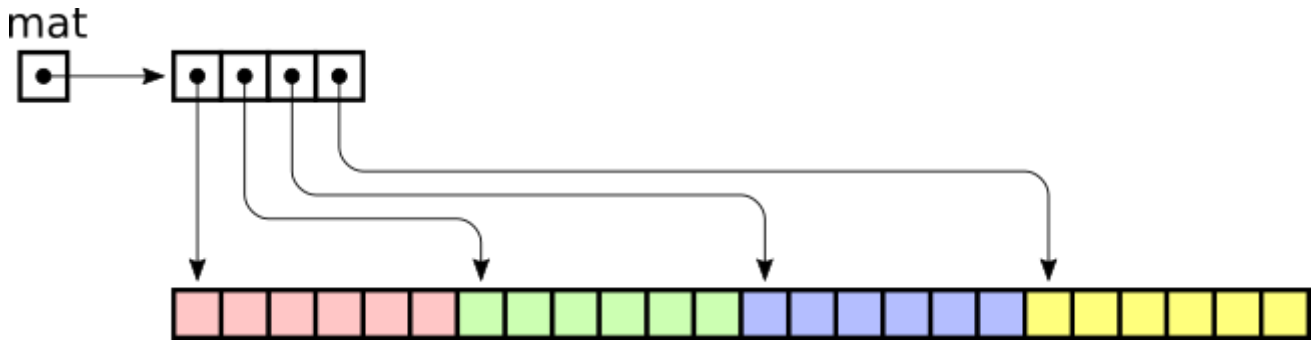



```c
#define LIN 4
#define COL 6

int **mat ;
int i, j ;

// aloca um vetor de LIN ponteiros para linhas
mat = malloc (LIN * sizeof (int*)) ;

// aloca cada uma das linhas (vetores de COL inteiros)
for (i = 0; i < LIN; i++)
  mat[i] = malloc (COL * sizeof (int)) ;

// percorre a matriz
for (i = 0; i < LIN; i++)
  for (j = 0; j < COL; j++)
    mat[i][j] = 0 ;         // acesso com sintaxe mais simples

// libera a memória da matriz
for (i = 0; i < LIN; i++)
  free (mat[i]) ;
free (mat) ;
```

Método 3: vetor de ponteiros de linhas contíguas

Neste método, a matriz é vista e alocada como um vetor de ponteiros para linhas, mas as linhas são alocadas como um único vetor de elementos. O vetor de ponteiros de linhas e o vetor de elementos são alocados separadamente.

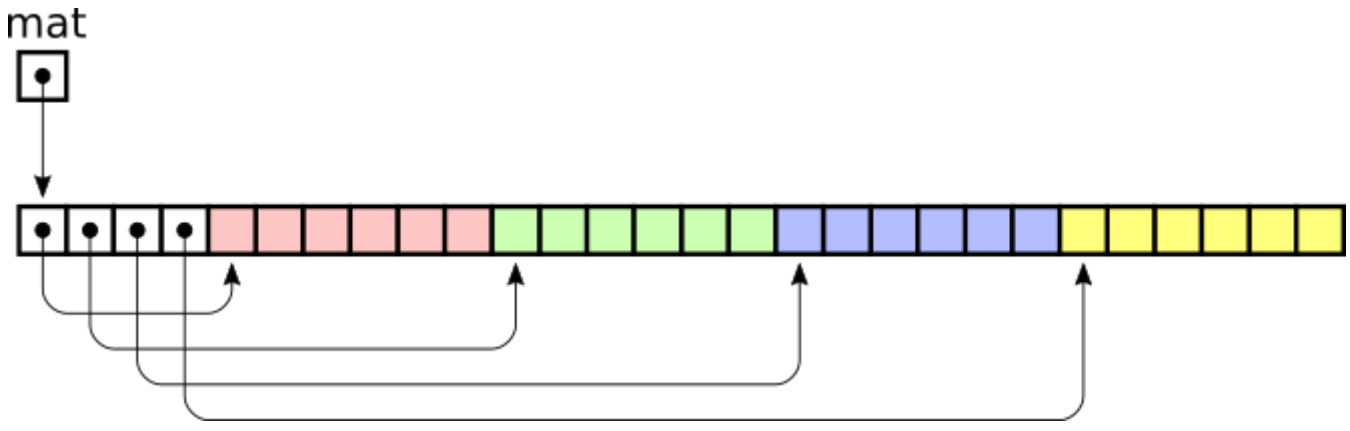
Além de usar a mesma sintaxe de acesso que uma matriz estática, esta abordagem tem mais duas vantagens: precisa somente de duas operações de alocação de memória e todos os elementos da matriz estão alocados sequencialmente na memória, o que facilita operações de cópia de matrizes (usando `memcpy`) ou de leitura/escrita da matriz para um arquivo (usando `fread` ou `fwrite`).




```c
#define LIN 4
#define COL 6

int **mat ;
int i, j ;

// aloca um vetor de LIN ponteiros para linhas
mat = malloc (LIN * sizeof (int*)) ;

// aloca um vetor com todos os elementos da matriz
mat[0] = malloc (LIN * COL * sizeof (int)) ;

// ajusta os demais ponteiros de linhas (i > 0)
for (i = 1; i < LIN; i++)
  mat[i] = mat[0] + i * COL ;

// percorre a matriz
for (i = 0; i < LIN; i++)
  for (j = 0; j < COL; j++)
    mat[i][j] = 0 ;

// libera a memória da matriz
free (mat[0]) ;
free (mat) ;
```

Método 4: área única com vetor de ponteiros e linhas contíguas

O método anterior pode ser modificado, juntando os ponteiros e as linhas da matriz em uma única área de memória:




```c
#define LIN 4
```

```
#define COL 6

int **mat ;
int i, j ;

// aloca um vetor com os ponteiros e os elementos da matriz
mat = malloc (LIN * sizeof (int*) + LIN * COL * sizeof (int)) ;

// ajusta o ponteiro da primeira linha
mat[0] = (int*) (mat + LIN) ;

// ajusta os ponteiros das demais linhas (i > 0)
for (i = 1; i < LIN; i++)
  mat[i] = mat[0] + (i * COL) ;

// percorre a matriz
for (i = 0; i < LIN; i++)
  for (j = 0; j < COL; j++)
    mat[i][j] = i ;

// libera a memória da matriz
free (mat) ;
```

Apesar de ser elegante, esta solução pode apresentar um desempenho fraco, devido a problemas de alinhamento da matriz no cache da memória RAM.

Função de alocação

Caso seu código use matrizes dinâmicas com frequência, recomenda-se criar uma função genérica de alocação, encapsulando um dos métodos acima, como mostra este exemplo:

```
// funções de alocação de matriz genérica 2D
void** aloca_matriz (int nlin, int ncol, int tamanho) ;
void libera_matriz (void **mat) ;

// aloca e libera uma matriz de floats
float** mf ;
mf = (float**) aloca_matriz (1000, 1000, sizeof (float)) ;
...
libera_matriz (mf) ;
```

Exercícios

1. Utilizando alocação dinâmica de matrizes, crie um programa para receber duas matrizes de tamanho 3×3 e calcular a multiplicação delas.
2. Um quadrado mágico é uma matriz com números distintos na qual a soma dos elementos de cada linha, coluna e diagonal é igual. Elabore um algoritmo capaz de encontrar e imprimir na tela um quadrado mágico de tamanho 3×3 e cuja soma dos elementos de cada linha, coluna e diagonal é 15.
3. Desenvolva um programa que recebe uma sequência de 10 palavras e as armazena em uma matriz do tipo `char**`, em seguida, o programa deve trocar a ordem das palavras a fim de ordená-las por ordem alfabética.
4. Utilizando alocação dinâmica, crie um programa que gera uma matriz de 3 dimensões e preenche cada

elemento dessa matriz com a soma dos índices do elemento. Por exemplo: `Matriz[1][2][3] = 1 + 2 + 3 = 6`

5. Reescreva o programa anterior utilizando desta vez alocação única para gerar a matriz de 3 dimensões.