Programação Estruturada
Prof. Rodrigo Hausen
http://progest.compscinet.org

Matrizes e Ponteiros Múltiplos

Recordando: PONTEIROS

Uma declaração tal como:

```
tipo *nome;
```

Declara um ponteiro: referência a uma posição de memória onde se encontra uma variável daquele tipo.

Exemplos:

```
int *numeros;

double *notas;

char *nomeDoMes;
```

Obs.: se apenas declararmos o ponteiro, ele indicará uma região indeterminada.

Recordando: PONTEIROS

- Um ponteiro pode indicar uma área de memória no heap, na pilha **ou em outras regiões de memória** (se houver), **que podem ser imutáveis**
- Declara ponteiro e aloca espaço no **heap** (deve ser desalocado c/ free):
  `char *p = (char *)malloc(tamanho);`
- Declara ponteiro e aloca espaço na **pilha** (desalocação automática):
  `char p[] = "`*inicialização*`";`
- Declara ponteiro e aponta p/ uma constante (região imutável):
  `char *p = "`*inicialização*`";`

Recordando: ARITMÉTICA de PONTEIROS

Operador de referência &

Colocado na frente de uma variável, nos dá o endereço onde esta variável está armazenada.

```
int i = 42;
int *p = &i;

p[0]++; // incrementa o primeiro int
        // na região de memória apontada
        // por p
```

Recordando: ARITMÉTICA de PONTEIROS

Operador de dereferência ***

Colocado na frente de uma **variável ou expressão**, permite acessar o **conteúdo** da posição de memória apontada por essa variável ou expressão.

```
int i = 42;
int *p = &i;

++(*p); // equivale a ++p[0]
```

Atenção! *Neste exmplo, apenas este uso do caractere * indica o operador de dereferência!*

QUIZ: ARITMÉTICA de PONTEIROS

Considere p declarado como

```
int *p;
```

Então **p[0]** equivale a...

a) **p**

b) ***p[0]**

c) **&p**

d) ***p**

QUIZ: ARITMÉTICA de PONTEIROS

Considere p declarado como

```
int *p;
```

Então **p[0]** equivale a...

d) ***p**

QUIZ: ARITMÉTICA de PONTEIROS

Considere p declarado como

```
int *p;
```

Então **&p[0]** equivale a...

a) **p**

b) ***p[0]**

c) **&p**

d) ***p**

QUIZ: ARITMÉTICA de PONTEIROS

Considere p declarado como

```
int *p;
```

Então **&p[0]** equivale a...

a) **p**

QUIZ: ARITMÉTICA de PONTEIROS

Considere p declarado como

```
int *p;
```

Então **p[i]** equivale a...

a) **p+i**

b) **p+(i-1)**

c) ***(p+i)**

d) ***p+i**

e) ***p+(i-1)**

QUIZ: ARITMÉTICA de PONTEIROS

Considere p declarado como

```
int *p;
```

Então **p[i]** equivale a...

c) ***(p+i)**

ARITMÉTICA DE PONTEIROS – exemplo

```
int palindromo(const char *p) {
  const char *q;
  q = p;
  // localiza final da string
  while (*q != '\0') ++q;
  --q; // q aponta para último char
       // diferente de '\0'
  // compara
  while (p < q) {
    if (*p != *q) return 0; // falso
    ++p; --q;
  }
  return 1; // verdadeiro
}
```

Relembrando: PASSAGEM DE PARÂMETROS

Crie o arquivo incrementa.c

```
void incrementa(int i) {
  ++i;
}
```

Teste no cling:

```
[cling]$ .L incrementa.c
[cling]$ int i = 0;
[cling]$ incrementa(i)
[cling]$ i
(int) 0
[cling]$
```

Relembrando: PASSAGEM DE PARÂMETROS

```
void incrementa(int i) {
  ++i;
}
```

- Em C, a passagem de parâmetros para uma função é **sempre por valor**, ou seja, quando passamos uma variável para uma função, o **valor dessa variável é copiado** para o parâmetro correspondente.

- Se quisermos alterar uma variável passada como parâmeto, temos que passar uma **referência** a ela.

Relembrando: PASSAGEM DE PARÂMETROS

Corrija o arquivo incrementa.c

```
void incrementa(int *i) {
  ++(*i);
}
```

Reinicie o cling e teste:

```
[cling]$ .L incrementa.c
[cling]$ int i = 0;
[cling]$ incrementa(&i)
[cling]$ i
(int) 1
[cling]$
```

Relembrando: VETORES e PONTEIROS

Em C, vetores e ponteiros são muito similares.

```
[cling]$ int v[] = { 42, 18, 99 };
[cling]$ int *p;
[cling]$ p = v;
[cling]$ v[1]
18
[cling]$ p[1] = 35;
[cling]$ v[1]
35
[cling]$ *(p+1)
35
```

Tanto v quanto p são referências a uma região de memória. (neste caso, na pilha)

Relembrando: VETORES e PONTEIROS

Diferenças:

- Gerenciamento de memória para vetores é automático. Para ponteiros, fica a cargo do programador.
- Funções não podem retornar vetores, mas podem retornar ponteiros
- Durante a compilação, o compilador guarda o tamanho de um vetor. Portanto, em tempo de compilação, é possível saber a quantidade de elementos de um vetor. Para ponteiros, isto não ocorre.

TAMANHO DE UM VETOR EM TEMPO DE COMPILAÇÃO

Suponha que executemos o cling em uma máquina para a qual um int seja representado em 4 bytes.

```
[cling]$ int v[] = { 42, 18, 99 };
```

TAMANHO DE UM VETOR EM TEMPO DE COMPILAÇÃO

Suponha que executemos o cling em uma máquina para a qual um int seja representado em 4 bytes.

```
[cling]$ int v[] = { 42, 18, 99 };
[cling]$
```

TAMANHO DE UM VETOR EM TEMPO DE COMPILAÇÃO

Suponha que executemos o cling em uma máquina para a qual um int seja representado em 4 bytes.

```
[cling]$ int v[] = { 42, 18, 99 };
[cling]$ sizeof(v[0])
```

TAMANHO DE UM VETOR EM TEMPO DE COMPILAÇÃO

Suponha que executemos o cling em uma máquina para a qual um int seja representado em 4 bytes.

```
[cling]$ int v[] = { 42, 18, 99 };
[cling]$ sizeof(v[0])
(unsigned long) 4
[cling]$
```

TAMANHO DE UM VETOR EM TEMPO DE COMPILAÇÃO

Suponha que executemos o cling em uma máquina para a qual um int seja representado em 4 bytes.

```
[cling]$ int v[] = { 42, 18, 99 };
[cling]$ sizeof(v[0])
(unsigned long) 4
[cling]$ sizeof(v)
```

TAMANHO DE UM VETOR EM TEMPO DE COMPILAÇÃO

Suponha que executemos o cling em uma máquina para a qual um int seja representado em 4 bytes.

```
[cling]$ int v[] = { 42, 18, 99 };
[cling]$ sizeof(v[0])
(unsigned long) 4
[cling]$ sizeof(v)
(unsigned long) 12
[cling]$
```

TAMANHO DE UM VETOR EM TEMPO DE COMPILAÇÃO

Suponha que executemos o cling em uma máquina para a qual um int seja representado em 4 bytes.

```
[cling]$ int v[] = { 42, 18, 99 };
[cling]$ sizeof(v[0])
(unsigned long) 4
[cling]$ sizeof(v)
(unsigned long) 12
[cling]$ // como determinar tamanho de v?
[cling]$
```

TAMANHO DE UM VETOR EM TEMPO DE COMPILAÇÃO

Suponha que executemos o cling em uma máquina para a qual um int seja representado em 4 bytes.

```
[cling]$ int v[] = { 42, 18, 99 };
[cling]$ sizeof(v[0])
(unsigned long) 4
[cling]$ sizeof(v)
(unsigned long) 12
[cling]$ // como determinar tamanho de v?
[cling]$ sizeof(v) / sizeof(v[0])
```

TAMANHO DE UM VETOR EM TEMPO DE COMPILAÇÃO

Suponha que executemos o cling em uma máquina para a qual um int seja representado em 4 bytes.

```
[cling]$ int v[] = { 42, 18, 99 };
[cling]$ sizeof(v[0])
(unsigned long) 4
[cling]$ sizeof(v)
(unsigned long) 12
[cling]$ // como determinar tamanho de v?
[cling]$ sizeof(v) / sizeof(v[0])
(unsigned long) 3
```

TAMANHO DE UM VETOR EM TEMPO DE COMPILAÇÃO

Os resultados são diferentes para ponteiros

```
[cling]$ int *w = (int*)malloc(3*sizeof(int));

[cling]$ w[0] = 42; w[1] = 18; w[2] = 99;

[cling]$ sizeof(w[0])
(unsigned long) 4 //  igual a sizeof(int)

[cling]$ sizeof(w)
(unsigned long) ?? // não é 12
```

VETORES versus PONTEIROS

```
int v[] = ...      int *p = ...
```

SEMELHANÇAS no acesso e na organização:

- Primeiro elemento:
  v[0] ou *v
  p[0] ou *p

- Acesso ao (i+1)-ésimo elto:
  v[i] ou *(v+i)
  p[i] ou *(p+i)

- Elementos armazenados em posições subsequentes na memória

VETORES versus PONTEIROS

```
int v[] = ...      int *p = ...
```

DIFERENÇAS:

- sizeof(v) dá a quantidade de bytes ocupados na memória pelo vetor. Logo, sizeof(v) / sizeof(v[0]) dará o número de elementos do vetor
- o único jeito de determinar o número de elementos de p é colocar essa quantidade em uma outra variável que sempre deve “acompanhar” p.
- gerência de memória para vetor é automática, para ponteiros é manual

MATRIZES

C possui sintaxe para declarar matrizes

tipo nomeMatriz[*n*][*m*];

Declara uma matriz de *n* linhas por *m* colunas.

Esta declaração aloca automaticamente espaço para *n* × *m* elementos, ou seja, *n***m**sizeof(**tipo**) bytes.

Os elementos de uma matriz declarada da forma acima ocupam, garantidamente, posições subsequentes na memória.

MATRIZES

Podemos também declarar e inicializar uma matriz. Por exemplo:

```
int m[2][3] = { { 18, 33, 99 },
                { 42, 21, 70 } };
```

Mesmo na atribuição com inicialização, é obrigatório declarar as dimensões.

Podemos acessar os elementos por meio de seus índices, que **sempre começam em 0**.

Após declarada a matriz acima, o elemento na segunda linha e primeira coluna é acessado por m[1][0].

MATRIZES

```
[cling]$ int m[2][3] = { { 18, 33, 99 },
                         { 42, 21, 70 }
                       };
[cling]$
```

MATRIZES

```
[cling]$ int m[2][3] = { { 18, 33, 99 },
                         { 42, 21, 70 }
                       };
[cling]$ m[1][0]
```

MATRIZES

```
[cling]$ int m[2][3] = { { 18, 33, 99 },
                         { 42, 21, 70 }
                       };
[cling]$ m[1][0]
(int) 42
[cling]$
```

MATRIZES

```
[cling]$ int m[2][3] = { { 18, 33, 99 },
                         { 42, 21, 70 }
                       };
[cling]$ m[1][0]
(int) 42
[cling]$ m[2][2]
```

MATRIZES

```
[cling]$ int m[2][3] = { { 18, 33, 99 },
                         { 42, 21, 70 }
                       };
[cling]$ m[1][0]
(int) 42
[cling]$ m[2][2] // elemento inválido
```

MATRIZES

```
[cling]$ int m[2][3] = { { 18, 33, 99 },
                         { 42, 21, 70 }
                       };
[cling]$ m[1][0]
(int) 42
[cling]$
```

MATRIZES

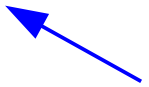
```
[cling]$ int m[2][3] = { { 18, 33, 99 },
                         { 42, 21, 70 }
                       };
[cling]$ m[1][0]
(int) 42
[cling]$ sizeof(m) / sizeof(m[0][0])
```

MATRIZES

```
[cling]$ int m[2][3] = { { 18, 33, 99 },
                         { 42, 21, 70 }
                       };
[cling]$ m[1][0]
(int) 42
[cling]$ sizeof(m) / sizeof(m[0][0])
(unsigned long) 6
[cling]$
```

MATRIZES

```
[cling]$ int m[2][3] = { { 18, 33, 99 },
                         { 42, 21, 70 }
                       };
[cling]$ m[1][0]
(int) 42
[cling]$ sizeof(m) / sizeof(m[0][0])
(unsigned long) 6
[cling]$
```



Quantidade total de elementos na matriz

MATRIZES

```
[cling]$ int m[2][3] = { { 18, 33, 99 },
                         { 42, 21, 70 }
                       };
[cling]$ m[1][0]
(int) 42
[cling]$ sizeof(m) / sizeof(m[0][0])
(unsigned long) 6
[cling]$ sizeof(m[0]) / sizeof(m[0][0])
```

MATRIZES

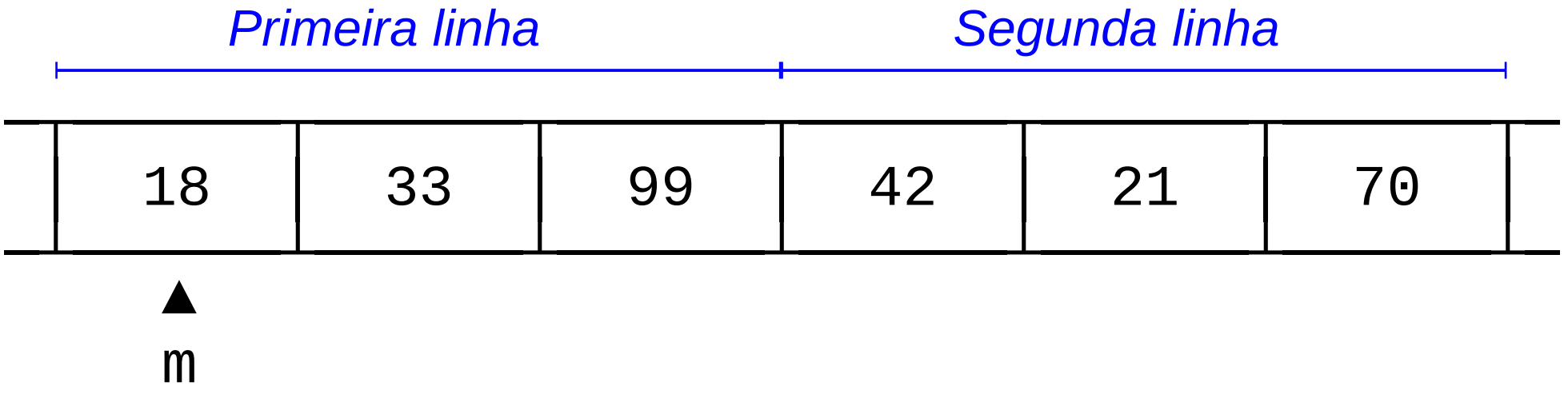
```
[cling]$ int m[2][3] = { { 18, 33, 99 },
                         { 42, 21, 70 }
                       };
[cling]$ m[1][0]
(int) 42
[cling]$ sizeof(m) / sizeof(m[0][0])
(unsigned long) 6
[cling]$ sizeof(m[0]) / sizeof(m[0][0])
(unsigned long) 3
[cling]$
```

Quantidade de colunas

MATRIZES – organização em memória

Os elementos de uma matriz sempre ocupam posições contíguas na memória, começando pelo primeiro elemento da primeira linha.

```
int m[2][3] = { { 18, 33, 99 },
                { 42, 21, 70 } };
```



MATRIZES

```
[cling]$ int m[2][3]= { { 18, 33, 99 },
                        { 42, 21, 70 } };
[cling]$ int *p = (int *)m;
```

Converte para o tipo ponteiro para inteiro

MATRIZES

```
[cling]$ int m[2][3]= { { 18, 33, 99 },
                        { 42, 21, 70 } };
[cling]$ int *p = (int *)m;
```

MATRIZES

```
[cling]$ int m[2][3]= { { 18, 33, 99 },
                        { 42, 21, 70 } };
[cling]$ int *p = (int *)m;
[cling]$ p[0]
```

MATRIZES

```
[cling]$ int m[2][3]= { { 18, 33, 99 },
                        { 42, 21, 70 } };
[cling]$ int *p = (int *)m;
[cling]$ p[0]
(int) 18
[cling]$
```

MATRIZES

```
[cling]$ int m[2][3]= { { 18, 33, 99 },
                        { 42, 21, 70 } };
[cling]$ int *p = (int *)m;
[cling]$ p[0]
(int) 18
[cling]$ p[2]
```

MATRIZES

```
[cling]$ int m[2][3]= { { 18, 33, 99 },
                        { 42, 21, 70 } };
[cling]$ int *p = (int *)m;
[cling]$ p[0]
(int) 18
[cling]$ p[2]
(int) 99
[cling]$
```

MATRIZES

```
[cling]$ int m[2][3]= { { 18, 33, 99 },
                        { 42, 21, 70 } };
[cling]$ int *p = (int *)m;
[cling]$ p[0]
(int) 18
[cling]$ p[2]
(int) 99
[cling]$ p[3]
```

MATRIZES

```
[cling]$ int m[2][3]= { { 18, 33, 99 },
                        { 42, 21, 70 } };
[cling]$ int *p = (int *)m;
[cling]$ p[0]
(int) 18
[cling]$ p[2]
(int) 99
[cling]$ p[3]
(int) 42
[cling]$
```

MATRIZES

```
[cling]$ int m[2][3]= { { 18, 33, 99 },
                        { 42, 21, 70 } };
[cling]$ int *p = (int *)m;
[cling]$ p[0]
(int) 18
[cling]$ p[2]
(int) 99
[cling]$ p[3]
(int) 42
[cling]$ p[4]
(int) 21
[cling]$
```

MATRIZES – organização em memória

A notação de matriz é apenas um auxílio de sintaxe para facilitar a vida do programador (*syntactic sugar*).

Declarar uma matriz com

```
tipo m[nlin][ncol];
```

e acessar os elementos com `m[i][j]`...

Equivale a declarar um vetor

```
tipo m[nlin*ncol];
```

e acessar os elementos por meio da expressão `m[i*nlin+j]`.

MATRIZES – organização em memória

A notação de matriz é apenas um auxílio de sintaxe para facilitar a vida do programador (*syntactic sugar*).

Declarar uma matriz com

```
tipo m[nlin][ncol];
```

e acessar os elementos com `m[i][j]`...

Equivale a declarar um vetor

```
tipo m[nlin*ncol];
```

e acessar os elementos por meio da expressão `m[i*nlin+j]`.

MATRIZES – passagem como parâmetro

Para passar uma matriz para uma função, devemos declarar as suas dimensões. Se as dimensões forem variáveis, os parâmetros devem ser declarados antes.

```
void transfProjetiva2D(double matriz[3][3],
                       double vetor[2],
                       double resultado[2]);

void transpoe(int nlin, int ncol,
              double matriz[nlin][ncol],
              double resultado[ncol][nlin]);
```

MATRIZES – passagem como parâmetro

Uma matriz é sempre passada como referência (ponteiro) para função.

Ou seja, na declaração:

func(**tipo** *matriz*[n][m])

A matriz é passada, implicitamente, como **tipo ***

Implicação: dentro da função, sizeof(*matriz*) equivale a sizeof(**tipo ***)

MATRIZES – retorno de função

Assim como ocorre com vetores, não existe sintaxe para retornar uma matriz de uma função.

Para isto, precisamos usar ponteiros com espaço alocado no heap.

Uma estratégia é alocar espaço contíguo e declarar retorno como **tipo ***

Desvantagem: seremos obrigados a acessar o elemento da linha i, coluna j usando [i*ncol + j].

Para evitar este inconveniente e usar a notação [i][j], podemos retornar um **ponteiro para ponteiro**.

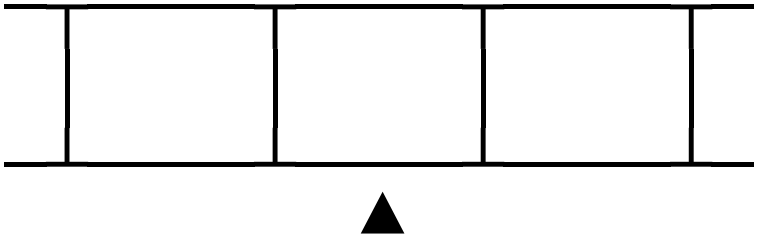
PONTEIROS para PONTEIROS

Um ponteiro para ponteiro, ou ponteiro duplo, é uma variável que armazena uma referência para uma outra variável, que por sua vez armazena outra referência.

PONTEIROS para PONTEIROS

Um ponteiro para ponteiro, ou ponteiro duplo, é uma variável que armazena uma referência para uma outra variável, que por sua vez armazena outra referência.

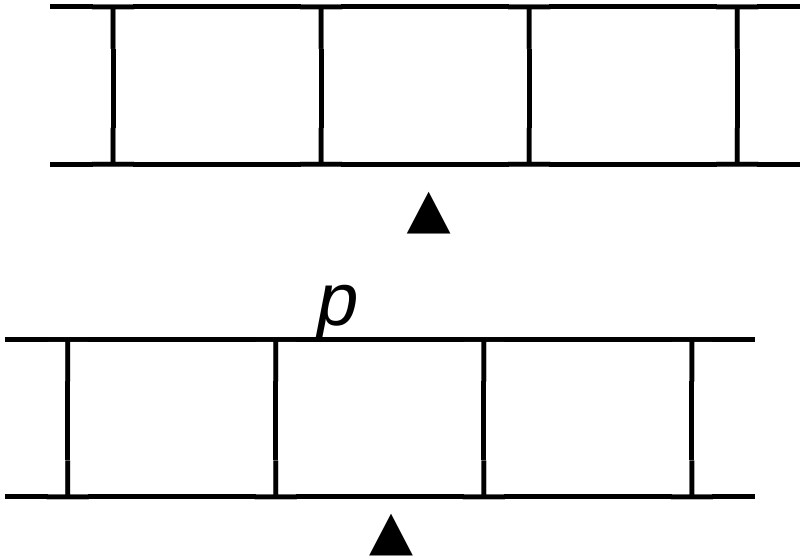
```
(tipo *)malloc(sizeof(tipo));
```



PONTEIROS para PONTEIROS

Um ponteiro para ponteiro, ou ponteiro duplo, é uma variável que armazena uma referência para uma outra variável, que por sua vez armazena outra referência.

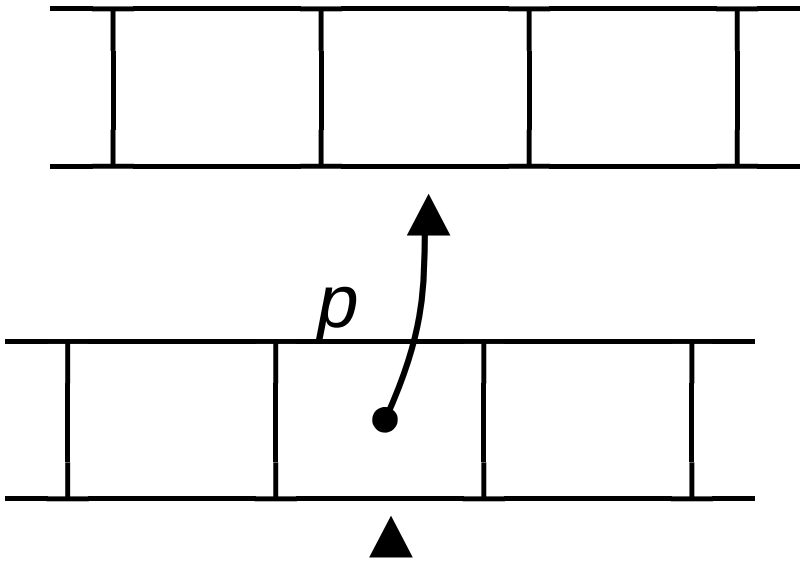
tipo ***p**



PONTEIROS para PONTEIROS

Um ponteiro para ponteiro, ou ponteiro duplo, é uma variável que armazena uma referência para uma outra variável, que por sua vez armazena outra referência.

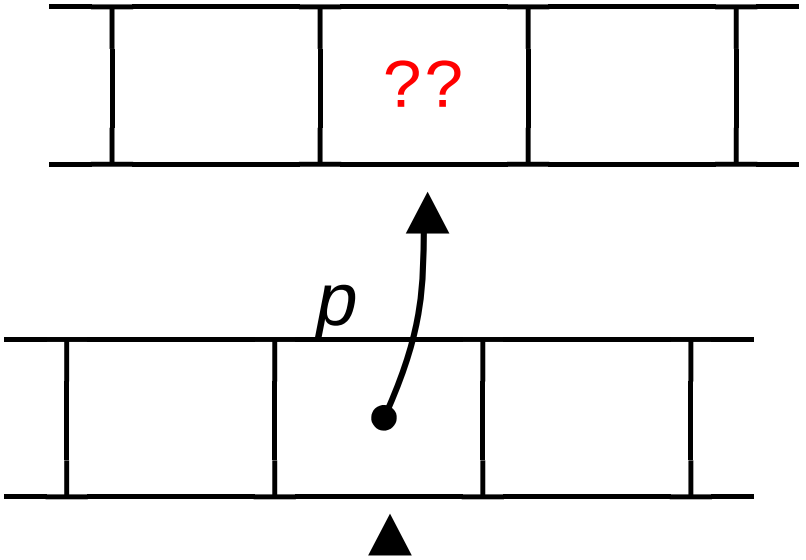
```
tipo *p = (tipo *)malloc(sizeof(tipo));
```



PONTEIROS para PONTEIROS

Um ponteiro para ponteiro, ou ponteiro duplo, é uma variável que armazena uma referência para uma outra variável, que por sua vez armazena outra referência.

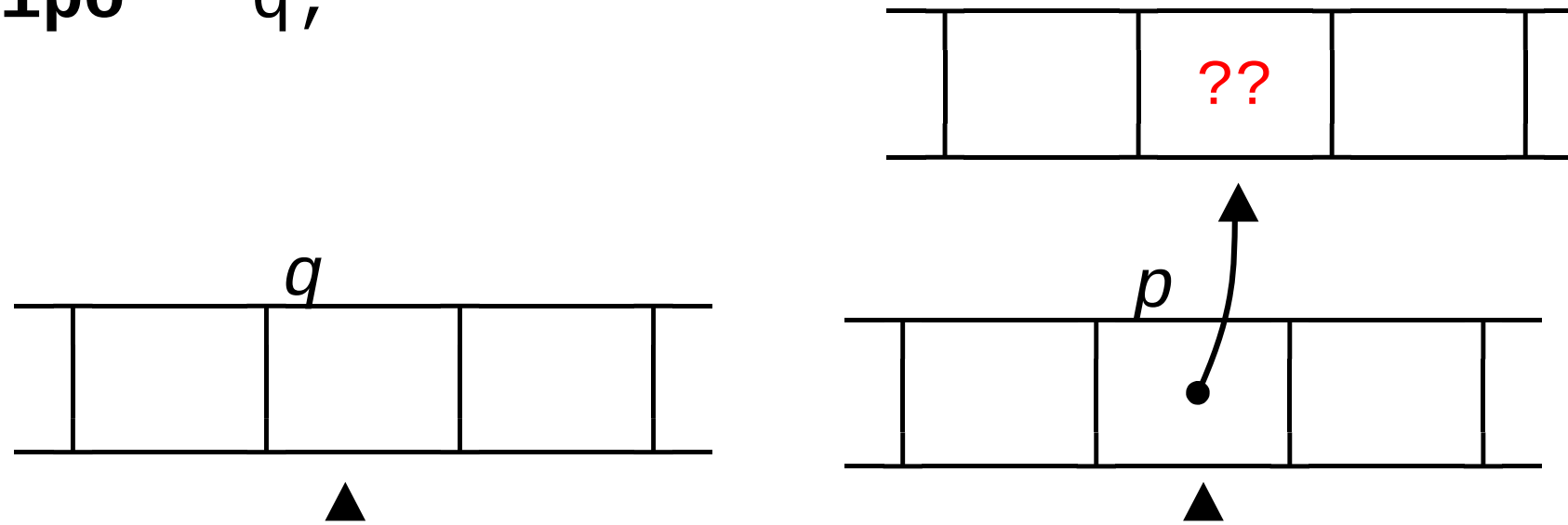
```
tipo *p = (tipo *)malloc(sizeof(tipo));
*p = ??;
```



PONTEIROS para PONTEIROS

Um ponteiro para ponteiro, ou ponteiro duplo, é uma variável que armazena uma referência para uma outra variável, que por sua vez armazena outra referência.

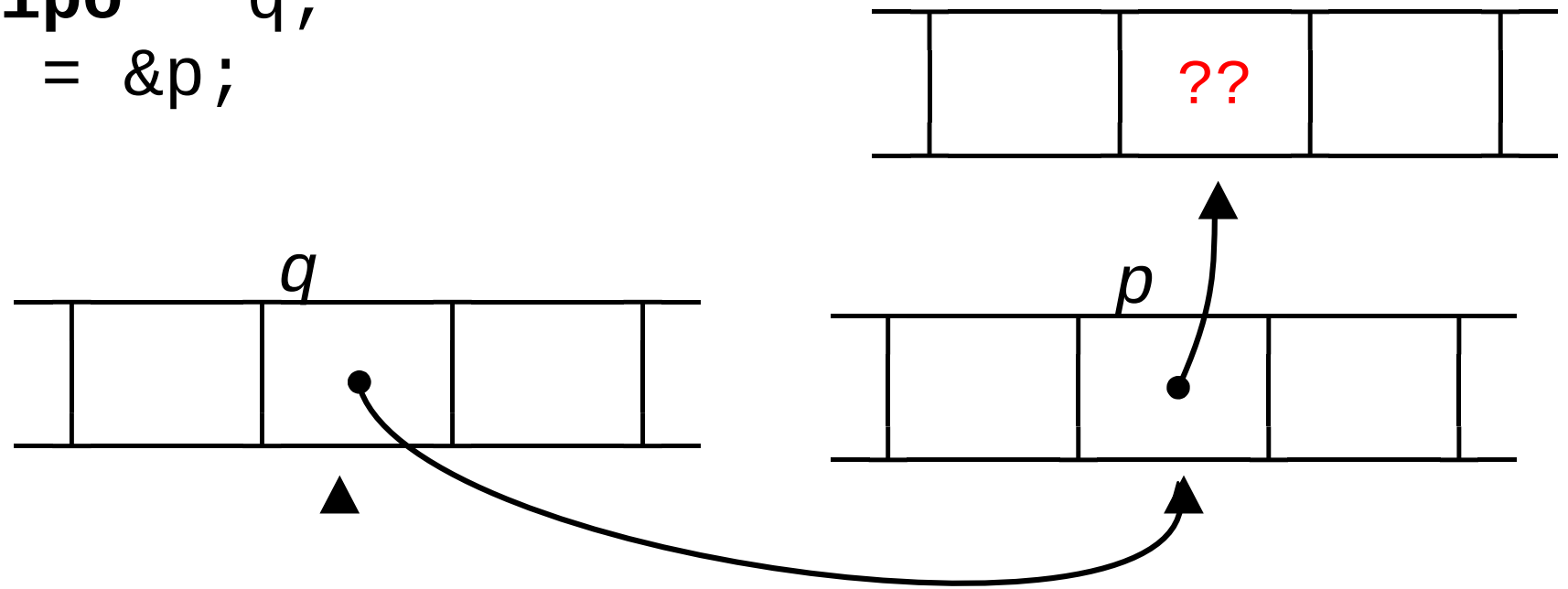
```
tipo *p = (tipo *)malloc(sizeof(tipo));
*p = ??;
tipo **q;
```



PONTEIROS para PONTEIROS

Um ponteiro para ponteiro, ou ponteiro duplo, é uma variável que armazena uma referência para uma outra variável, que por sua vez armazena outra referência.

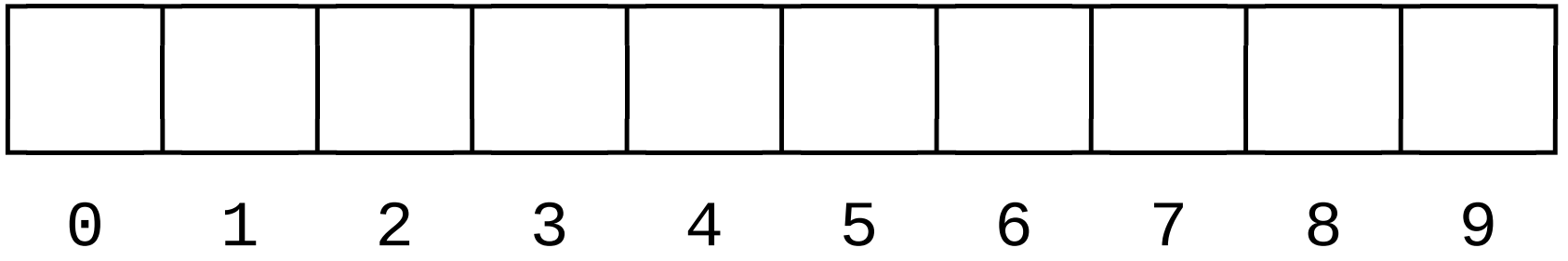
```
tipo *p = (tipo *)malloc(sizeof(tipo));
*p = ??;
tipo **q;
q = &p;
```



PONTEIROS para PONTEIROS

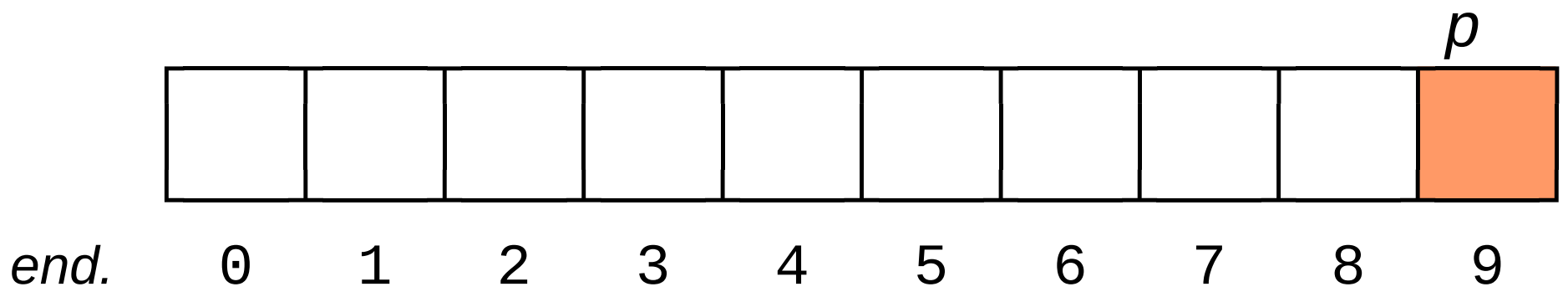
Exemplo: suponha que estamos trabalhando com ponteiros para char em uma memória com 10 bytes e cada ponteiro ocupa 1 byte.

Suponha que a pilha esteja no final da memória e cresça em direção ao endereço 0. O heap está no começo da memória, a partir do endereço 1.



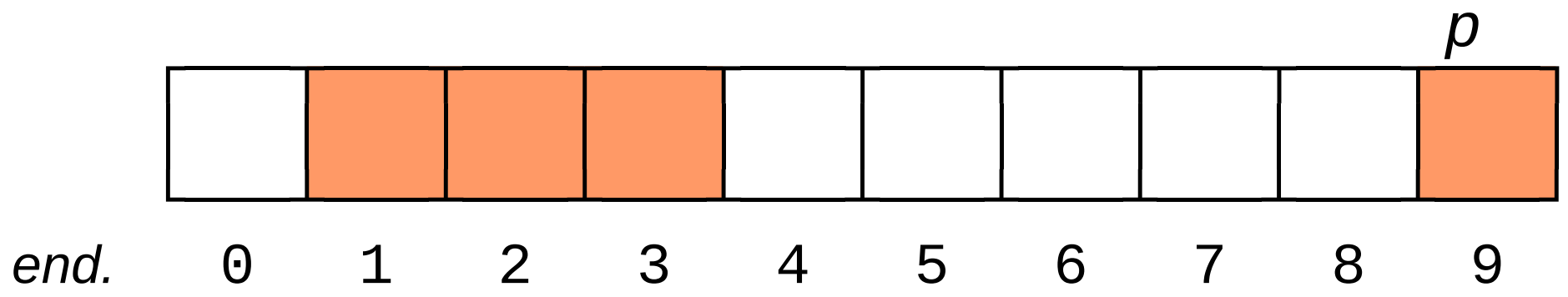
PONTEIROS para PONTEIROS

```
char *p;
```



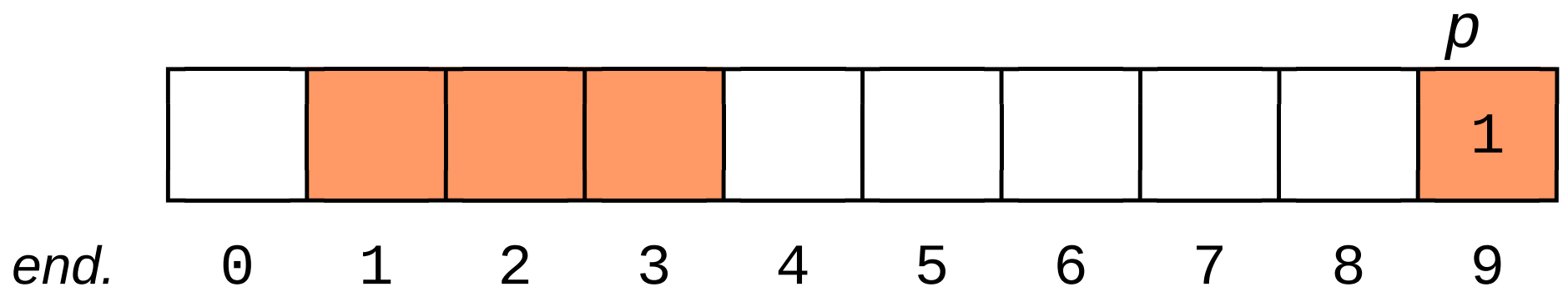
PONTEIROS para PONTEIROS

```
char *p;
        malloc(3*sizeof(char));
```



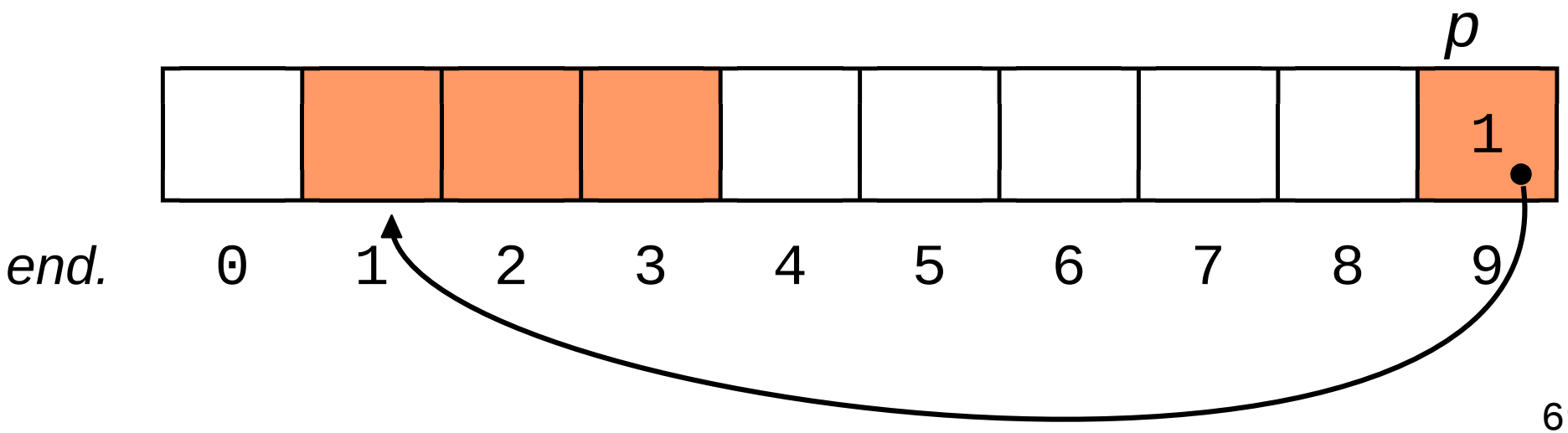
PONTEIROS para PONTEIROS

```
char *p;
p = (char *)malloc(3*sizeof(char));
```



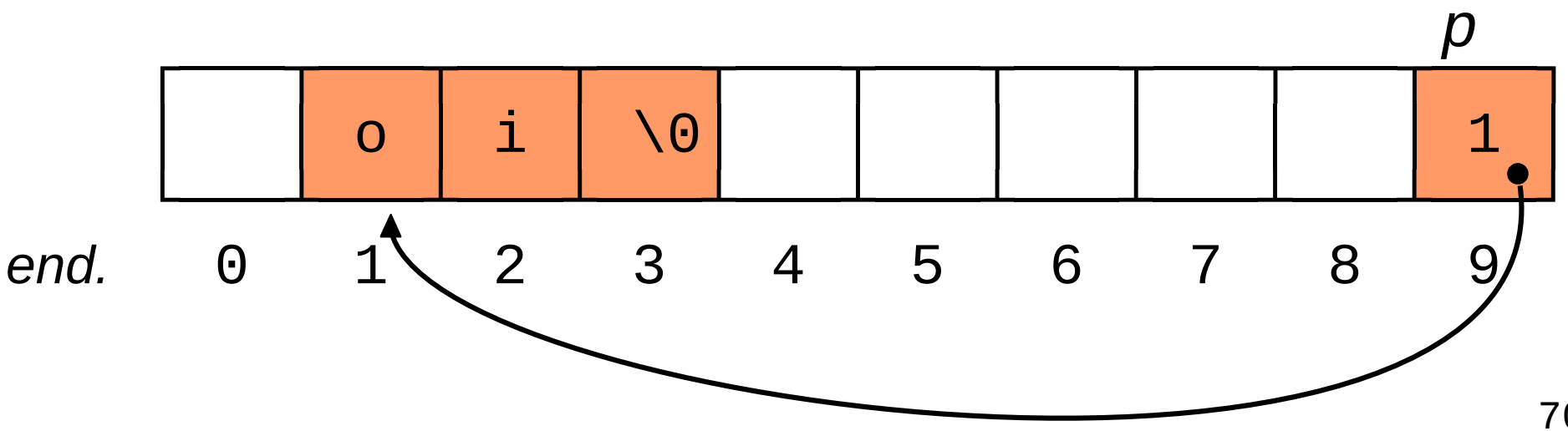
PONTEIROS para PONTEIROS

```
char *p;
p = (char *)malloc(3*sizeof(char));
```



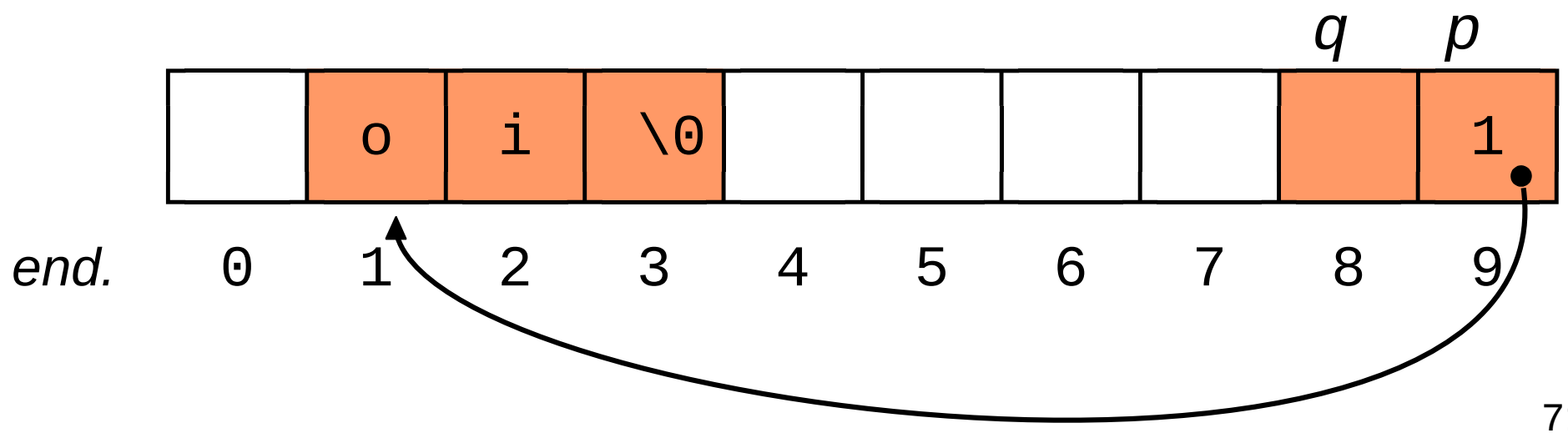
PONTEIROS para PONTEIROS

```
char *p;
p = (char *)malloc(3*sizeof(char));
p[0] = 'o'; p[1] = 'i'; p[2] = '\0';
```



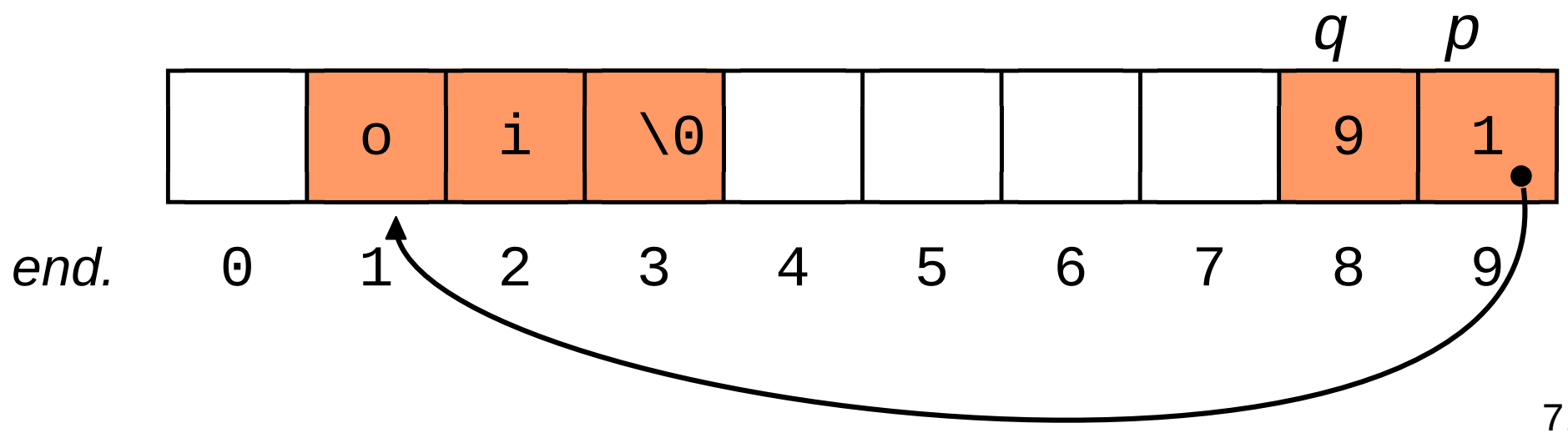
PONTEIROS para PONTEIROS

```
char *p;
p = (char *)malloc(3*sizeof(char));
p[0] = 'o'; p[1] = 'i'; p[2] = '\0';
char **q;
```



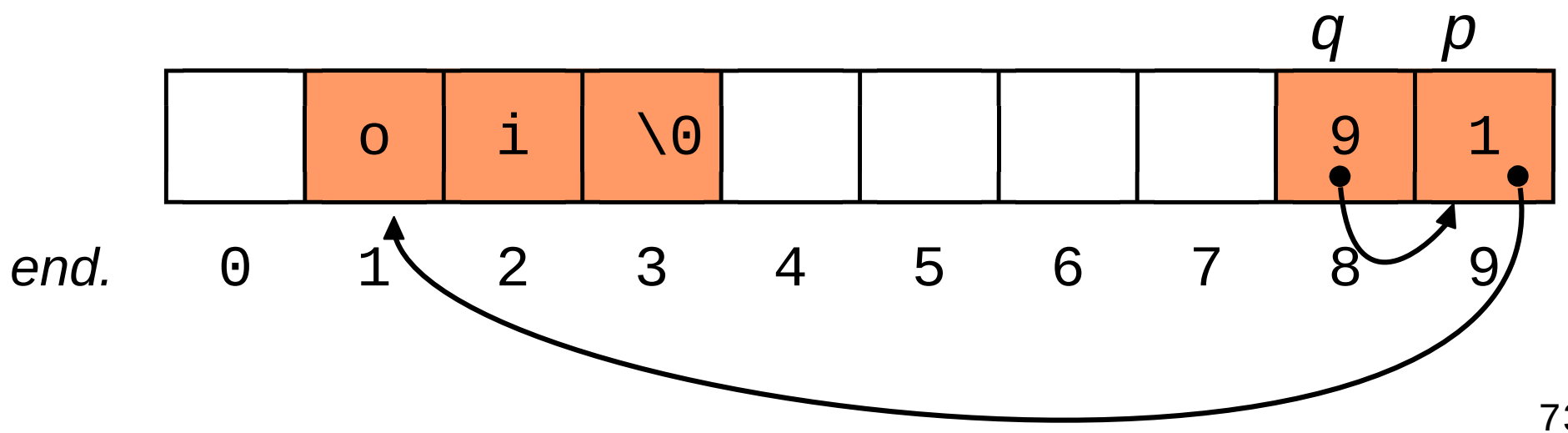
PONTEIROS para PONTEIROS

```
char *p;
p = (char *)malloc(3*sizeof(char));
p[0] = 'o'; p[1] = 'i'; p[2] = '\0';
char **q;
*q = &p;
```



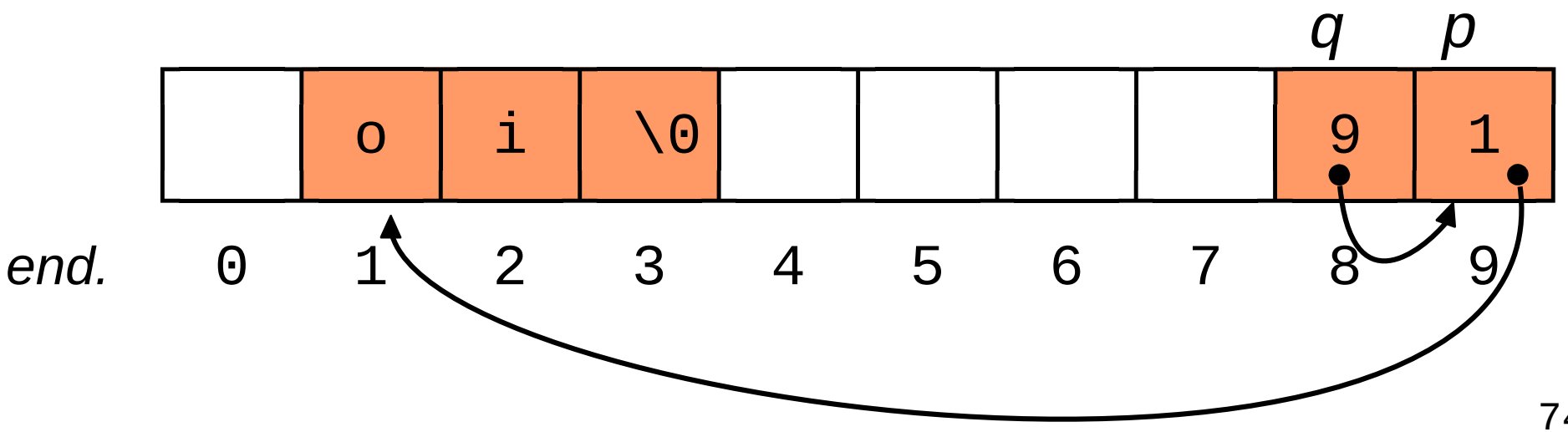
PONTEIROS para PONTEIROS

```
char *p;
p = (char *)malloc(3*sizeof(char));
p[0] = 'o'; p[1] = 'i'; p[2] = '\0';
char **q;
*q = &p;
```



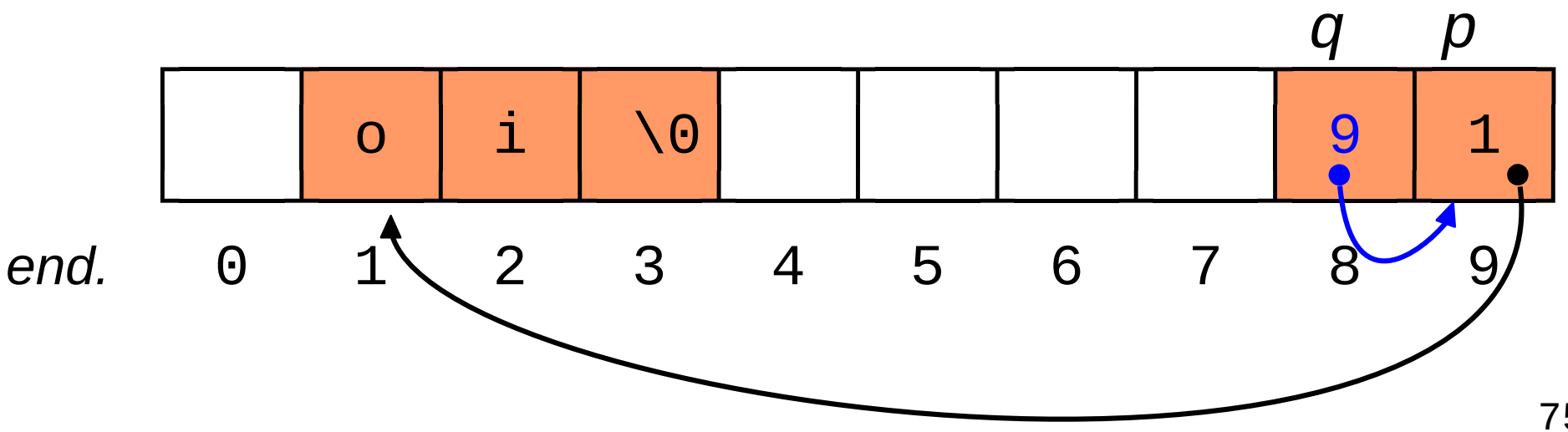
PONTEIROS para PONTEIROS

```
char *p;
p = (char *)malloc(3*sizeof(char));
p[0] = 'o'; p[1] = 'i'; p[2] = '\0';
char **q;
*q = &p;
**q = 'h';
```



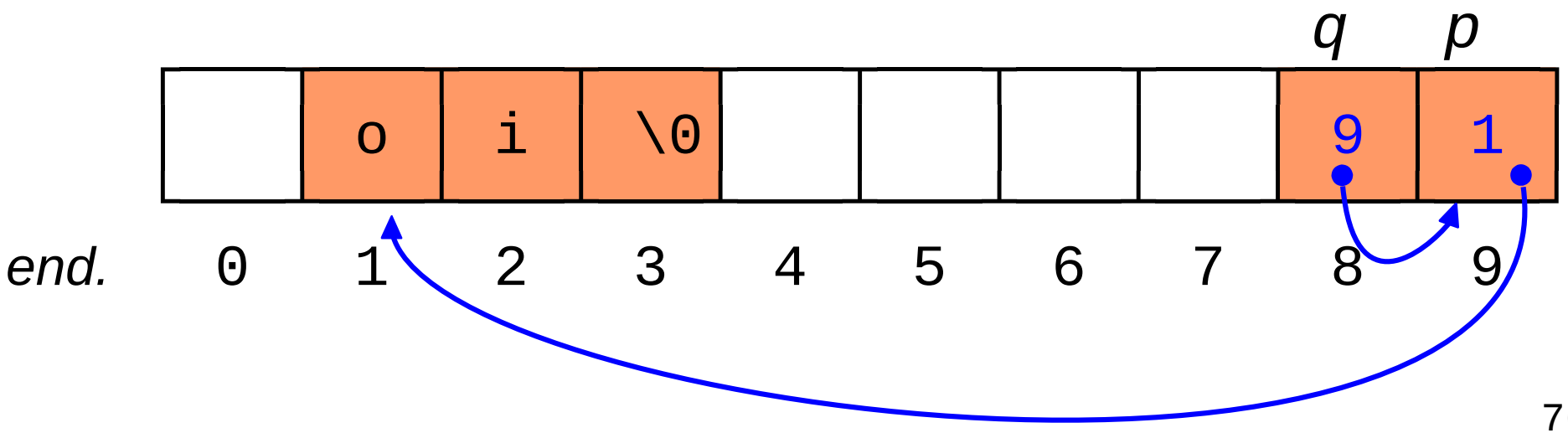
PONTEIROS para PONTEIROS

```
char *p;
p = (char *)malloc(3*sizeof(char));
p[0] = 'o'; p[1] = 'i'; p[2] = '\0';
char **q;
*q = &p;
**q = 'h';
```



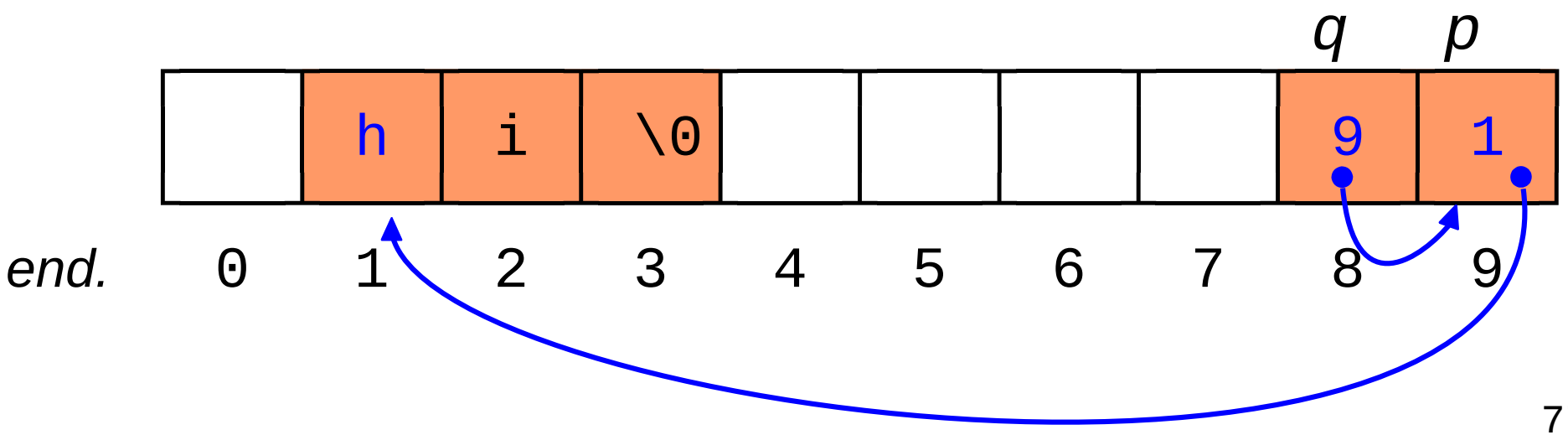
PONTEIROS para PONTEIROS

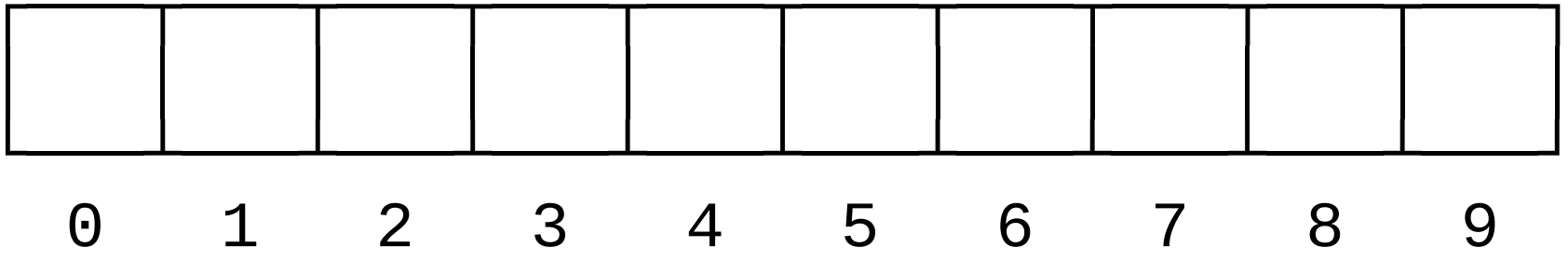
```
char *p;
p = (char *)malloc(3*sizeof(char));
p[0] = 'o'; p[1] = 'i'; p[2] = '\0';
char **q;
*q = &p;
**q = 'h';
```



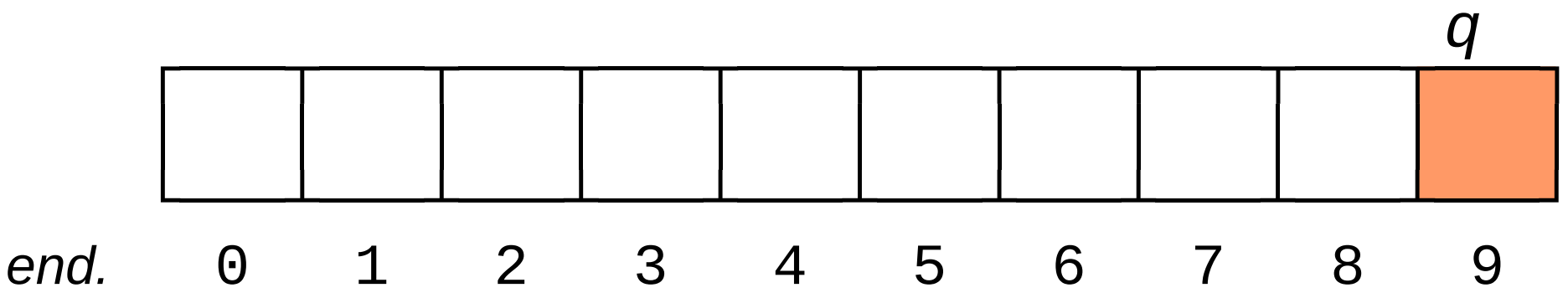
PONTEIROS para PONTEIROS

```
char *p;
p = (char *)malloc(3*sizeof(char));
p[0] = 'o'; p[1] = 'i'; p[2] = '\0';
char **q;
*q = &p;
**q = 'h';
```

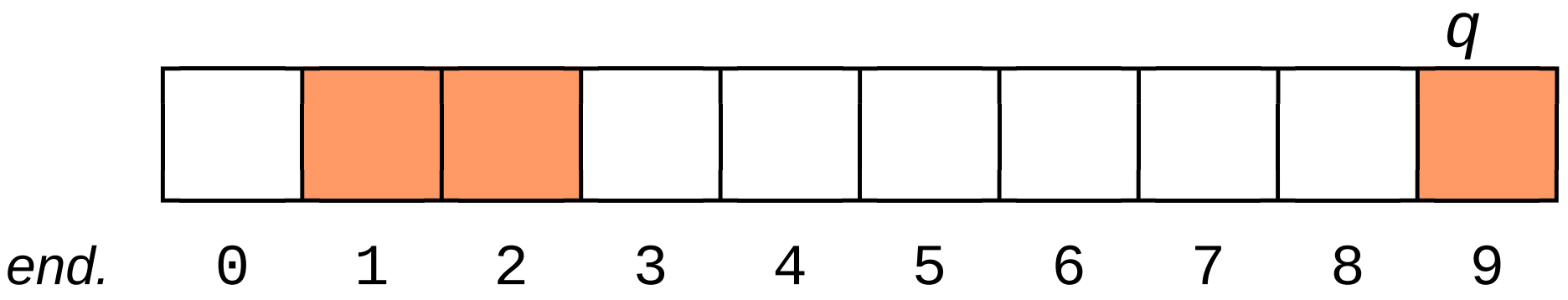



end.
0
1
2
3
4
5
6
7
8
9

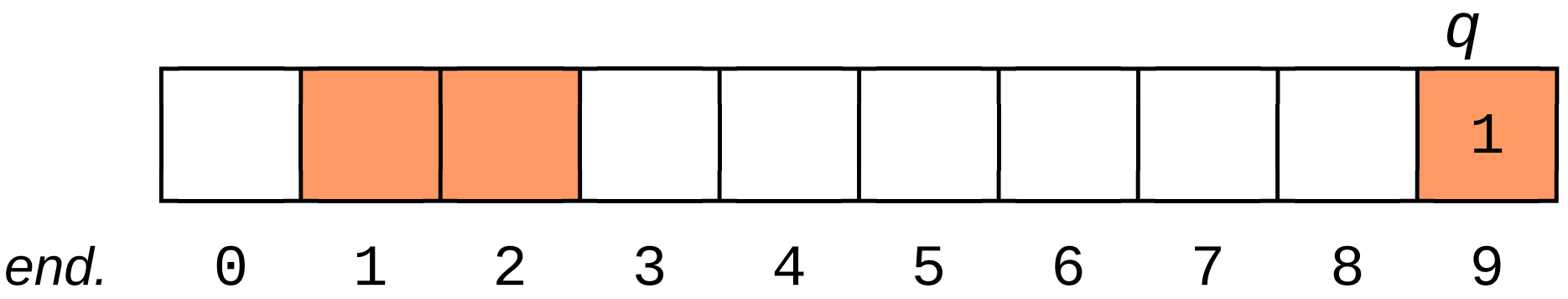
```
char **q;
```



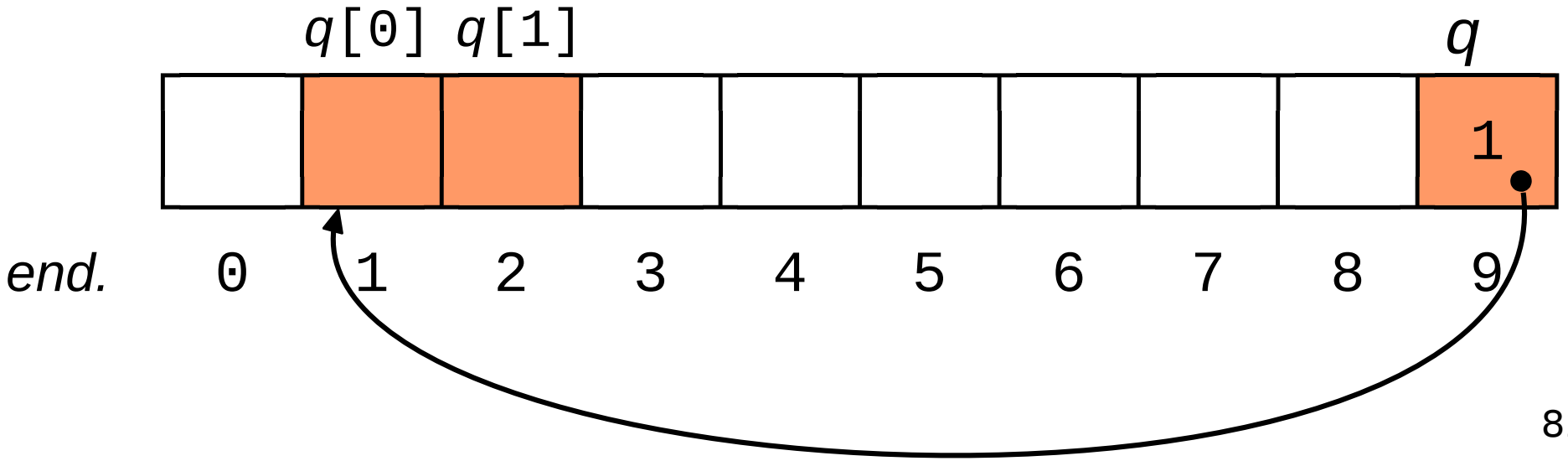
```
char **q;
                malloc(2*sizeof(char*));
```



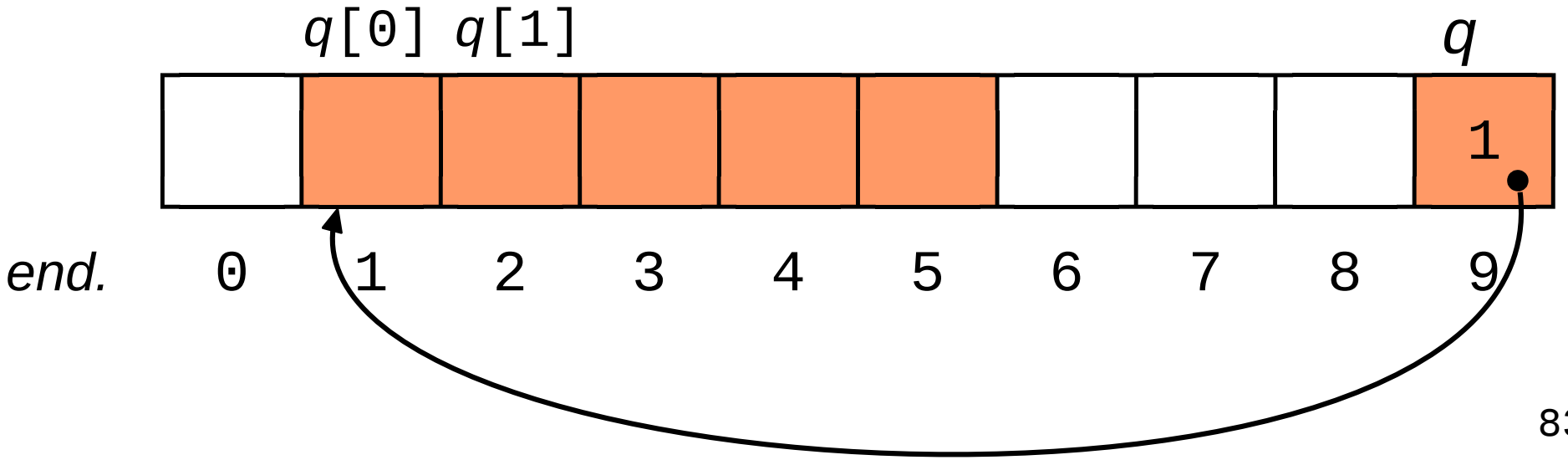
```
char **q;
q = (char**)malloc(2*sizeof(char*));
```



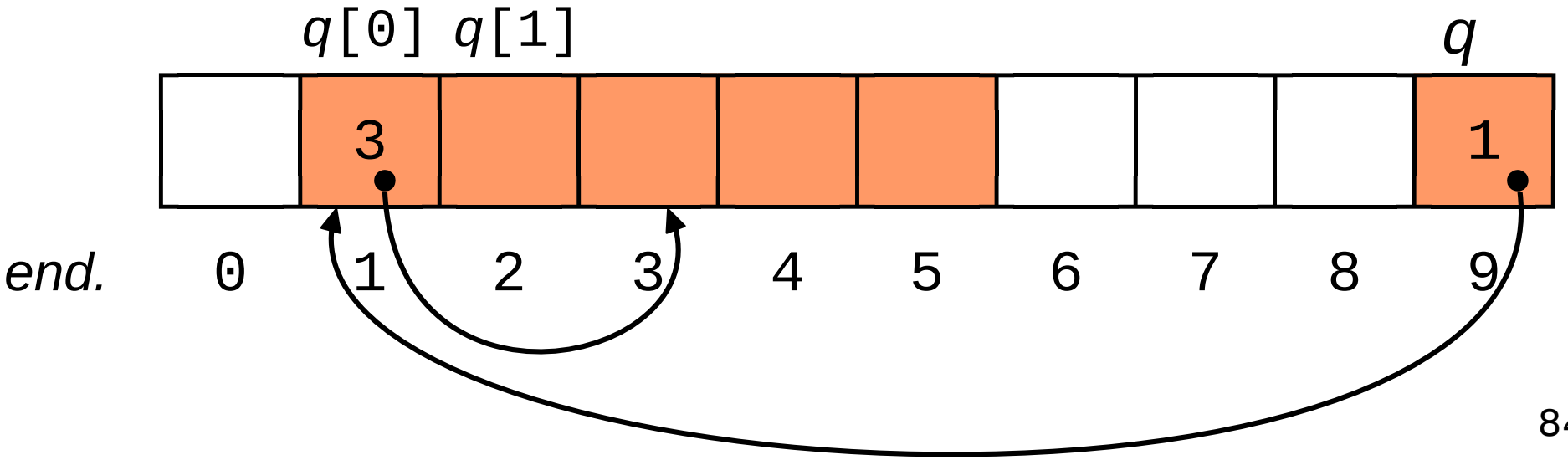
```
char **q;
q = (char**)malloc(2*sizeof(char*));
```



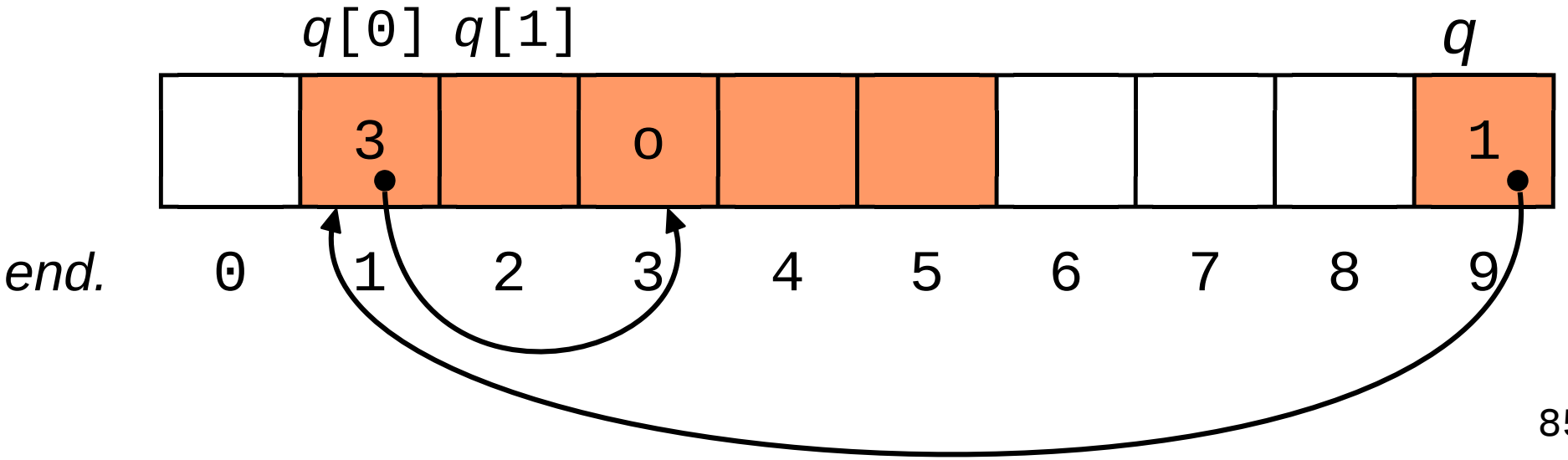
```
char **q;
q = (char**)malloc(2*sizeof(char*));
                malloc(3*sizeof(char));
```



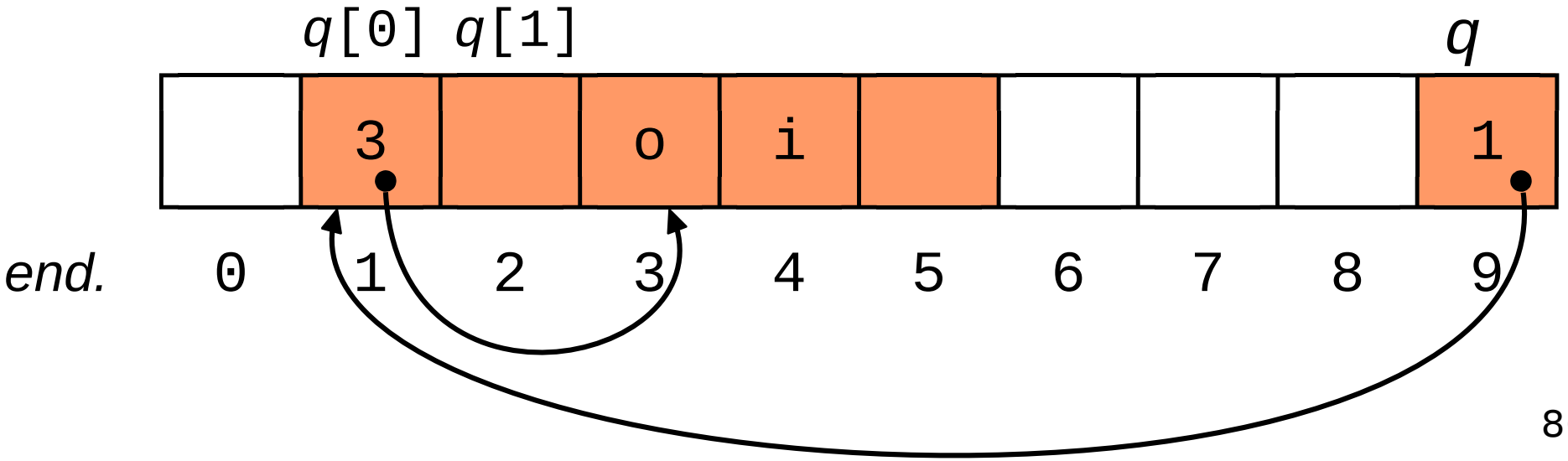
```
char **q;
q = (char**)malloc(2*sizeof(char*));
q[0] = (char*)malloc(3*sizeof(char));
```



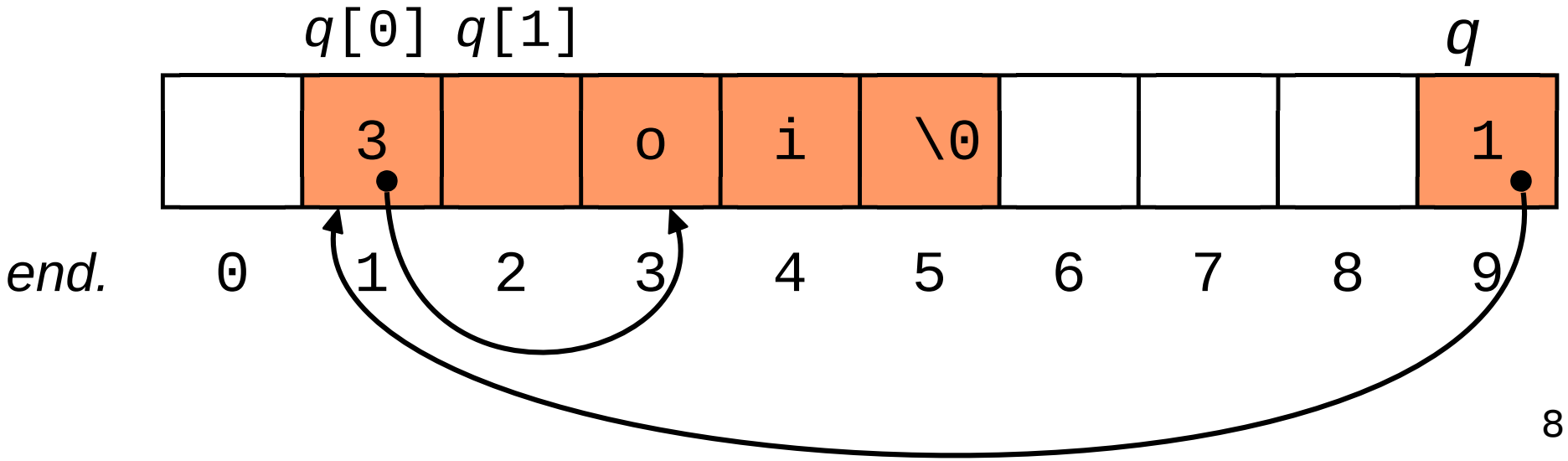
```
char **q;
q = (char**)malloc(2*sizeof(char*));
q[0] = (char*)malloc(3*sizeof(char));
q[0][0] = 'o';
```



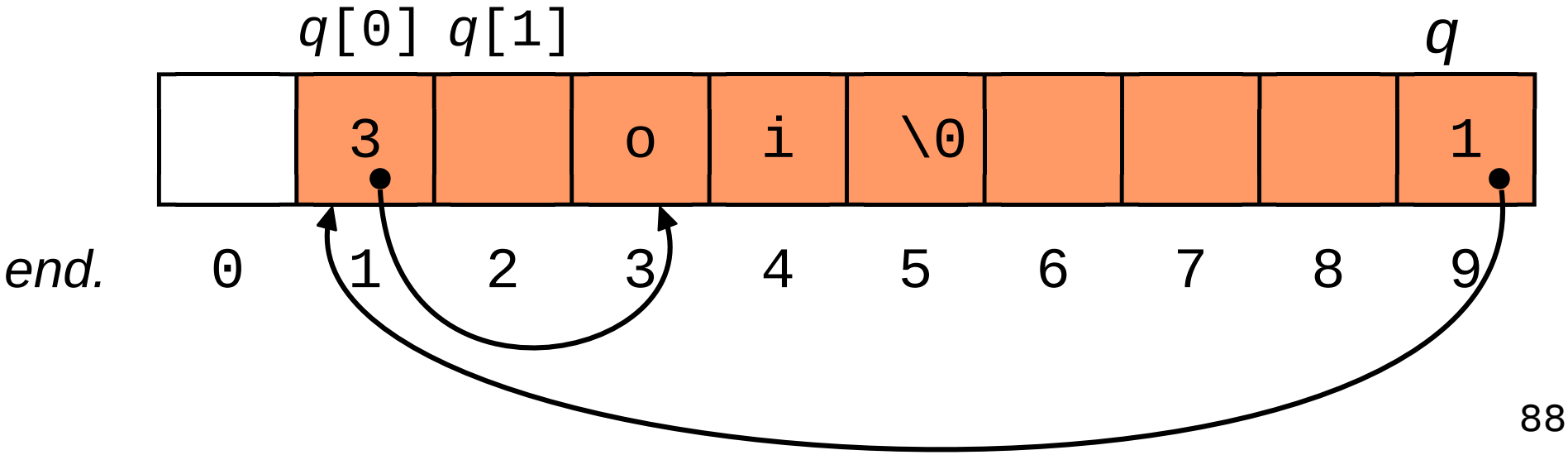
```
char **q;
q = (char**)malloc(2*sizeof(char*));
q[0] = (char*)malloc(3*sizeof(char));
q[0][0] = 'o'; q[0][1] = 'i';
```



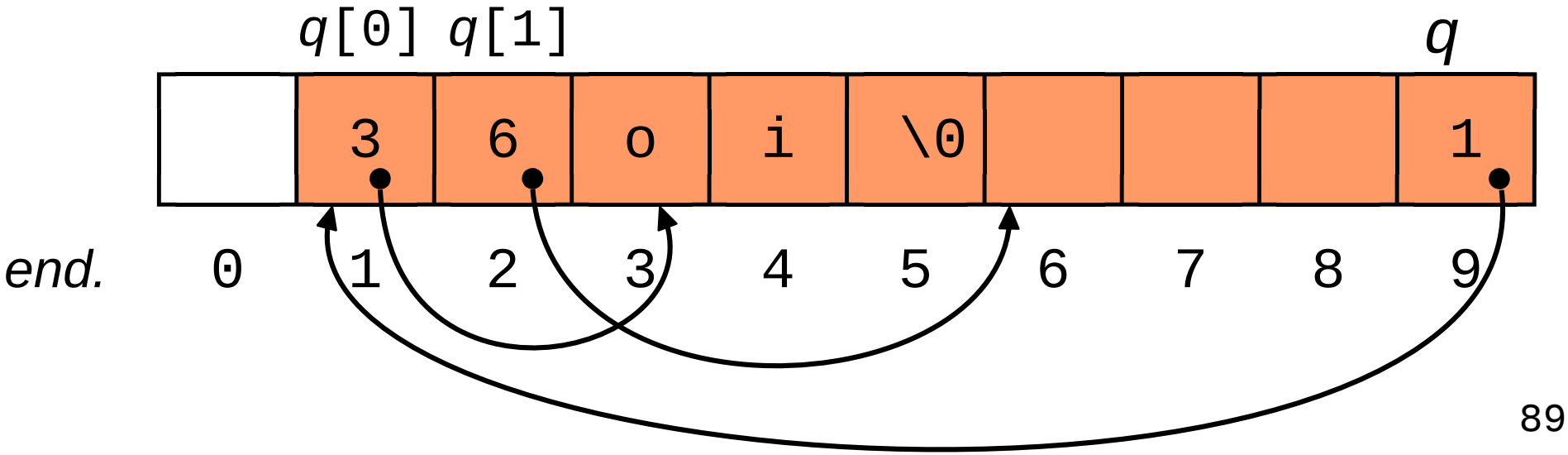
```
char **q;
q = (char**)malloc(2*sizeof(char*));
q[0] = (char*)malloc(3*sizeof(char));
q[0][0] = 'o'; q[0][1] = 'i';
q[0][2] = '\0';
```



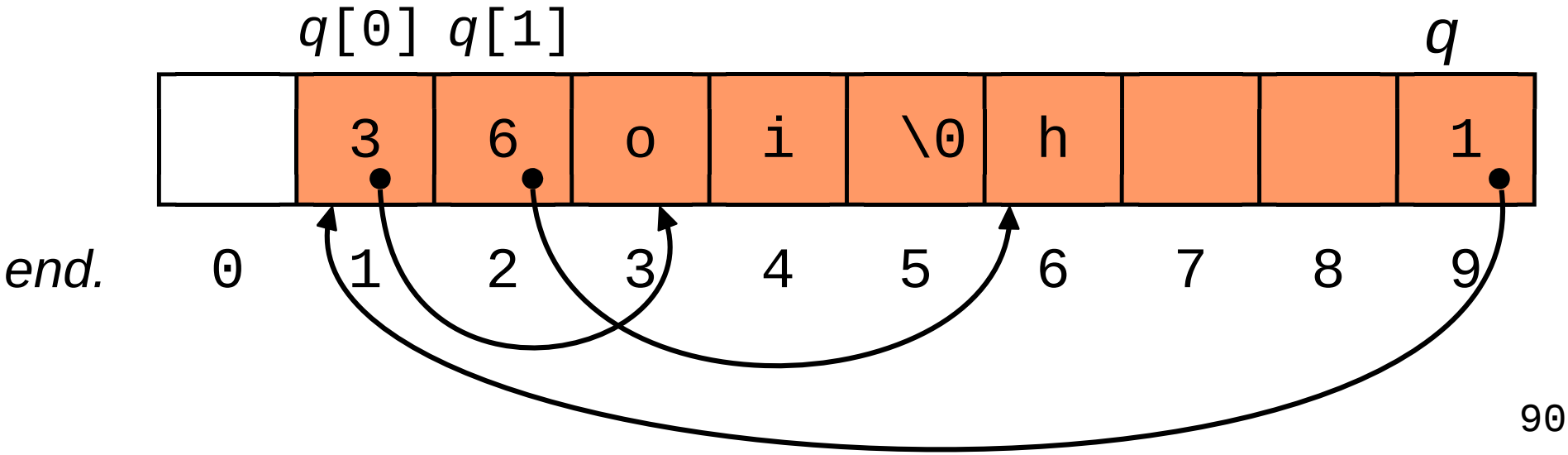
```
char **q;
q = (char**)malloc(2*sizeof(char*));
q[0] = (char*)malloc(3*sizeof(char));
q[0][0] = 'o'; q[0][1] = 'i';
q[0][2] = '\0';
                malloc(3*sizeof(char));
```



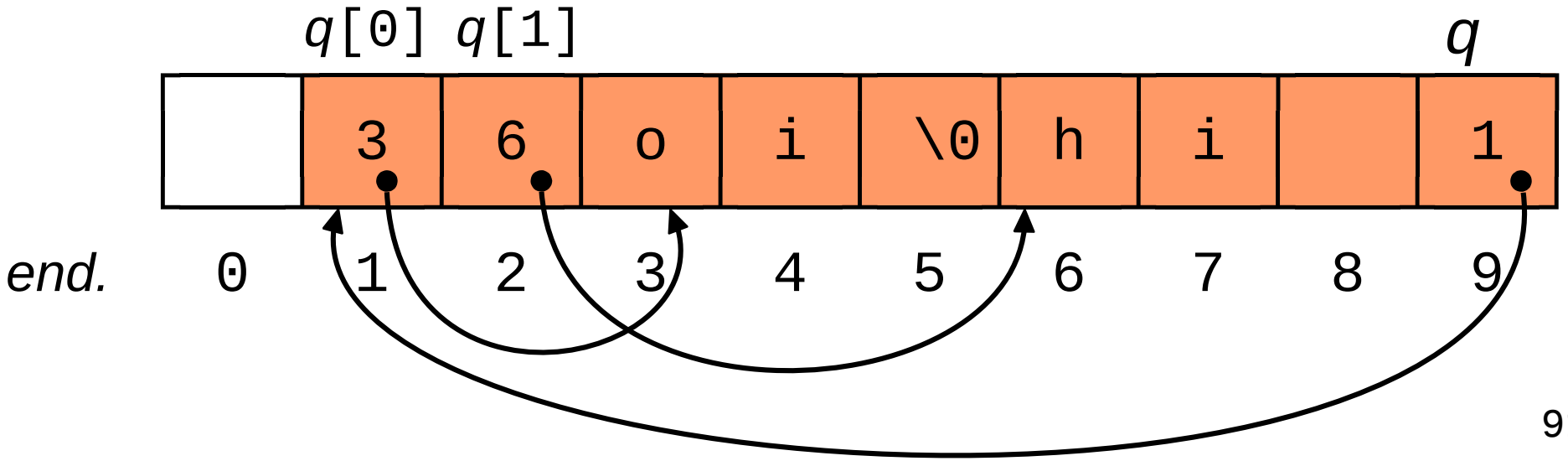
```
char **q;
q = (char**)malloc(2*sizeof(char*));
q[0] = (char*)malloc(3*sizeof(char));
q[0][0] = 'o'; q[0][1] = 'i';
q[0][2] = '\0';
q[1] = (char*)malloc(3*sizeof(char));
```



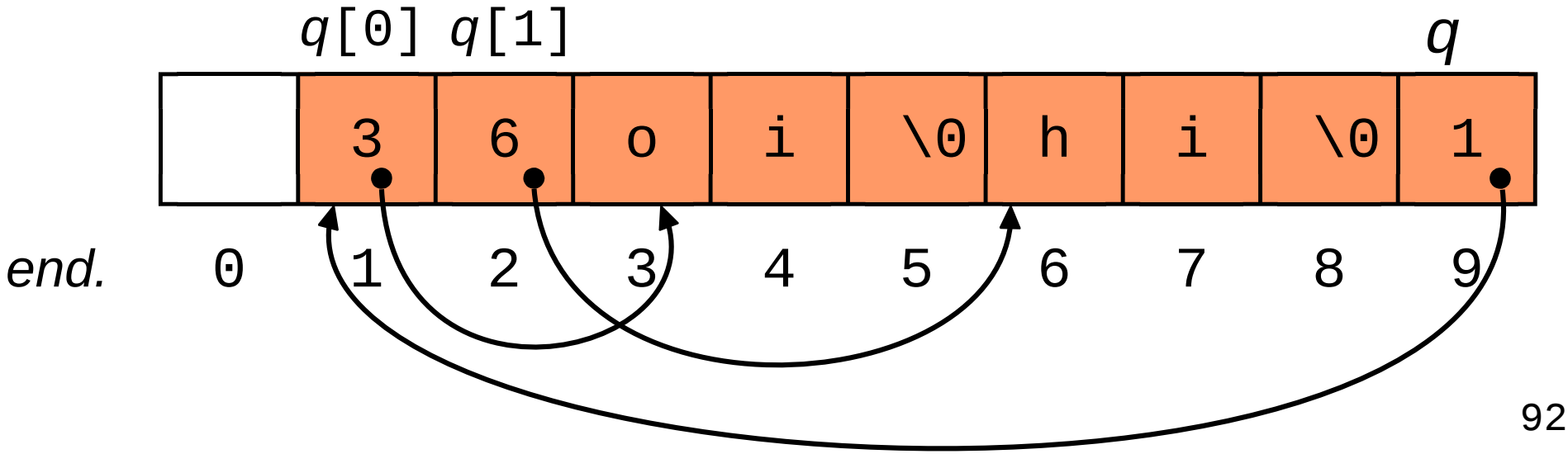
```
char **q;
q = (char**)malloc(2*sizeof(char*));
q[0] = (char*)malloc(3*sizeof(char));
q[0][0] = 'o'; q[0][1] = 'i';
q[0][2] = '\0';
q[1] = (char*)malloc(3*sizeof(char));
q[1][0] = 'h';
```



```
char **q;
q = (char**)malloc(2*sizeof(char*));
q[0] = (char*)malloc(3*sizeof(char));
q[0][0] = 'o'; q[0][1] = 'i';
q[0][2] = '\0';
q[1] = (char*)malloc(3*sizeof(char));
q[1][0] = 'h'; q[1][1] = 'i';
```



```
char **q;
q = (char**)malloc(2*sizeof(char*));
q[0] = (char*)malloc(3*sizeof(char));
q[0][0] = 'o'; q[0][1] = 'i';
q[0][2] = '\0';
q[1] = (char*)malloc(3*sizeof(char));
q[1][0] = 'h'; q[1][1] = 'i';
q[1][2] = '\0';
```



“MATRIZES” com PONTEIROS DUPLOS

A notação para acesso a elemento referenciado por ponteiro duplo pode ser aproveitada para “matrizes.”

M[i][j] → acessa o elemento na linha i, coluna j

(linhas/colunas começam no 0)

ATENÇÃO: apenas a notação é parecida com o conceito de matriz em C. A organização em memória será diferente!

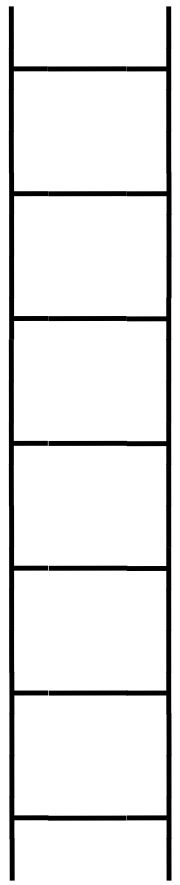
“MATRIZES” com PONTEIROS DUPLOS

Para construir uma “matriz” usando ponteiros duplos, devemos alocar manualmente espaço para cada linha da matriz.

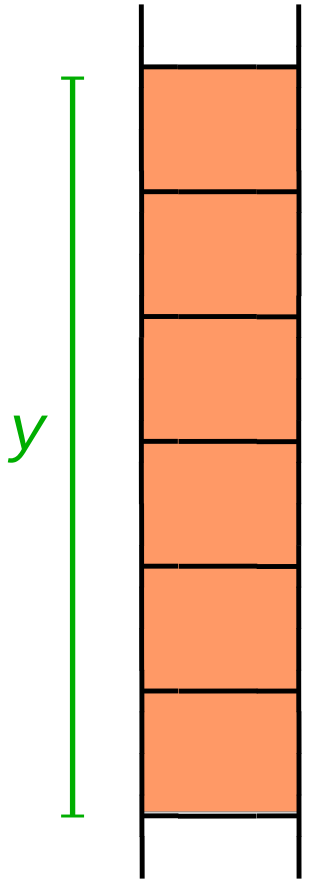
Ex.: alocando espaço para uma matriz M com *y* linhas e *x* colunas.

```
tipo **M = (tipo**)malloc(y*sizeof(tipo*));
for(int i = 0; i < y; ++i) {
  M[i] = (tipo*)malloc(x*sizeof(tipo));
}
```

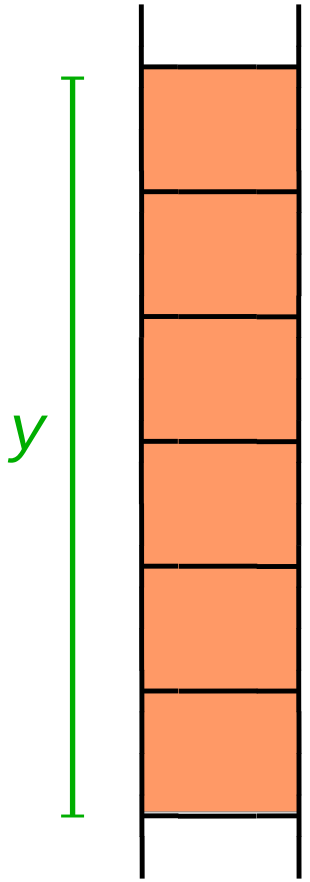
```
tipo **M = (tipo**)malloc(y*sizeof(tipo*));
for(int i = 0; i < y; ++i) {
  M[i] = (tipo*)malloc(x*sizeof(tipo));
}
```



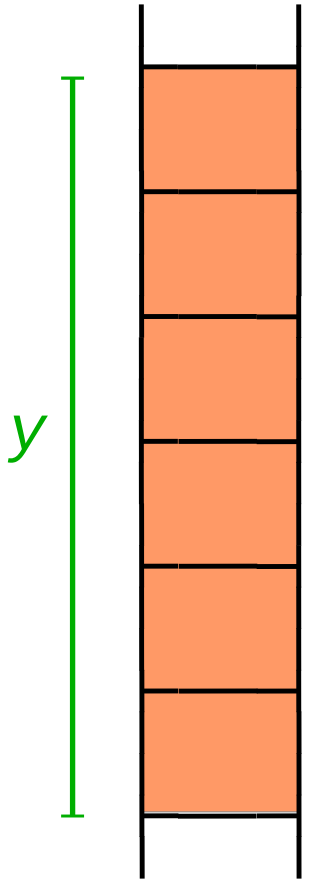
```
tipo **M = (tipo**)malloc(y*sizeof(tipo*));
for(int i = 0; i < y; ++i) {
  M[i] = (tipo*)malloc(x*sizeof(tipo));
}
```



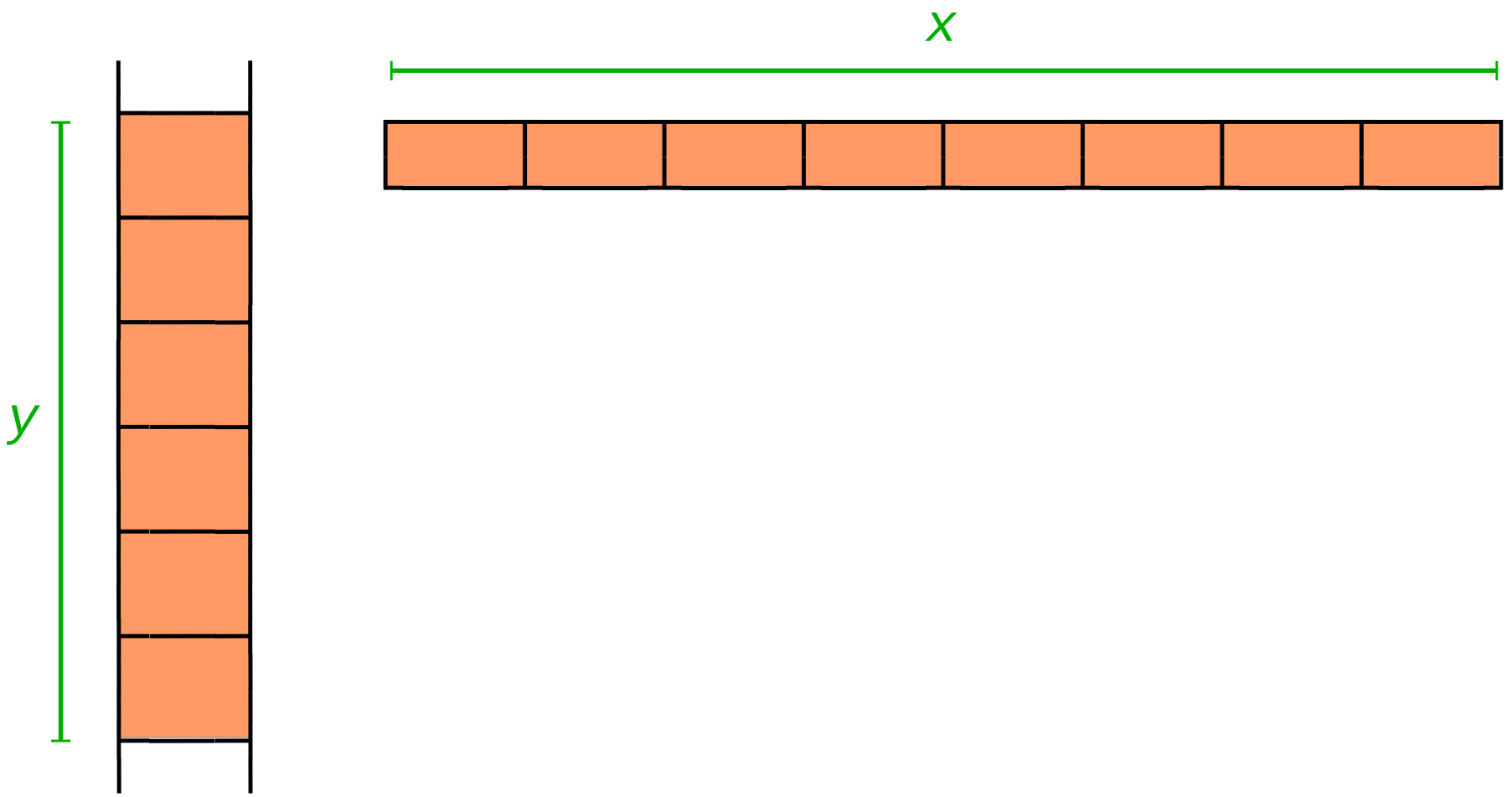
```
tipo **M = (tipo**)malloc(y*sizeof(tipo*));
for(int i = 0; i < y; ++i) {
  M[i] = (tipo*)malloc(x*sizeof(tipo));
}
```



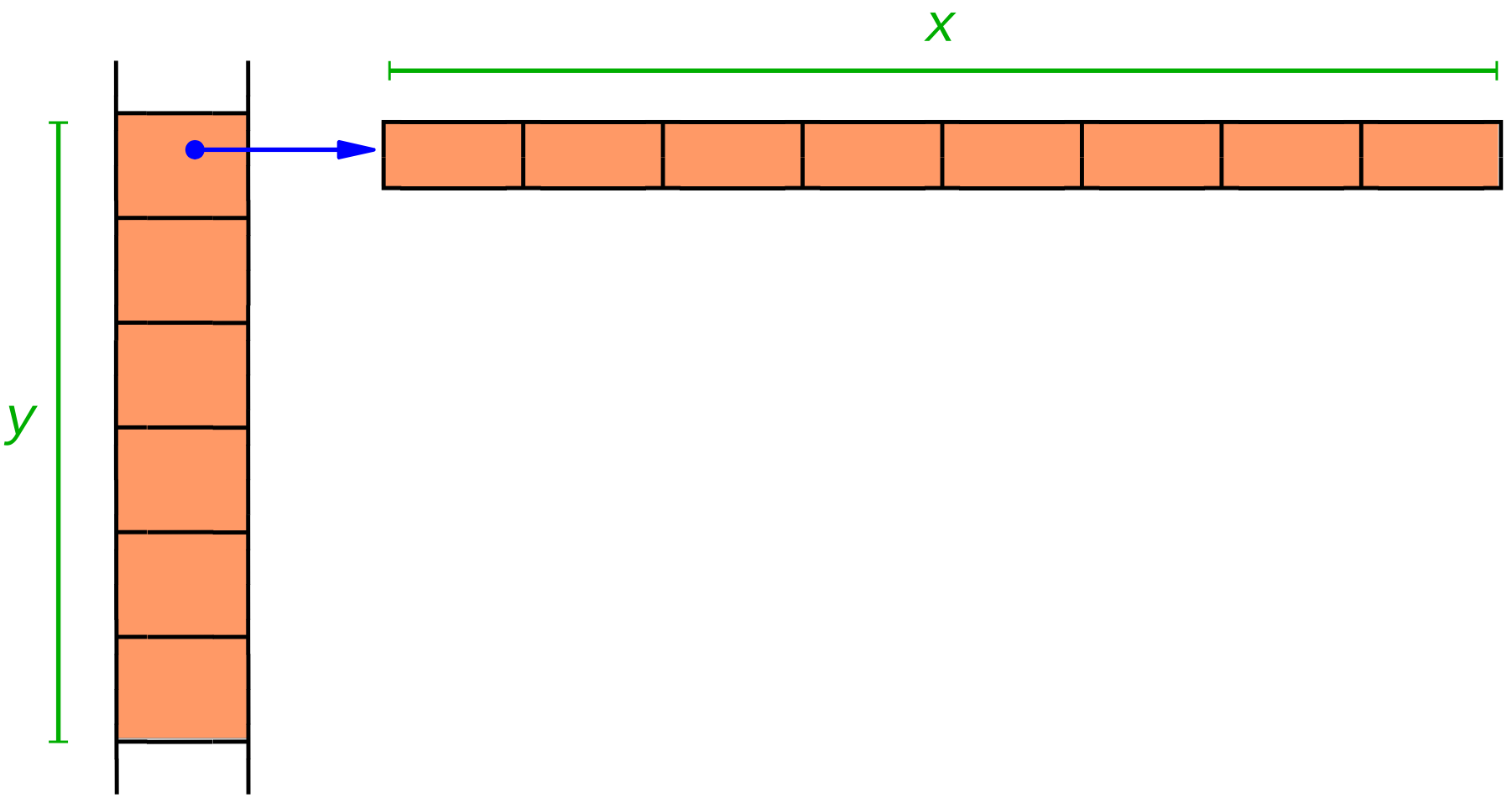
```
tipo **M = (tipo**)malloc(y*sizeof(tipo*));
for(int i = 0;▶i < y; ++i) {
  M[i] = (tipo*)malloc(x*sizeof(tipo));
}
```



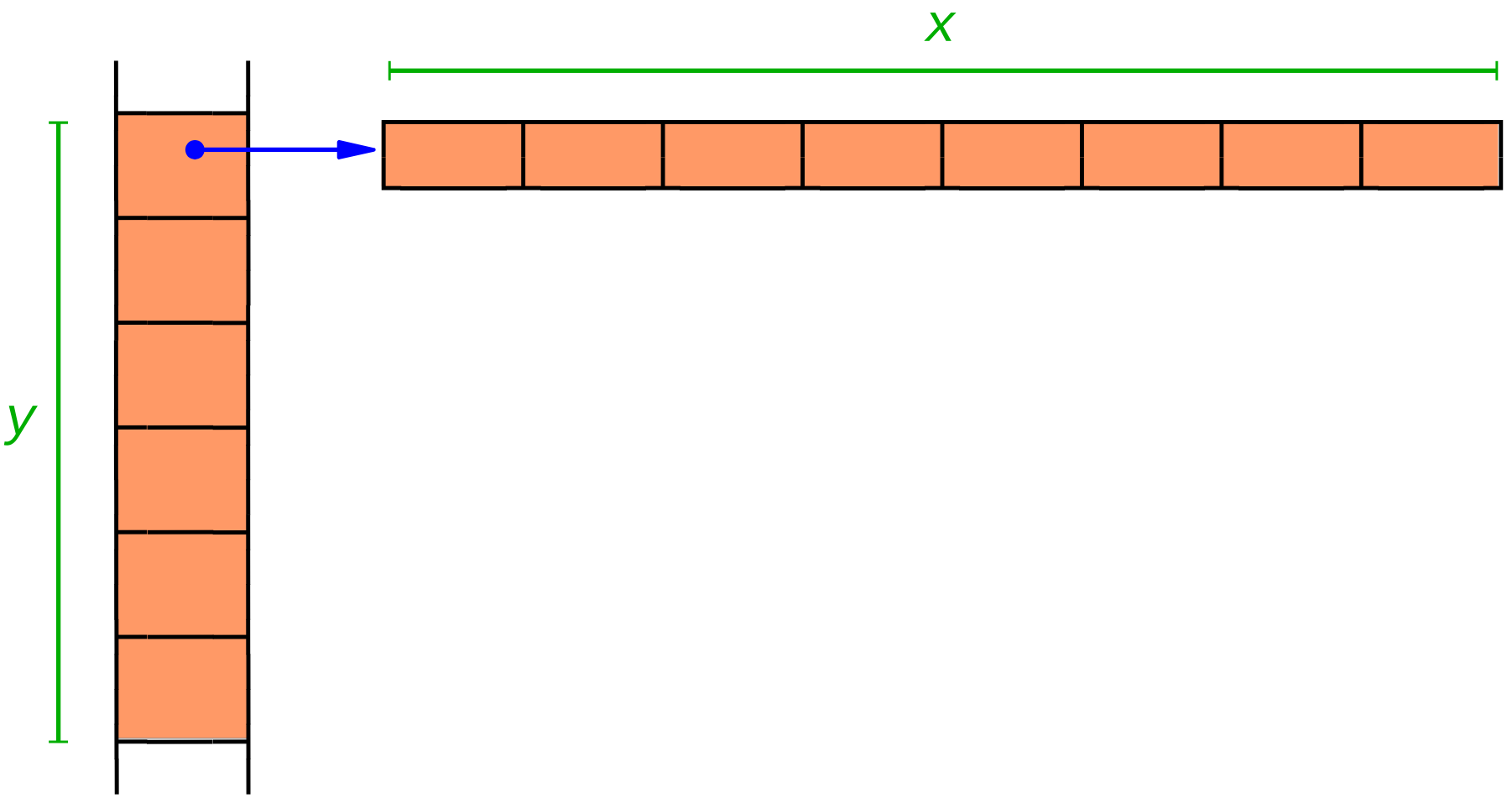
```
tipo **M = (tipo**)malloc(y*sizeof(tipo*));
for(int i = 0; i < y; ++i) {
  M[i] =▶(tipo*)malloc(x*sizeof(tipo));
}
```



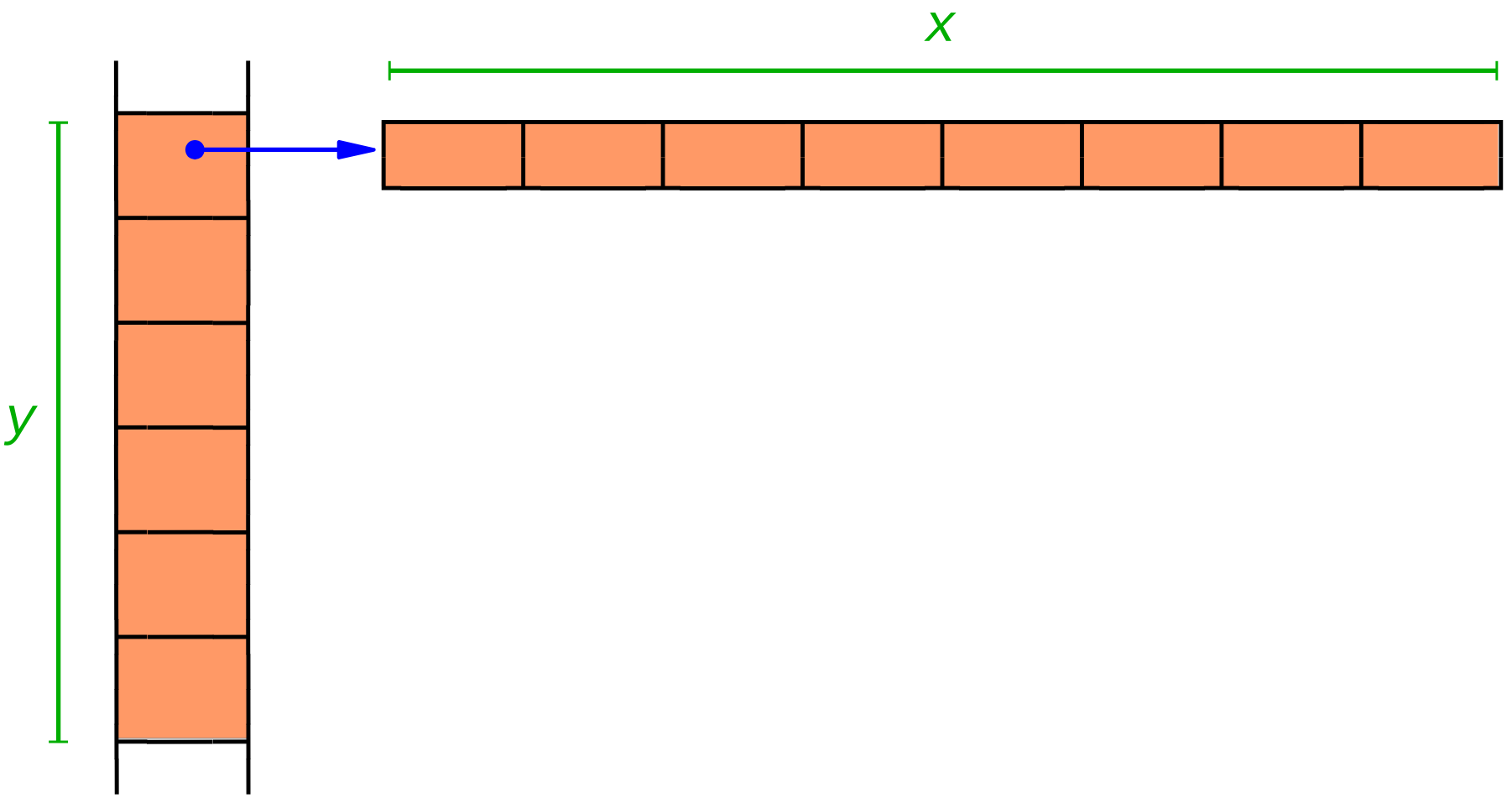
```
tipo **M = (tipo**)malloc(y*sizeof(tipo*));
for(int i = 0; i < y; ++i) {
 ▶M[i] = (tipo*)malloc(x*sizeof(tipo));
}
```



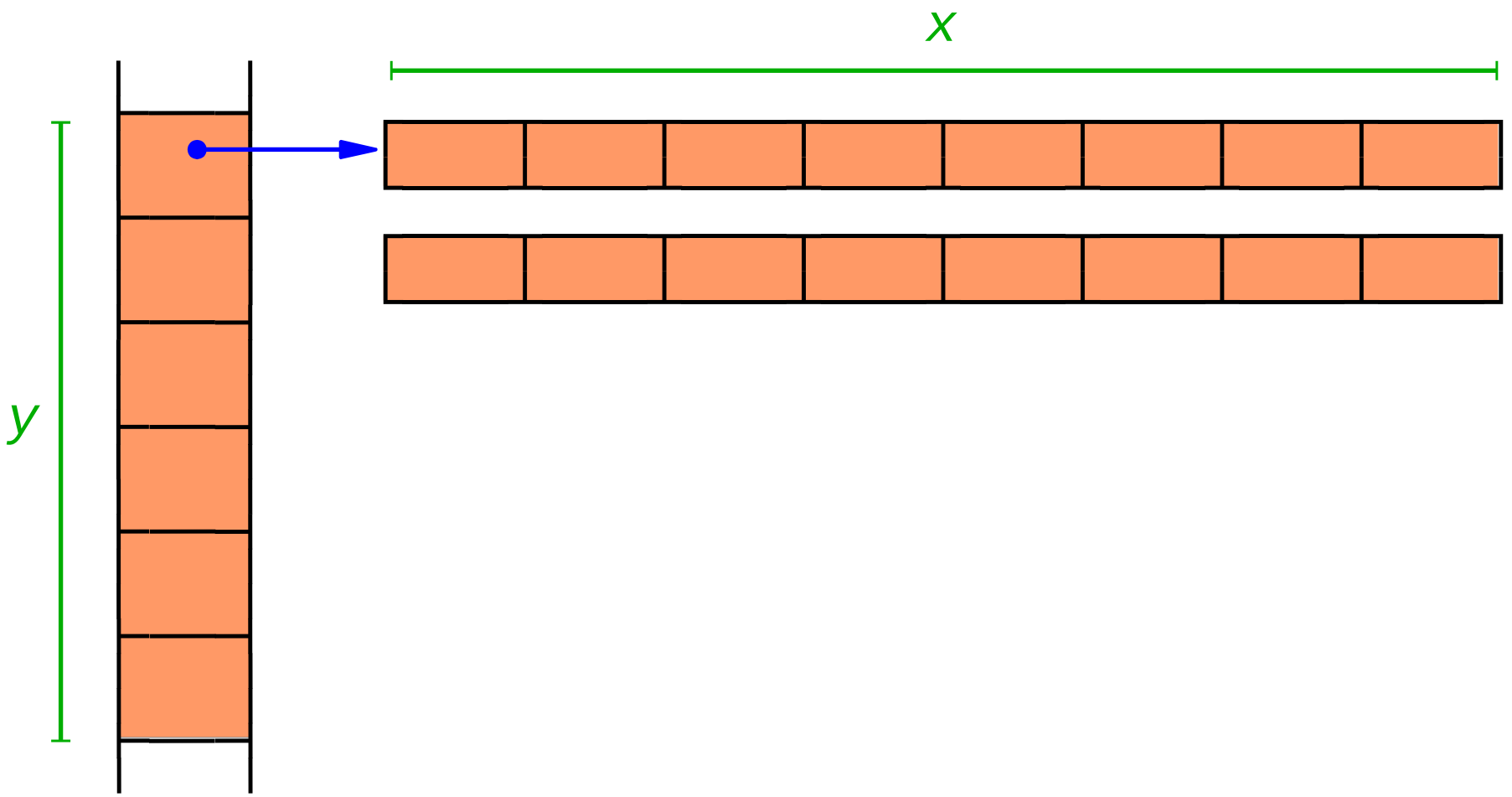
```
tipo **M = (tipo**)malloc(y*sizeof(tipo*));
for(int i = 0; i < y; ++i) {
  M[i] = (tipo*)malloc(x*sizeof(tipo));
}
```



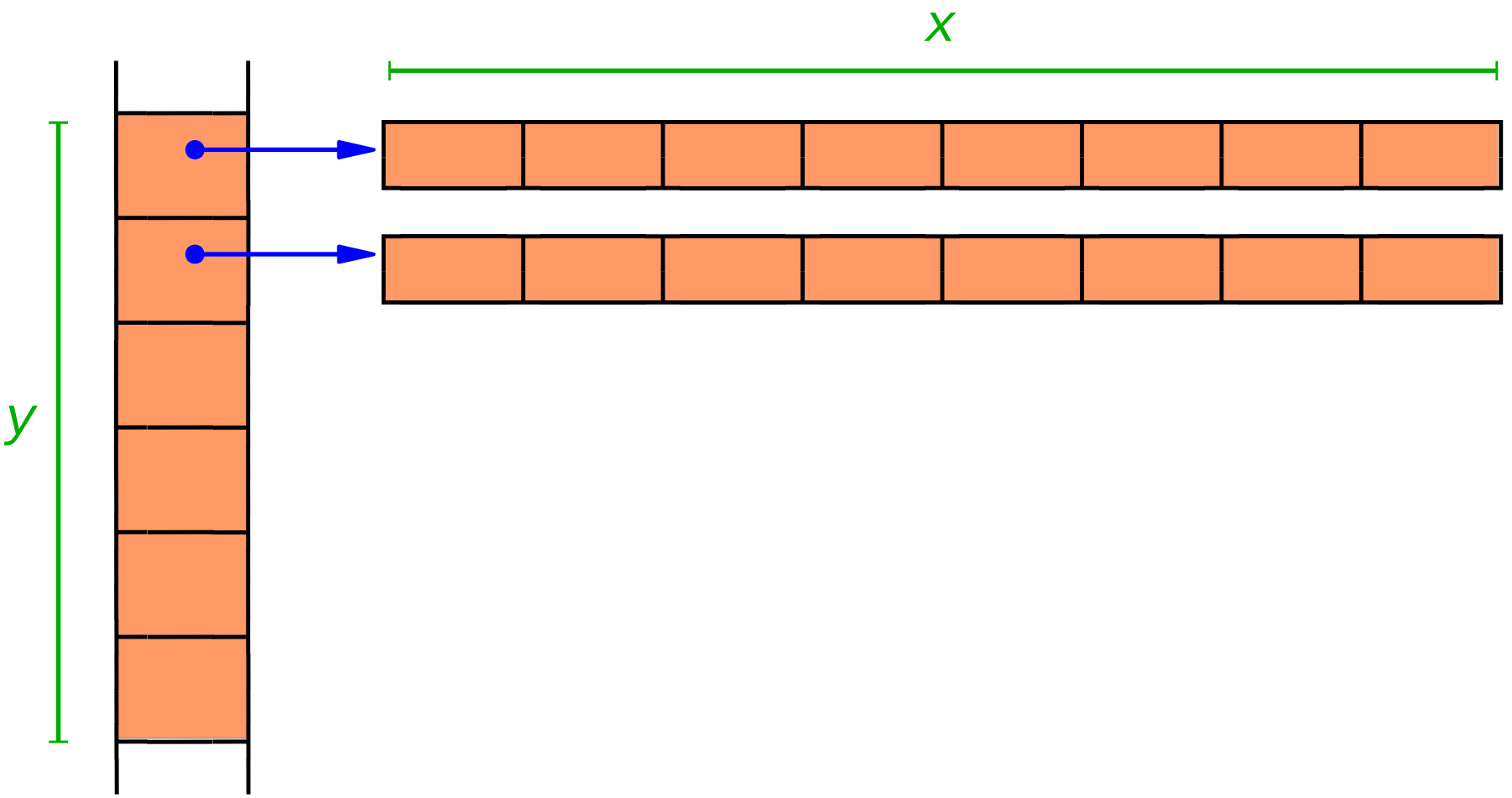
```
tipo **M = (tipo**)malloc(y*sizeof(tipo*));
for(int i = 0;▶i < y; ++i) {
  M[i] = (tipo*)malloc(x*sizeof(tipo));
}
```



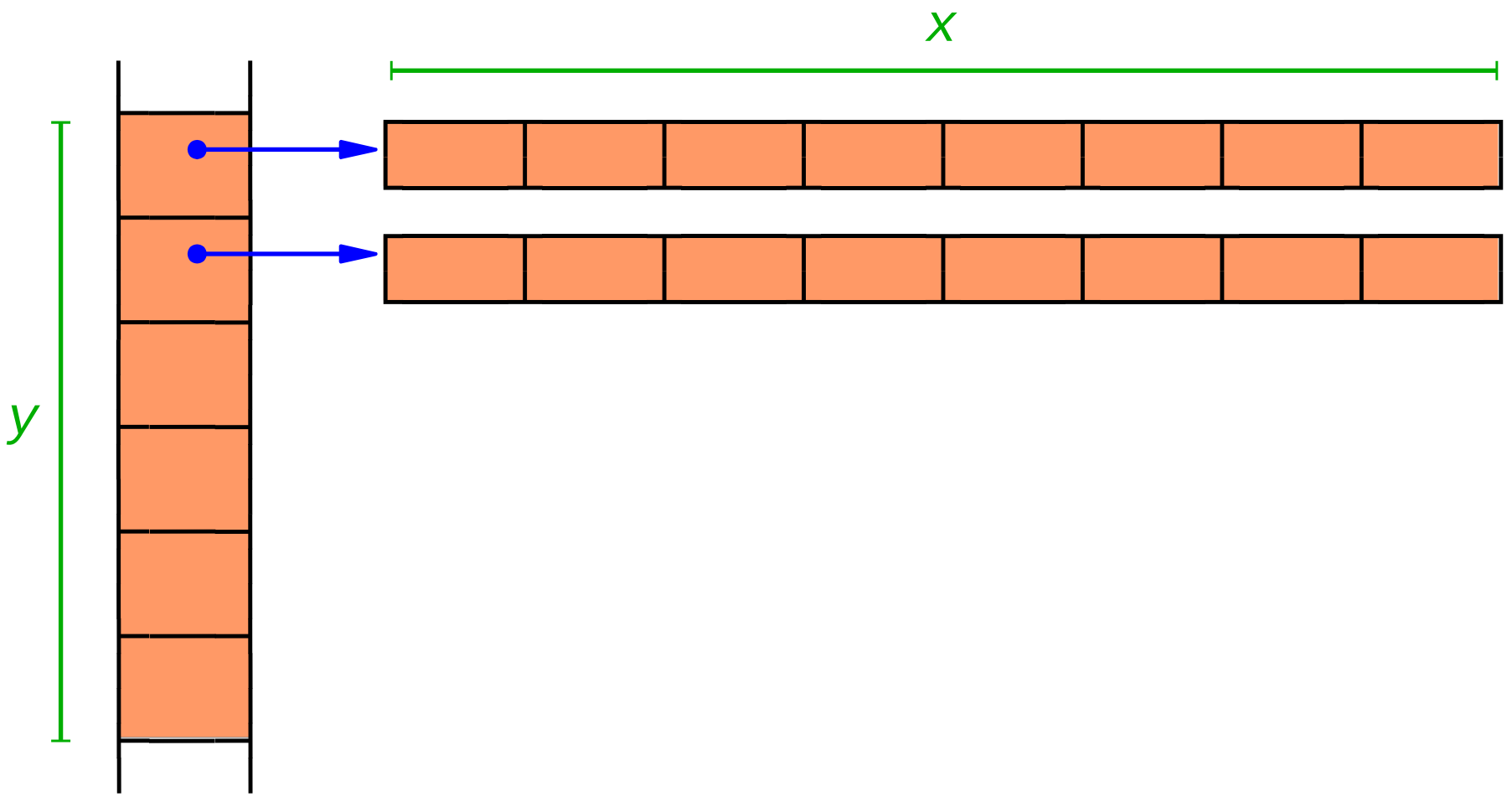
```
tipo **M = (tipo**)malloc(y*sizeof(tipo*));
for(int i = 0; i < y; ++i) {
  M[i] = (tipo*)malloc(x*sizeof(tipo));
}
```



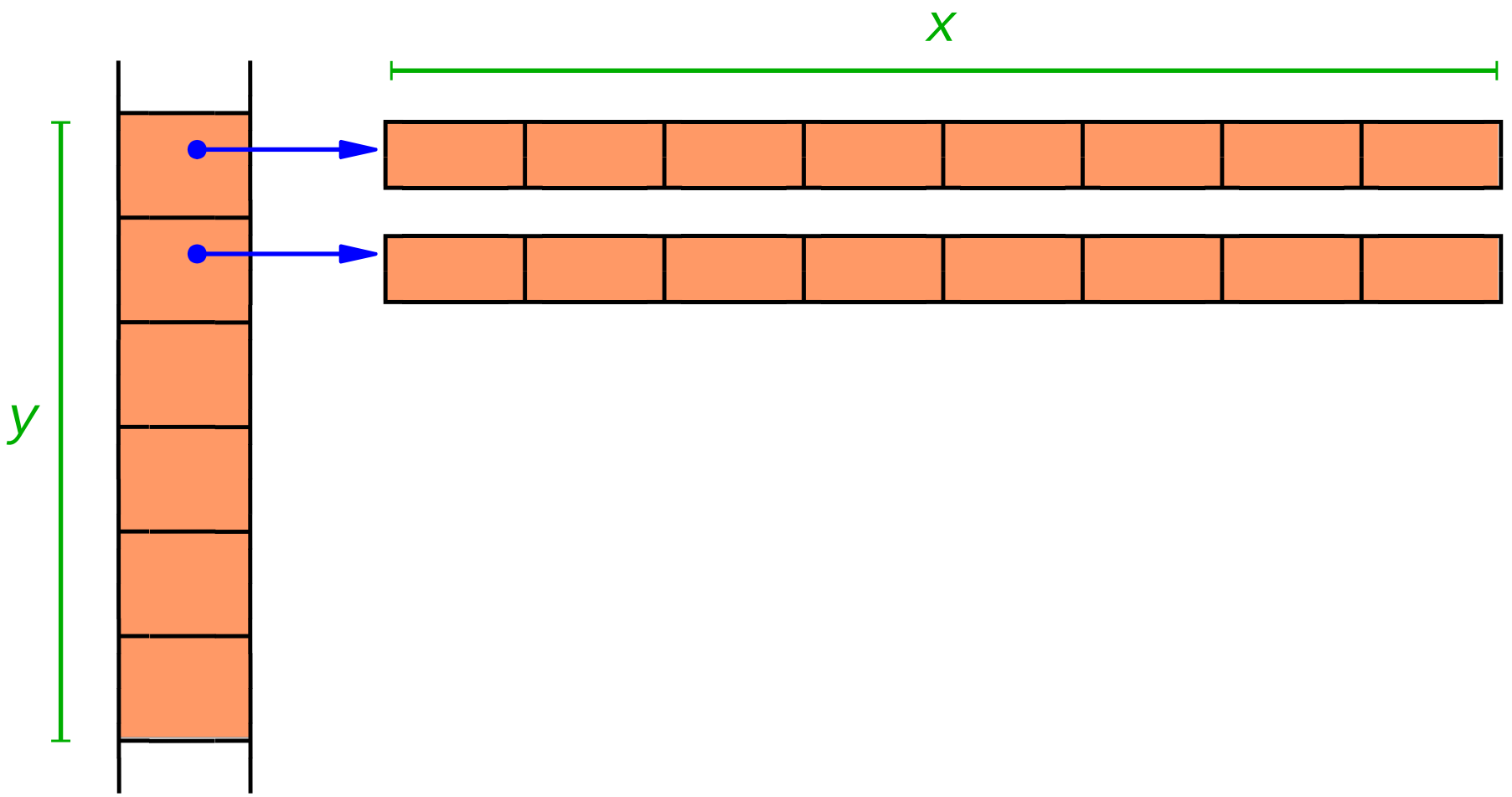
```
tipo **M = (tipo**)malloc(y*sizeof(tipo*));
for(int i = 0; i < y; ++i) {
 ▶M[i] = (tipo*)malloc(x*sizeof(tipo));
}
```



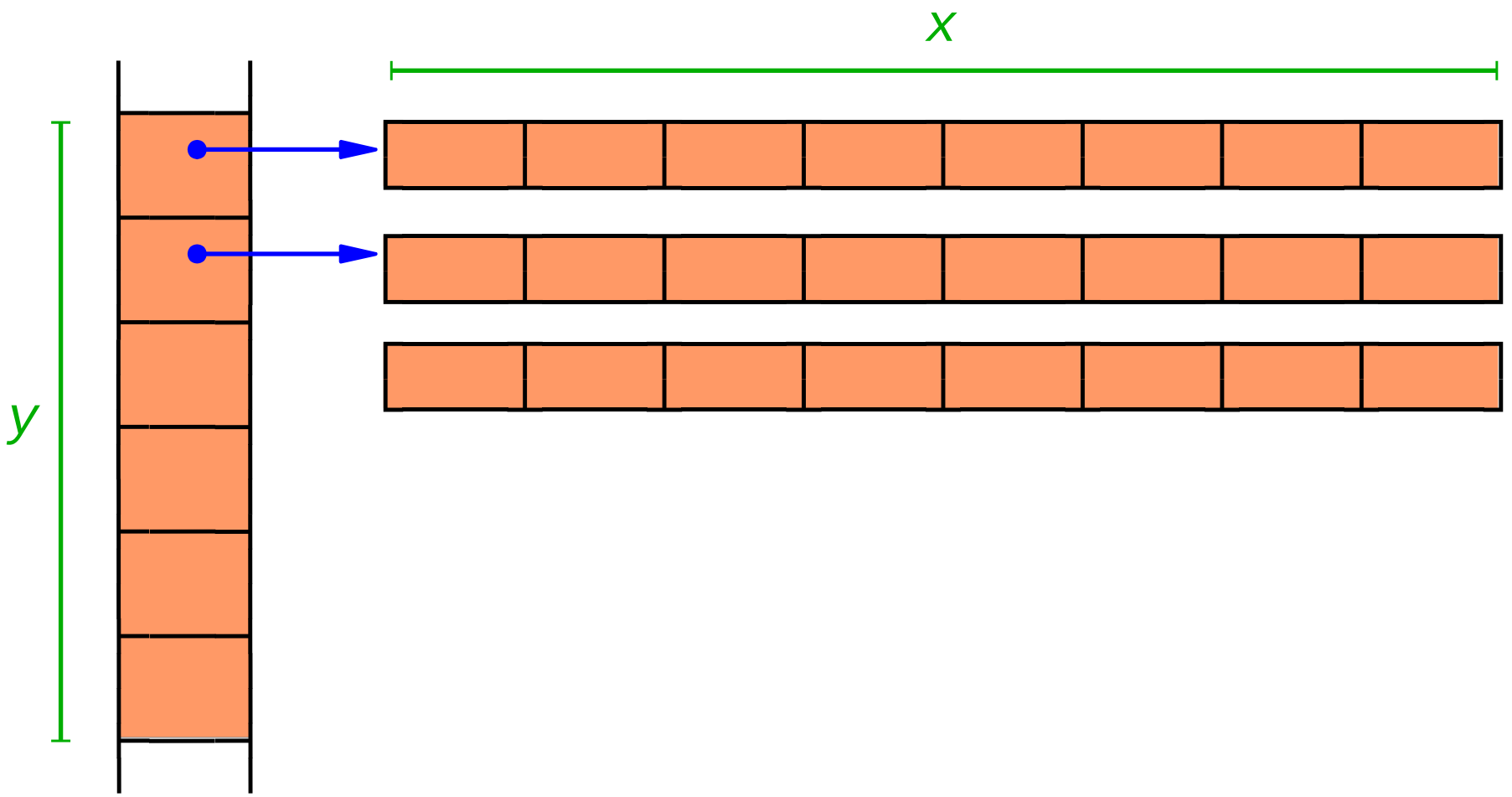
```
tipo **M = (tipo**)malloc(y*sizeof(tipo*));
for(int i = 0; i < y;▶++i) {
  M[i] = (tipo*)malloc(x*sizeof(tipo));
}
```



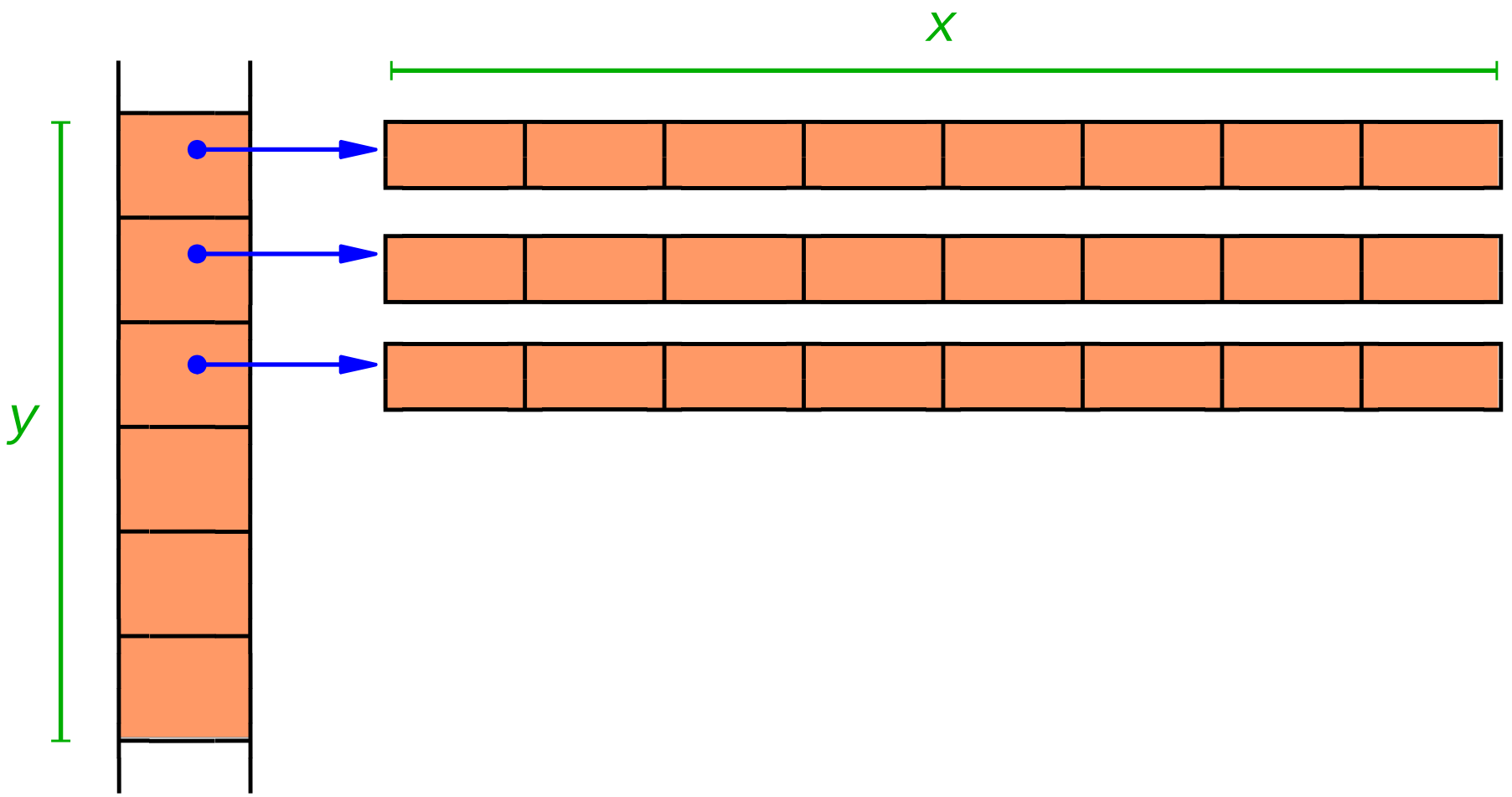
```
tipo **M = (tipo**)malloc(y*sizeof(tipo*));
for(int i = 0;▶i < y; ++i) {
  M[i] = (tipo*)malloc(x*sizeof(tipo));
}
```



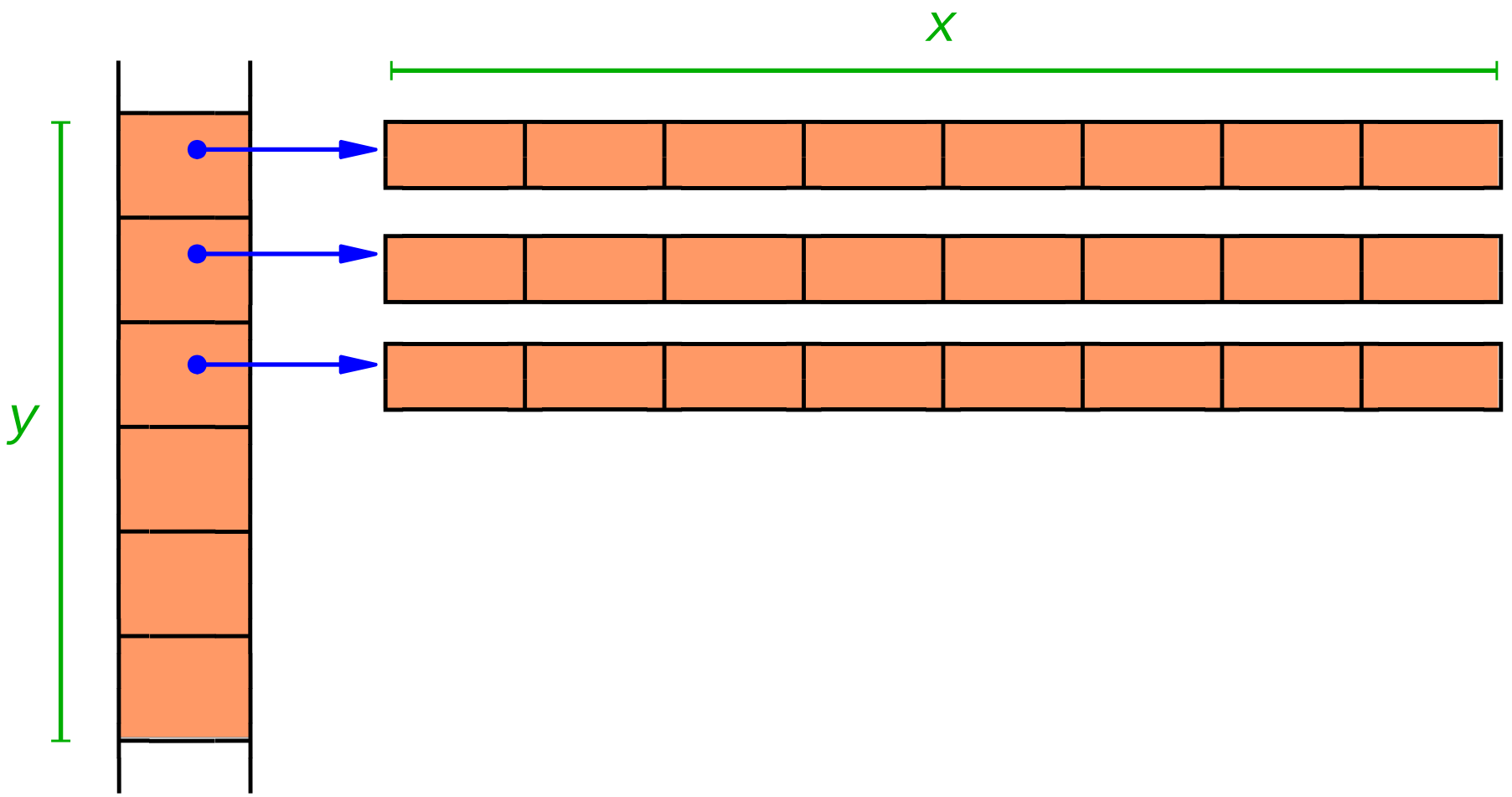
```
tipo **M = (tipo**)malloc(y*sizeof(tipo*));
for(int i = 0; i < y; ++i) {
  M[i] = (tipo*)malloc(x*sizeof(tipo));
}
```



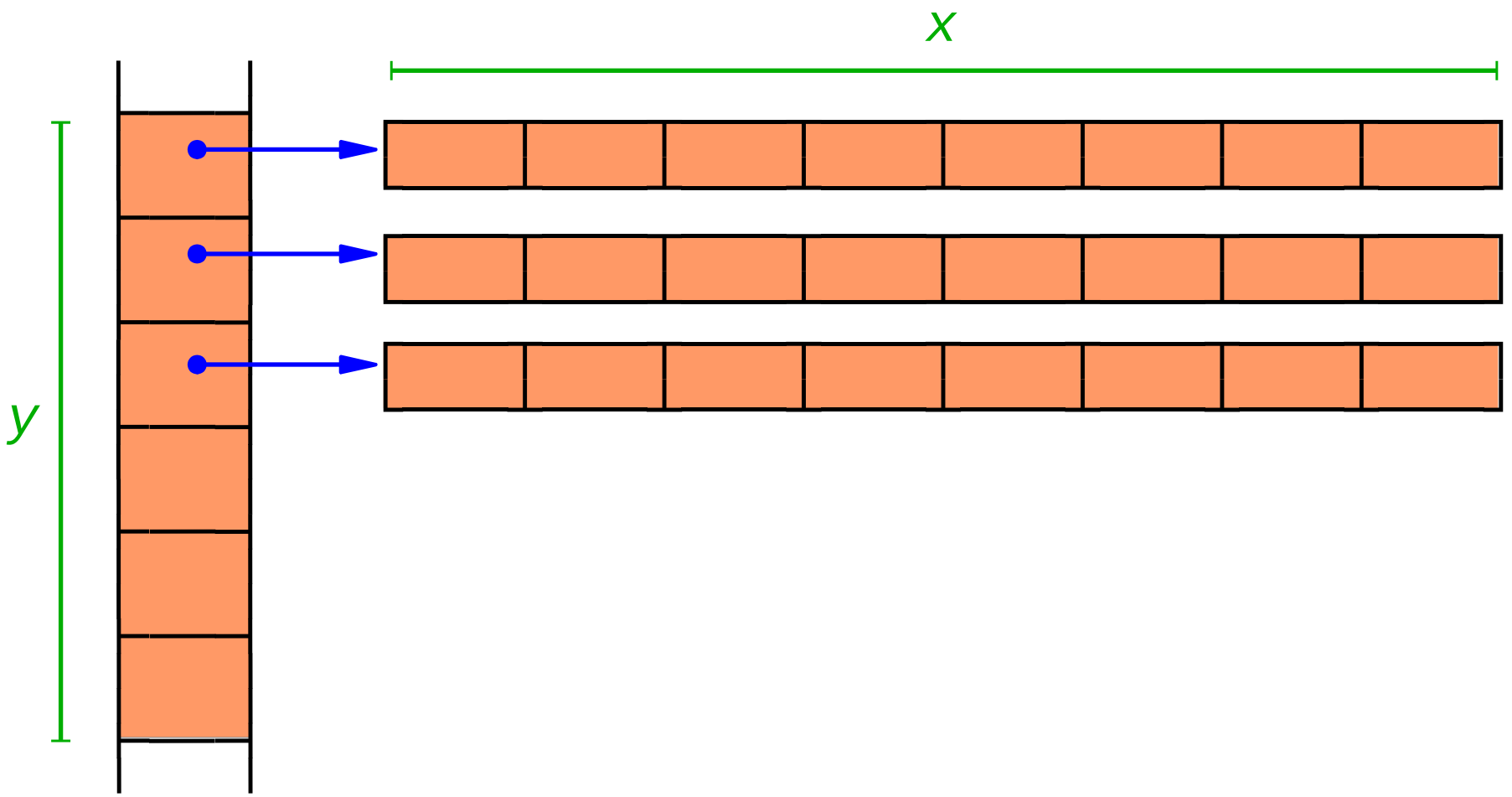
```
tipo **M = (tipo**)malloc(y*sizeof(tipo*));
for(int i = 0; i < y; ++i) {
 ▶M[i] = (tipo*)malloc(x*sizeof(tipo));
}
```



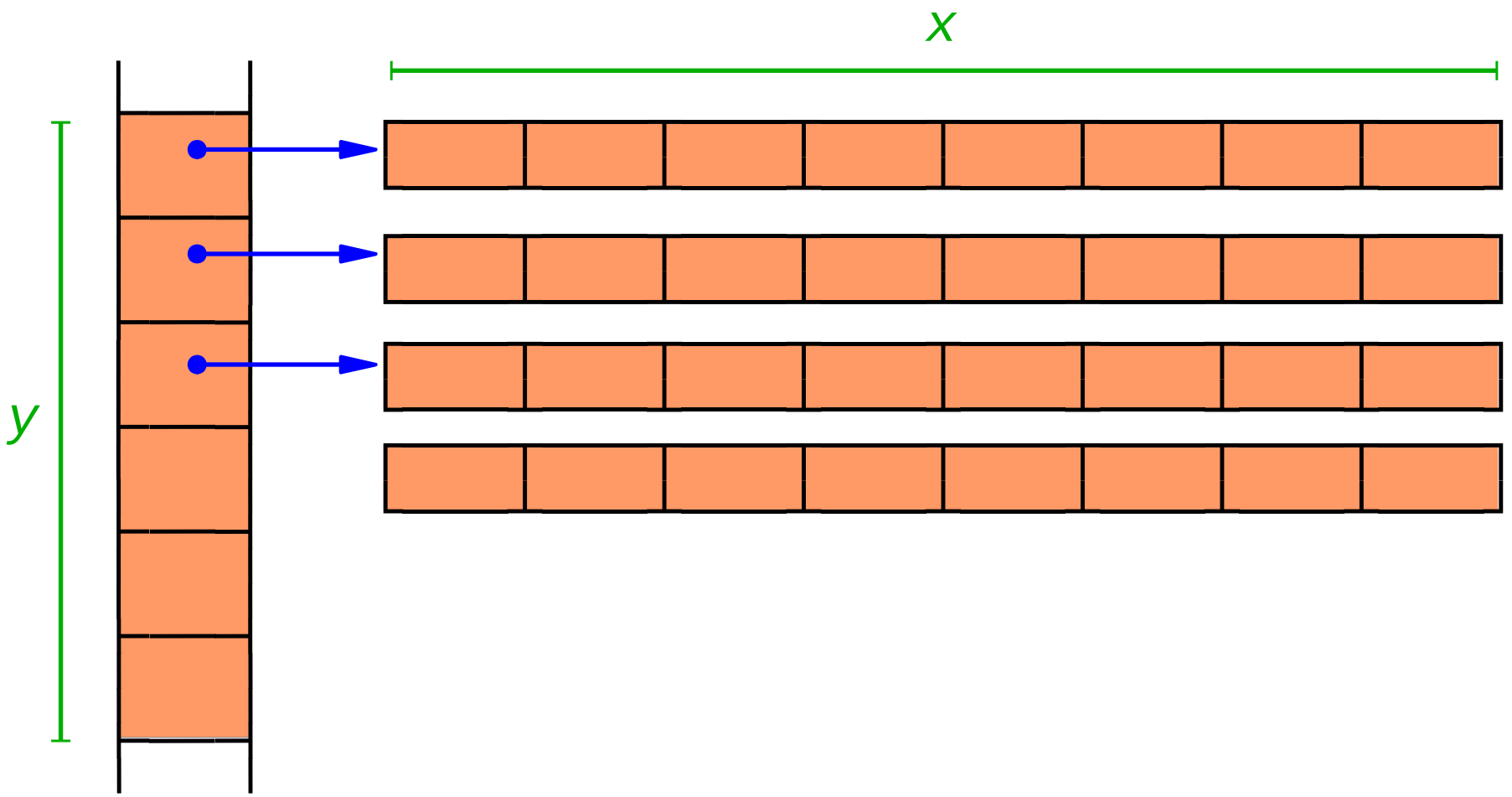
```
tipo **M = (tipo**)malloc(y*sizeof(tipo*));
for(int i = 0; i < y; ++i) {
  M[i] = (tipo*)malloc(x*sizeof(tipo));
}
```



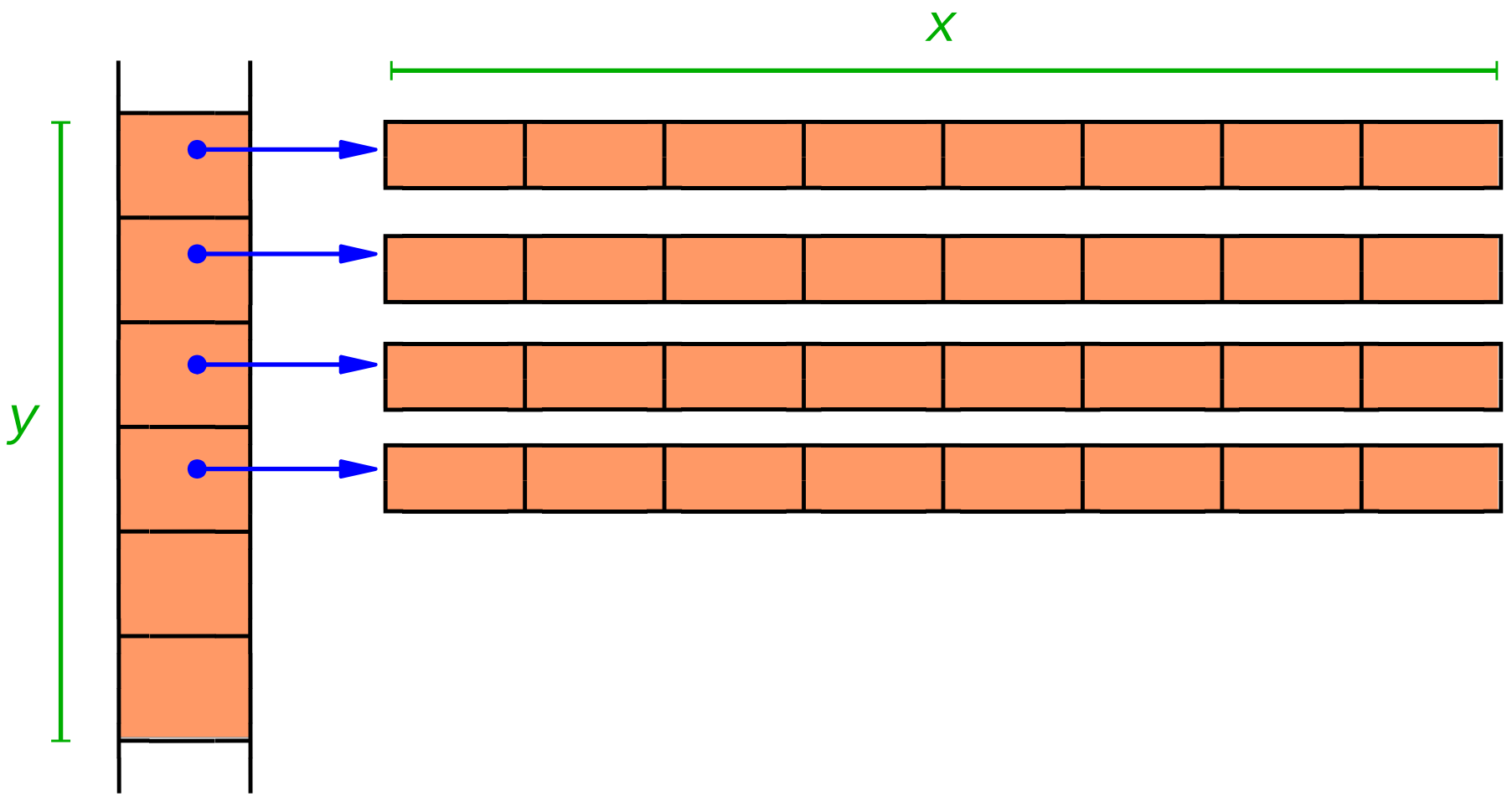
```
tipo **M = (tipo**)malloc(y*sizeof(tipo*));
for(int i = 0;▶i < y; ++i) {
  M[i] = (tipo*)malloc(x*sizeof(tipo));
}
```



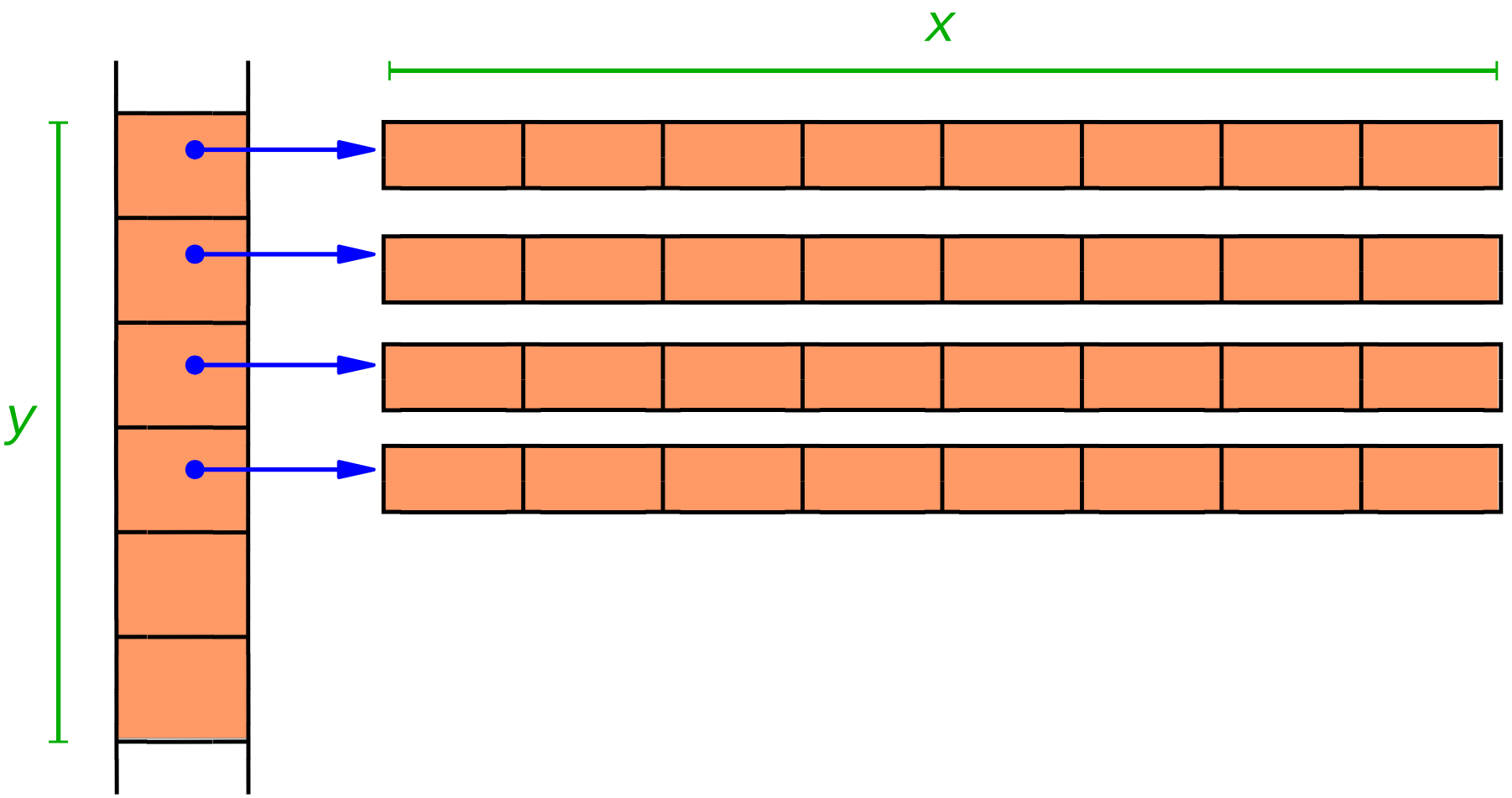
```
tipo **M = (tipo**)malloc(y*sizeof(tipo*));
for(int i = 0; i < y; ++i) {
  M[i] = (tipo*)malloc(x*sizeof(tipo));
}
```



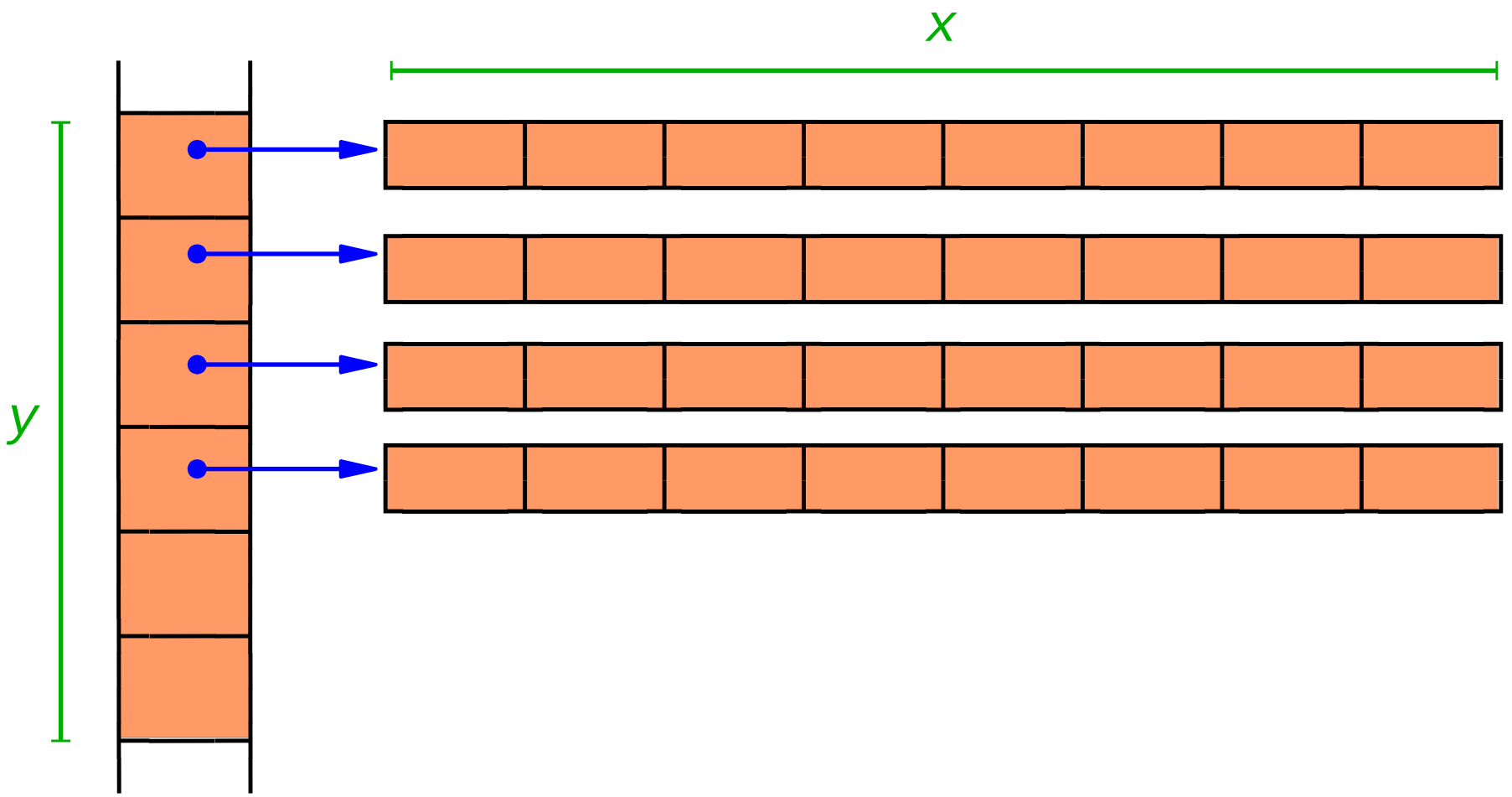
```
tipo **M = (tipo**)malloc(y*sizeof(tipo*));
for(int i = 0; i < y; ++i) {
 ▶M[i] = (tipo*)malloc(x*sizeof(tipo));
}
```



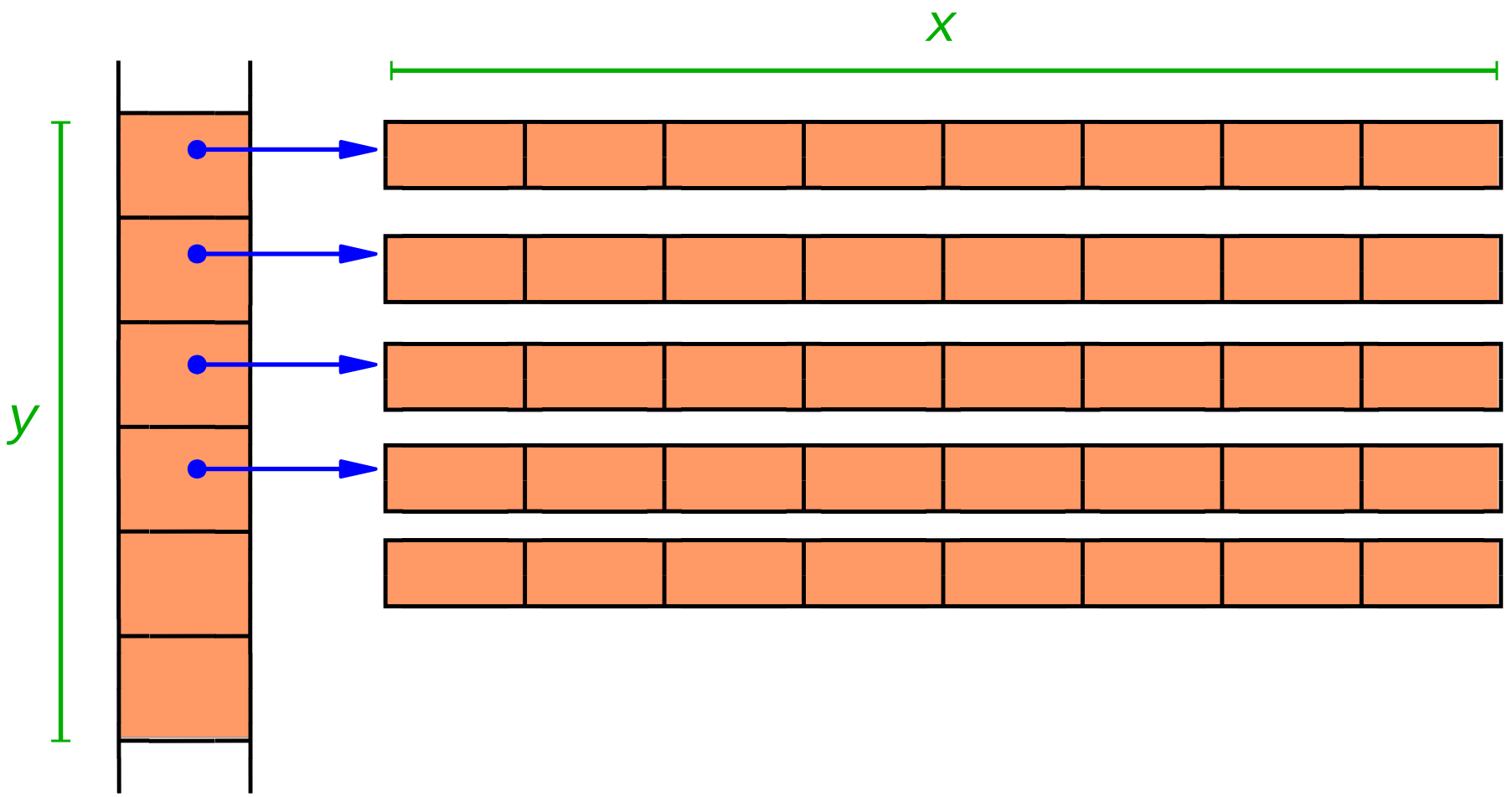
```
tipo **M = (tipo**)malloc(y*sizeof(tipo*));
for(int i = 0; i < y; ++i) {
  M[i] = (tipo*)malloc(x*sizeof(tipo));
}
```



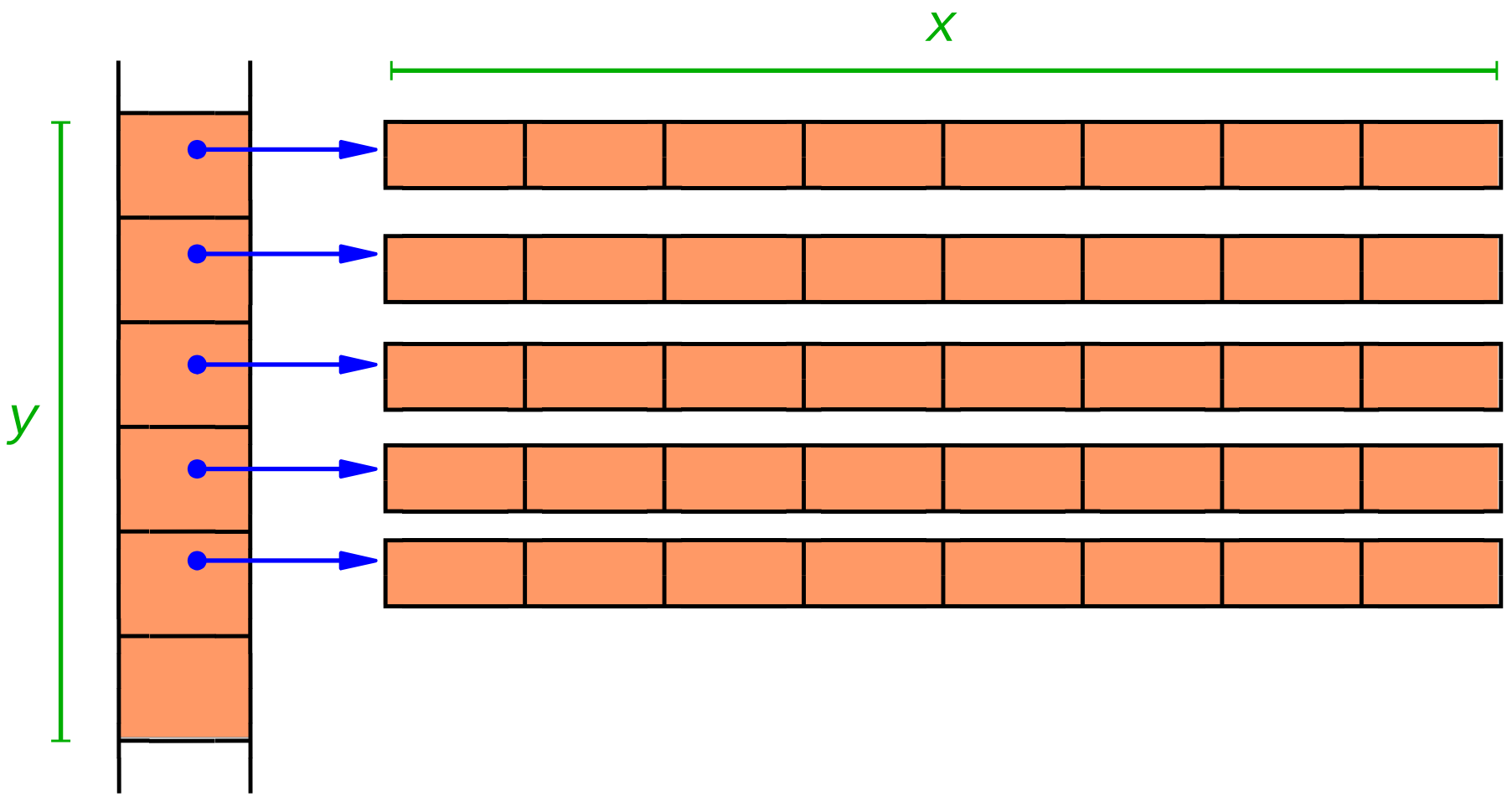
```
tipo **M = (tipo**)malloc(y*sizeof(tipo*));
for(int i = 0;▶i < y; ++i) {
  M[i] = (tipo*)malloc(x*sizeof(tipo));
}
```



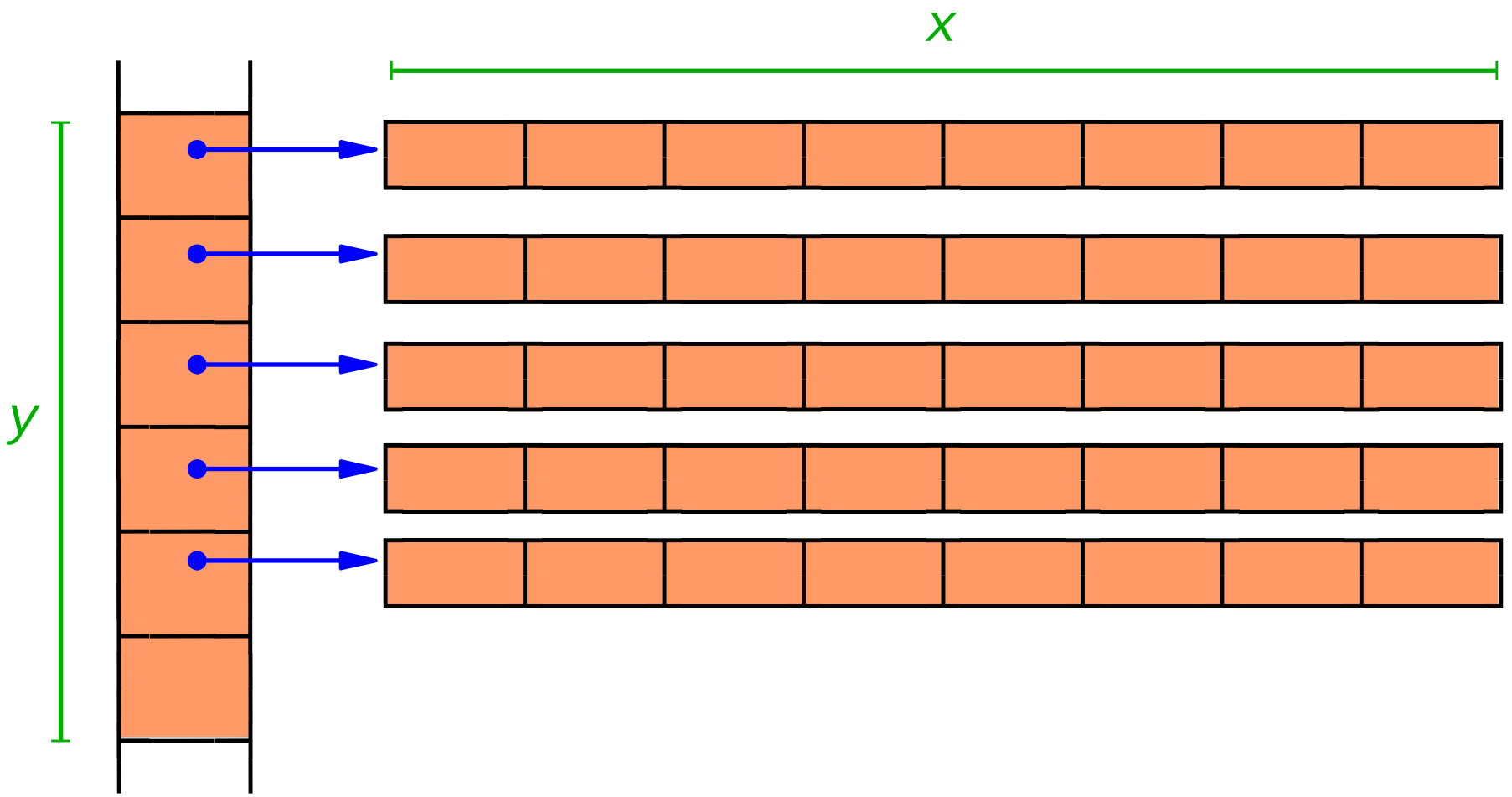
```
tipo **M = (tipo**)malloc(y*sizeof(tipo*));
for(int i = 0; i < y; ++i) {
  M[i] =(tipo*)malloc(x*sizeof(tipo));
}
```



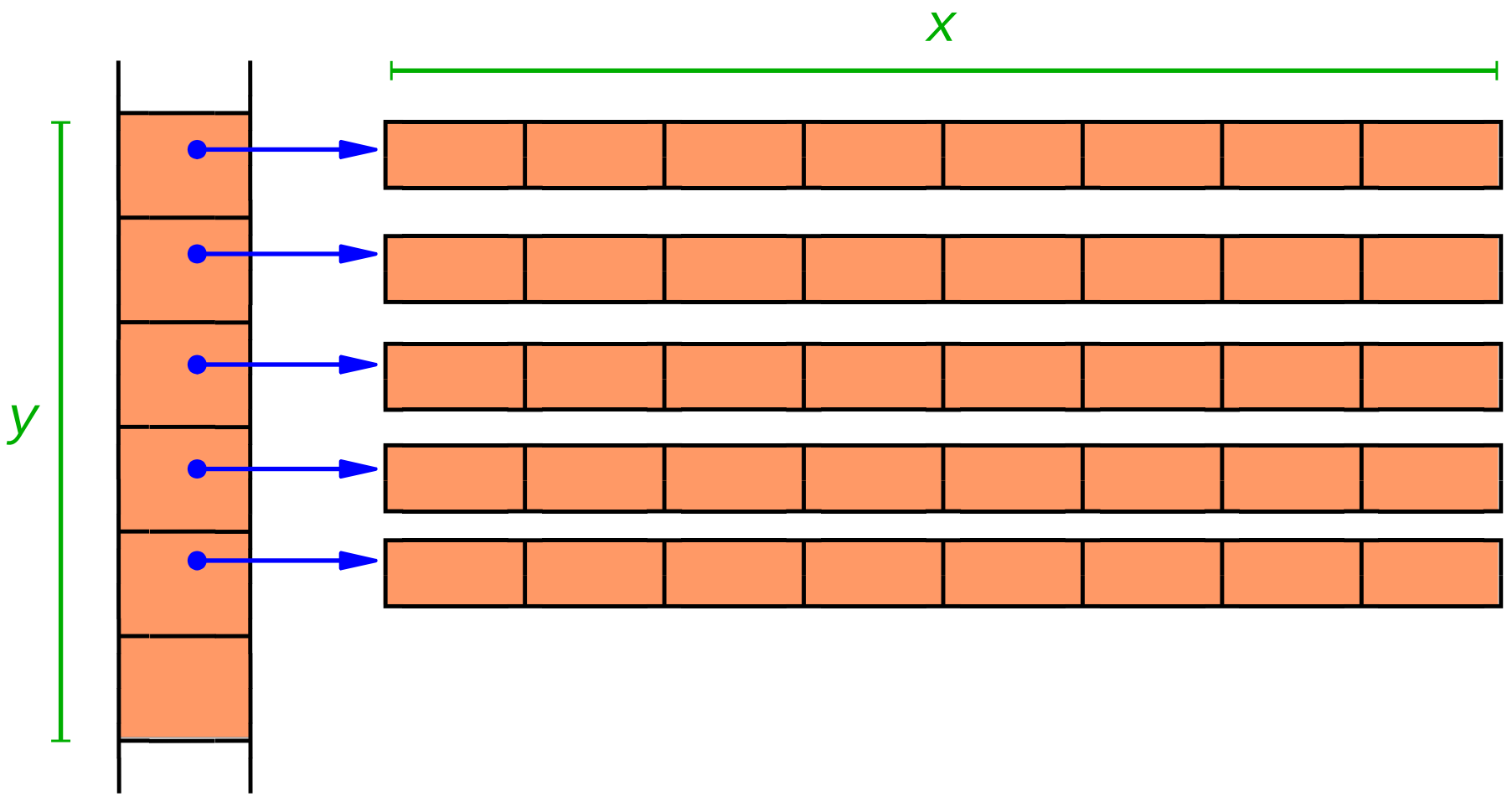
```
tipo **M = (tipo**)malloc(y*sizeof(tipo*));
for(int i = 0; i < y; ++i) {
 ▶M[i] = (tipo*)malloc(x*sizeof(tipo));
}
```



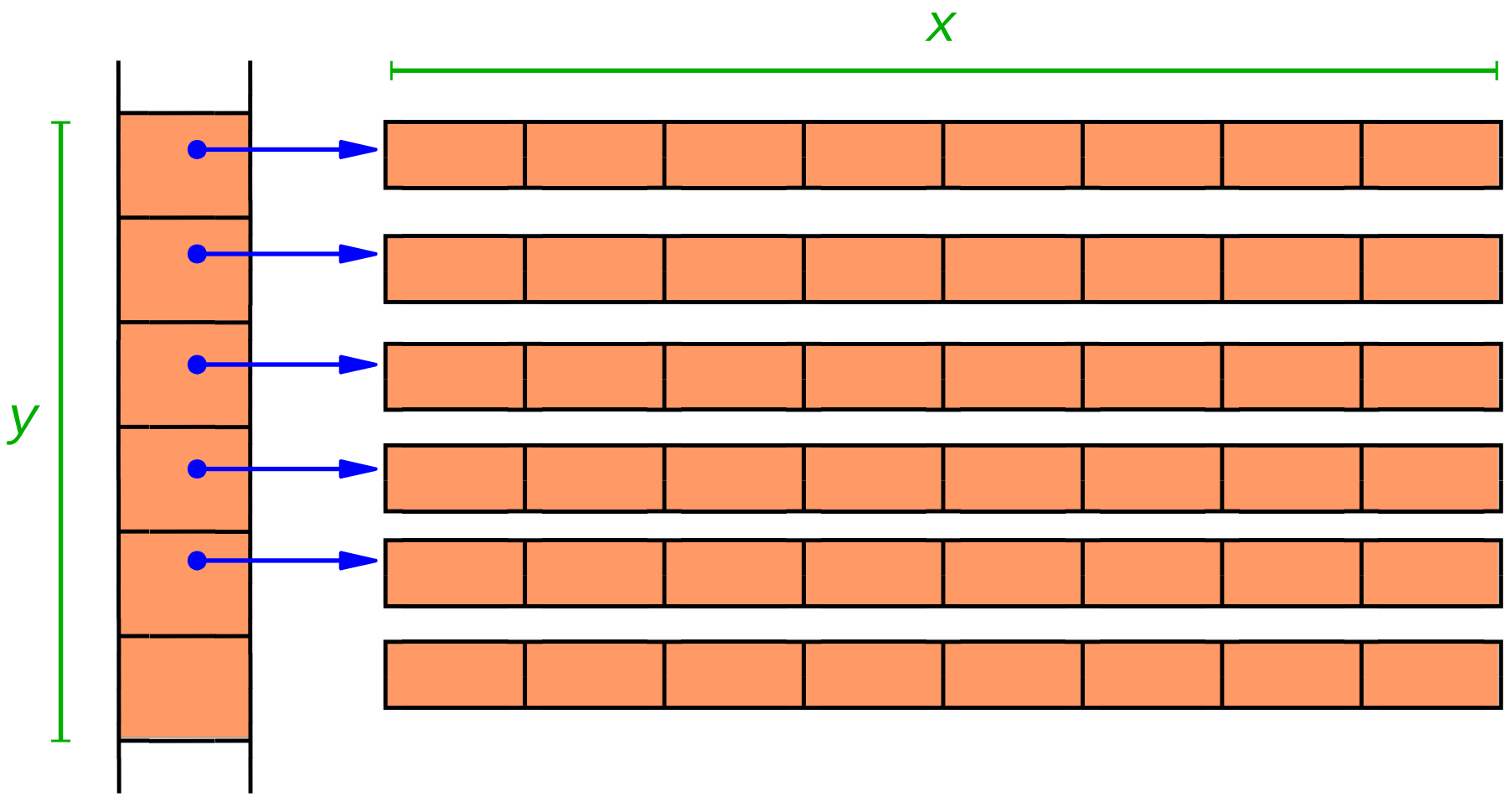
```
tipo **M = (tipo**)malloc(y*sizeof(tipo*));
for(int i = 0; i < y; ++i) {
  M[i] = (tipo*)malloc(x*sizeof(tipo));
}
```



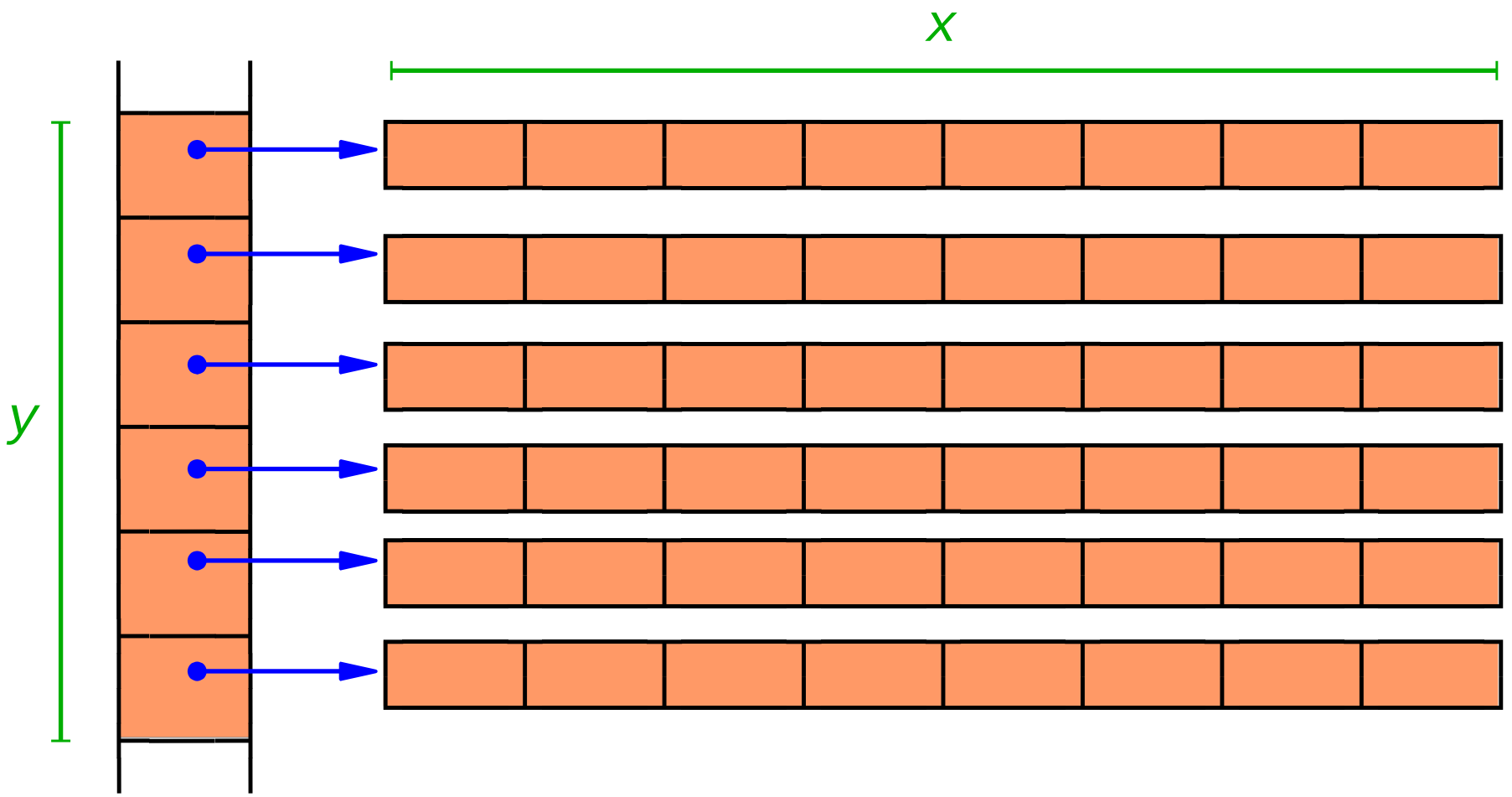
```
tipo **M = (tipo**)malloc(y*sizeof(tipo*));
for(int i = 0;▶i < y; ++i) {
  M[i] = (tipo*)malloc(x*sizeof(tipo));
}
```



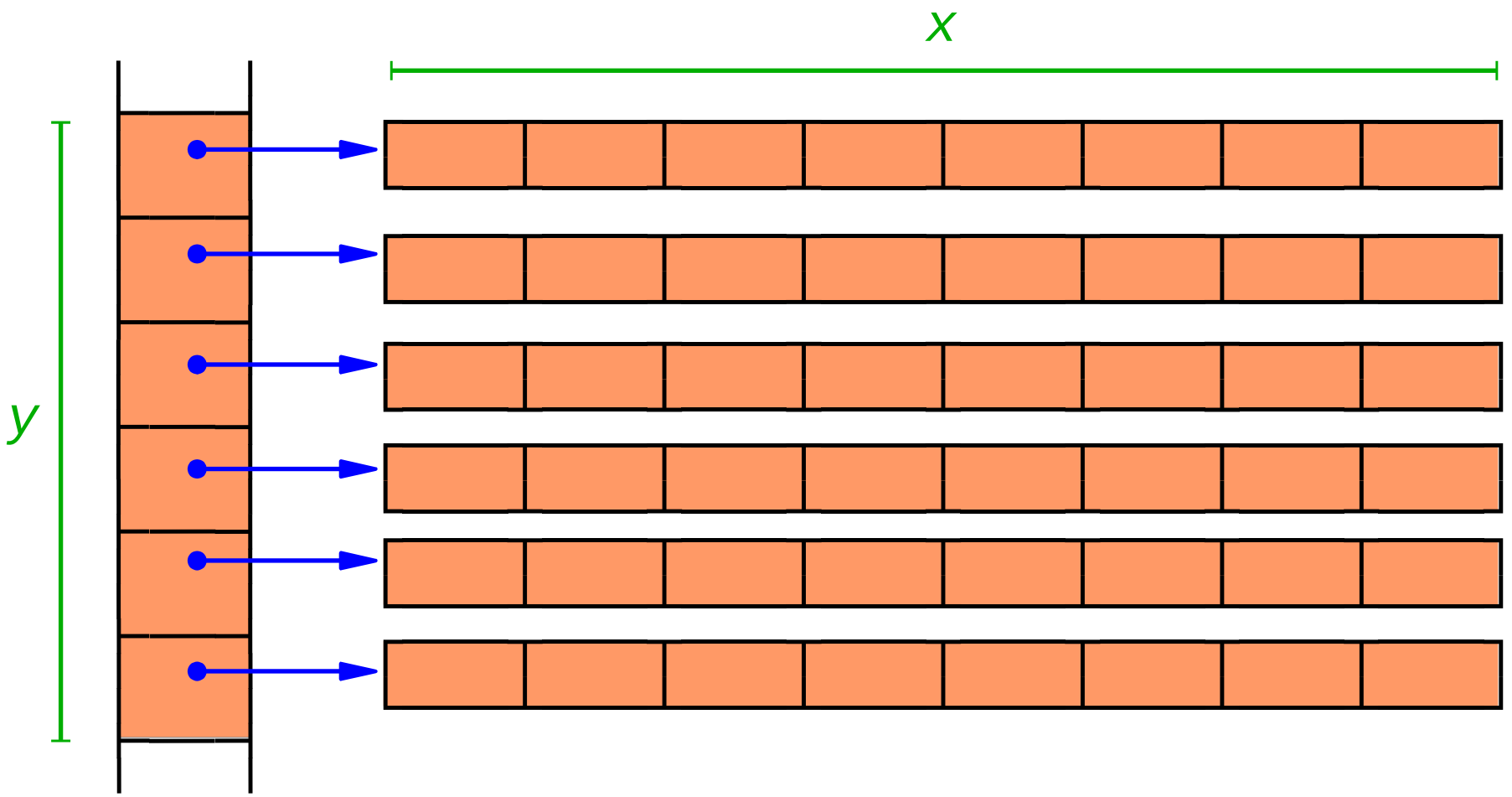
```
tipo **M = (tipo**)malloc(y*sizeof(tipo*));
for(int i = 0; i < y; ++i) {
  M[i] =▶(tipo*)malloc(x*sizeof(tipo));
}
```



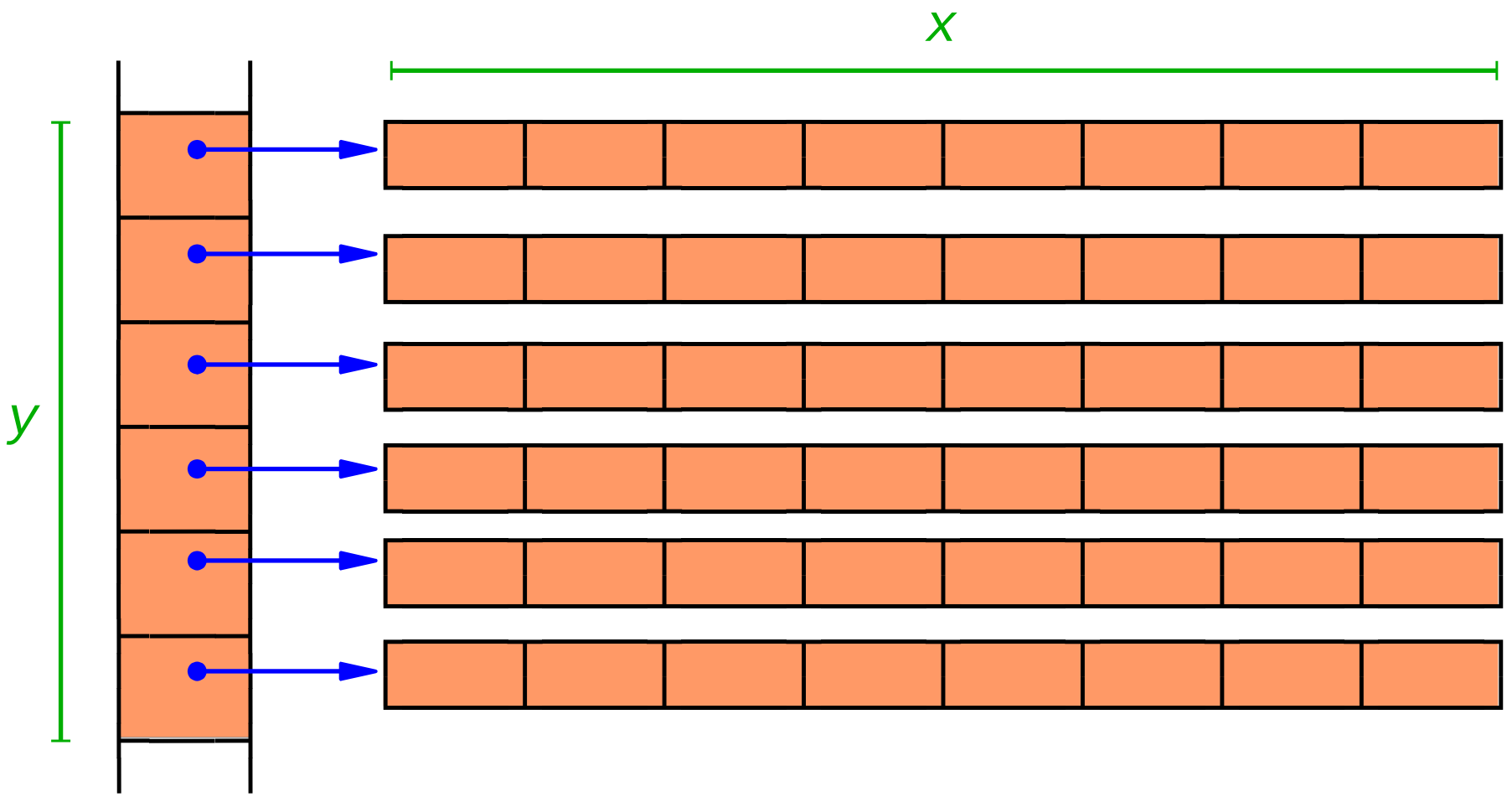
```
tipo **M = (tipo**)malloc(y*sizeof(tipo*));
for(int i = 0; i < y; ++i) {
 ▶M[i] = (tipo*)malloc(x*sizeof(tipo));
}
```



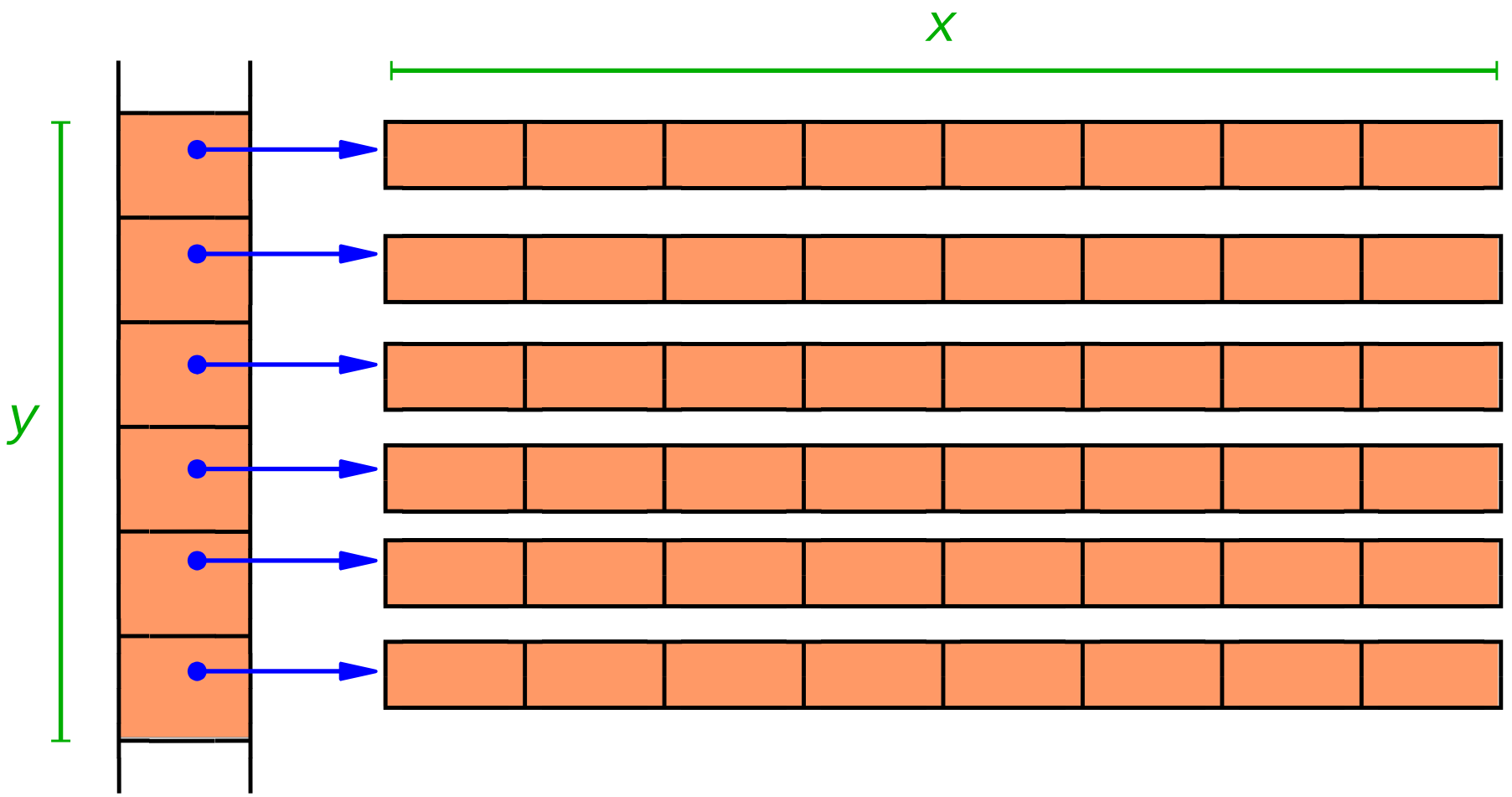
```
tipo **M = (tipo**)malloc(y*sizeof(tipo*));
for(int i = 0; i < y; ++i) {
  M[i] = (tipo*)malloc(x*sizeof(tipo));
}
```



```
tipo **M = (tipo**)malloc(y*sizeof(tipo*));
for(int i = 0;▶i < y; ++i) {
  M[i] = (tipo*)malloc(x*sizeof(tipo));
}
```

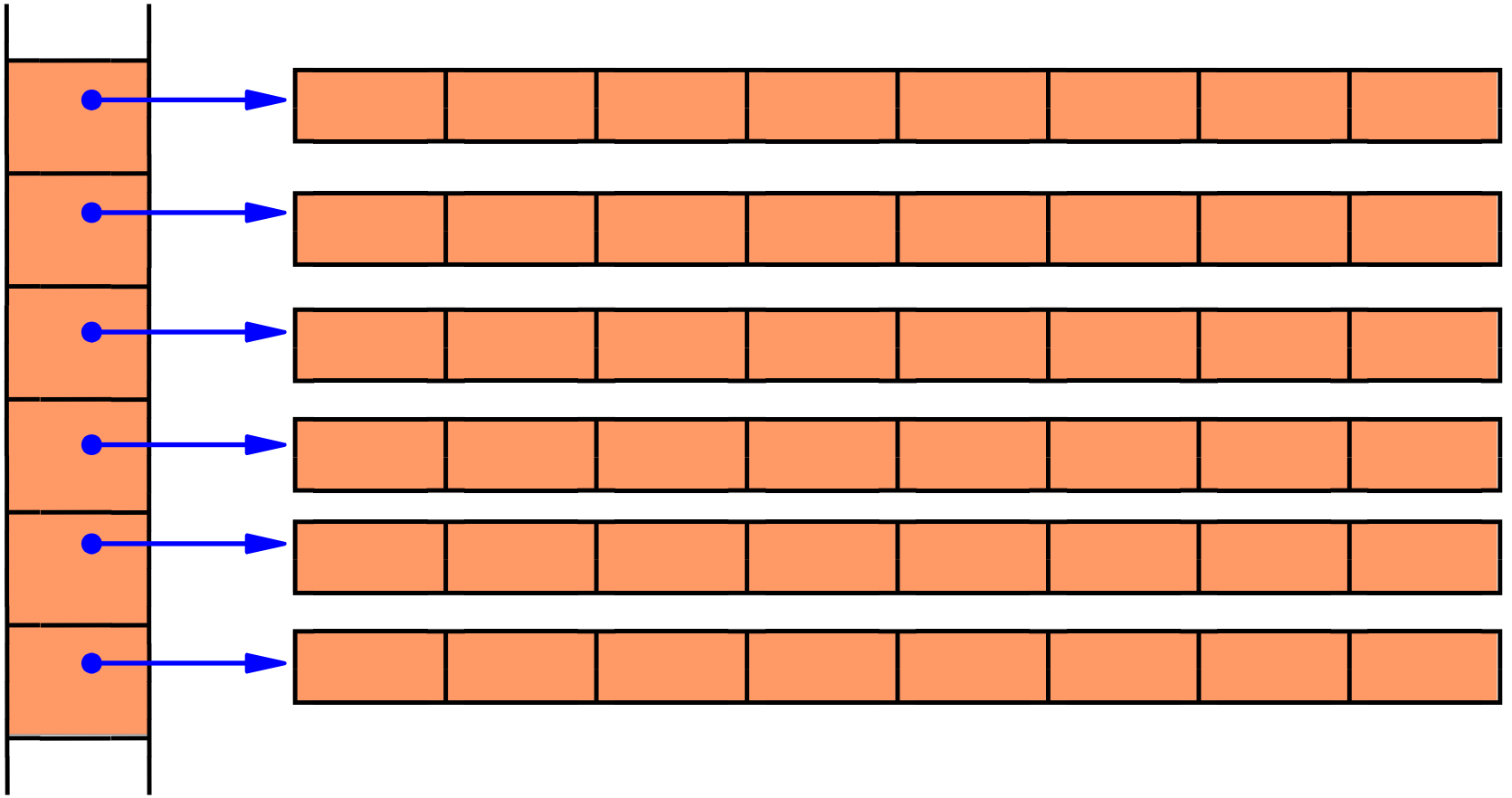


```
tipo **M = (tipo**)malloc(y*sizeof(tipo*));
for(int i = 0; i < y; ++i) {
  M[i] = (tipo*)malloc(x*sizeof(tipo));
}
```



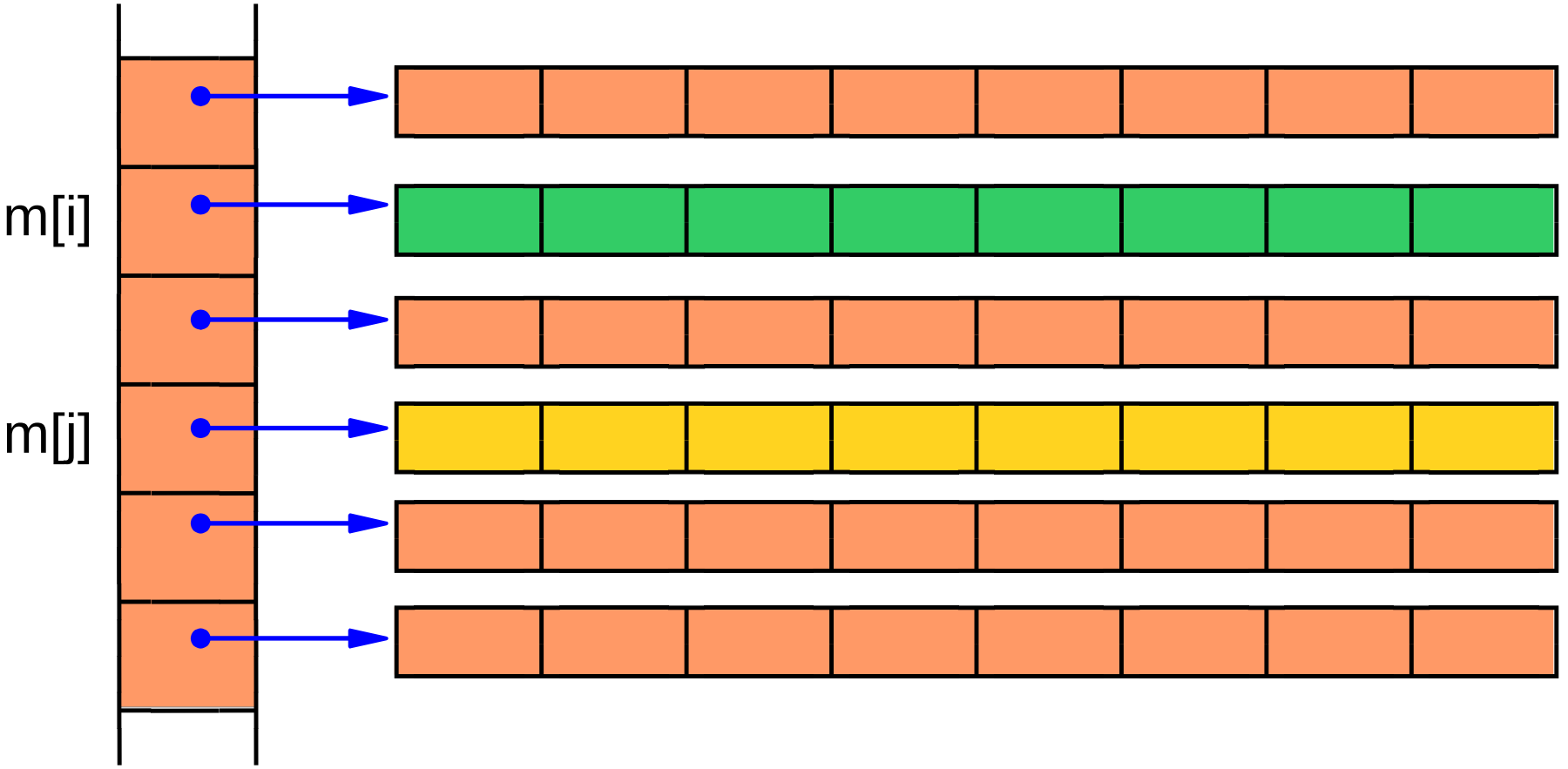
VANTAGENS

É fácil trocar quaisquer duas linhas da matriz entre si.



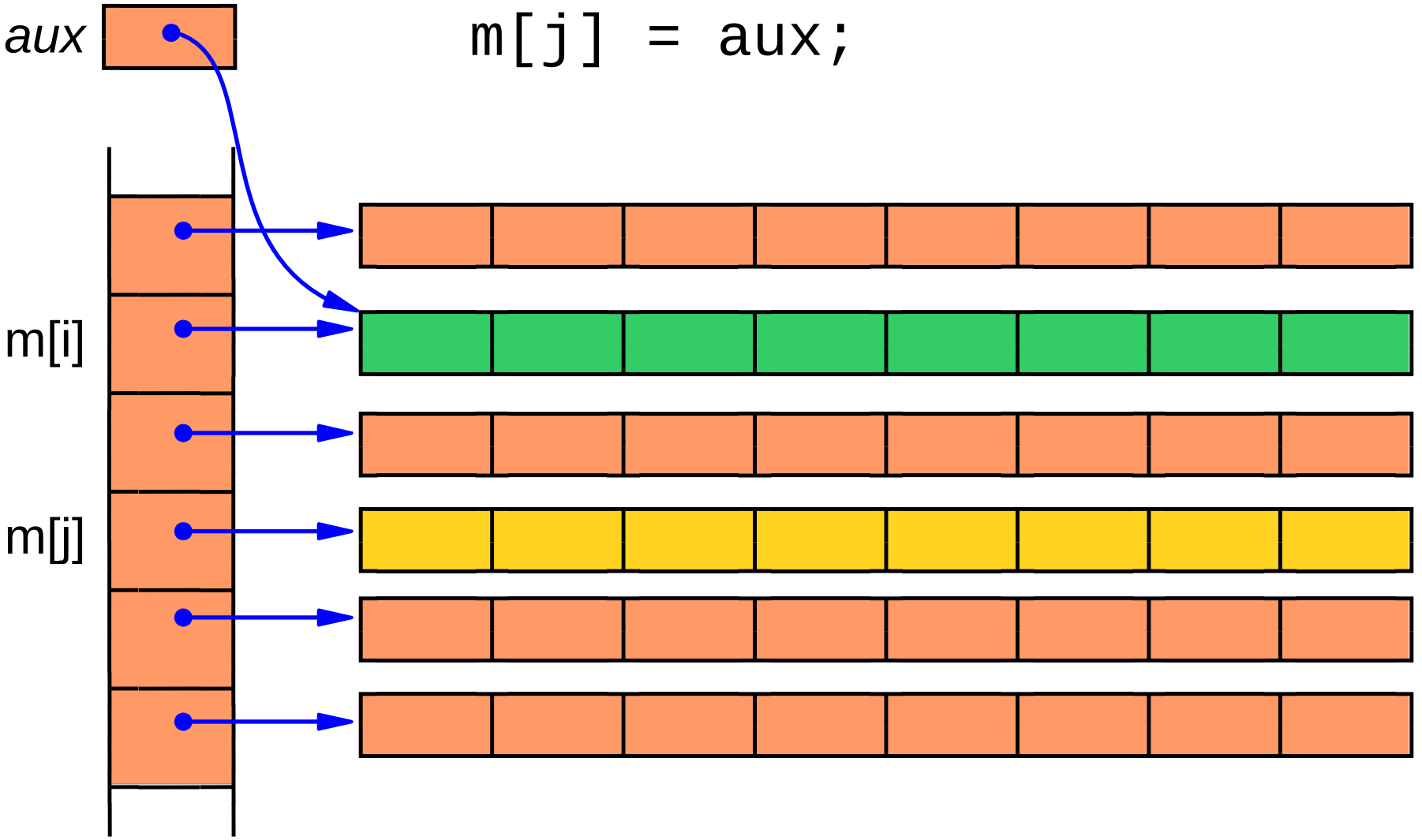
Trocar m[i] com m[j]:

```
tipo *aux = m[i];
m[i] = m[j];
m[j] = aux;
```



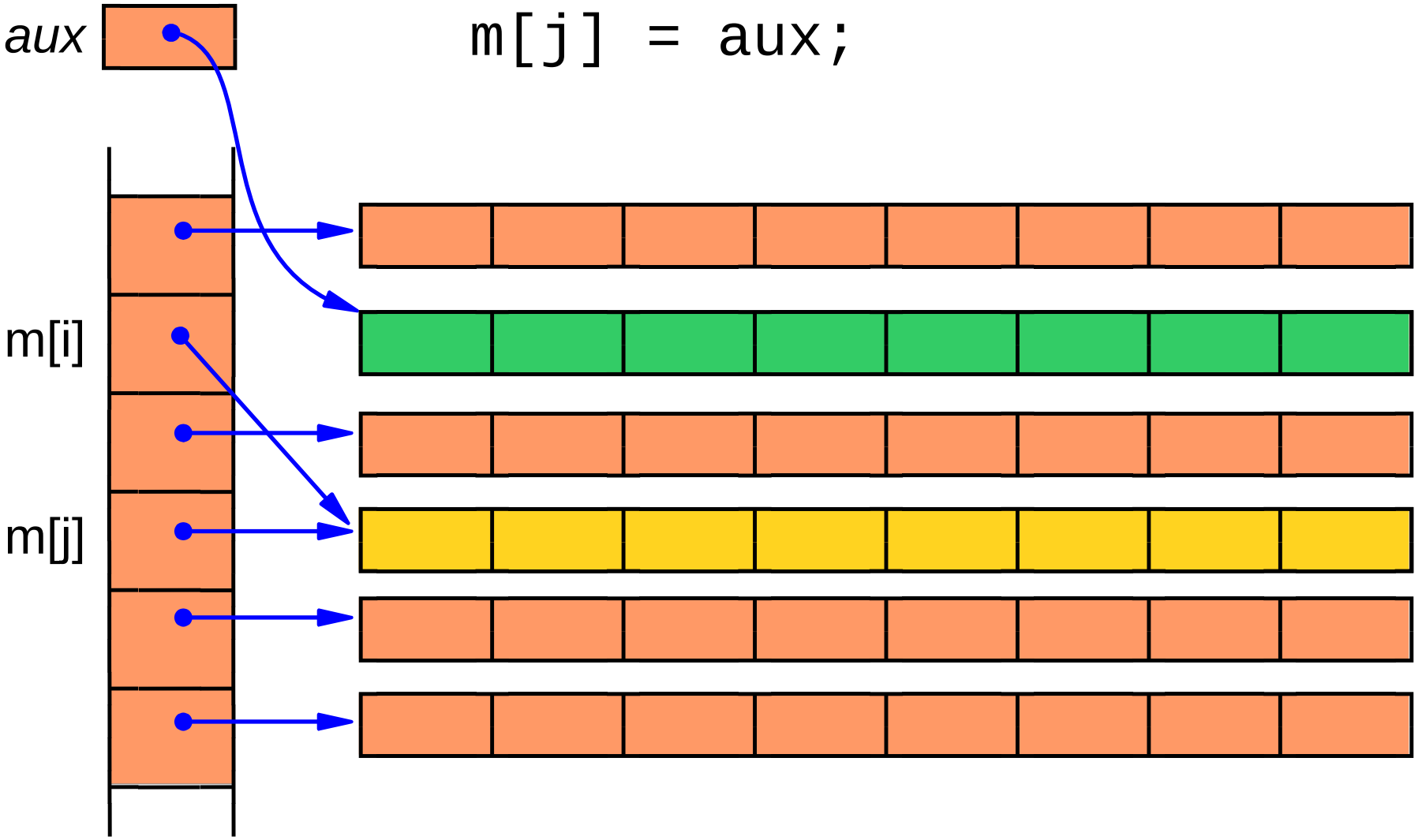
Trocar m[i] com m[j]:

```
tipo *aux = m[i];
m[i] = m[j];
m[j] = aux;
```



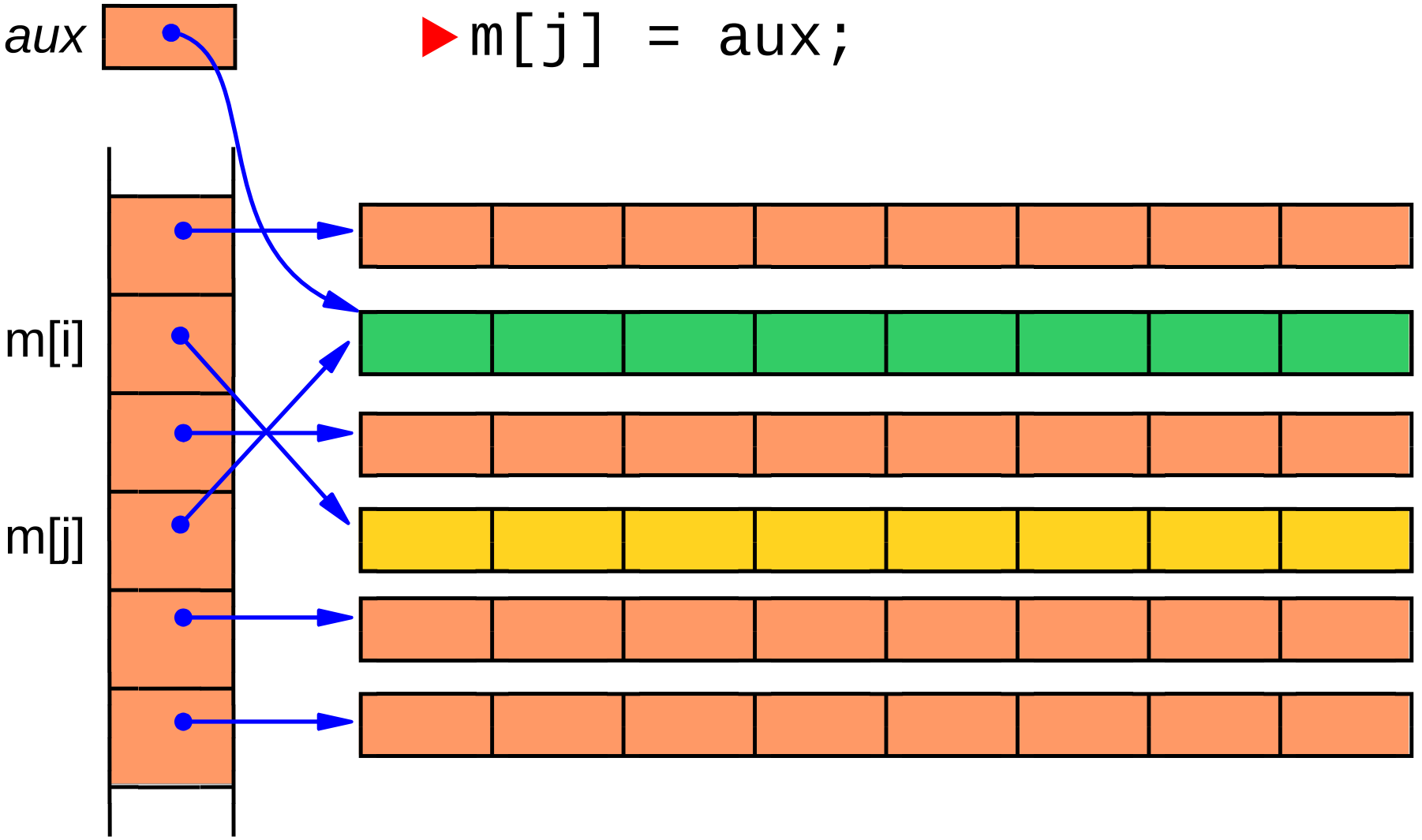
Trocar m[i] com m[j]:

```
tipo *aux = m[i];
m[i] = m[j];
m[j] = aux;
```



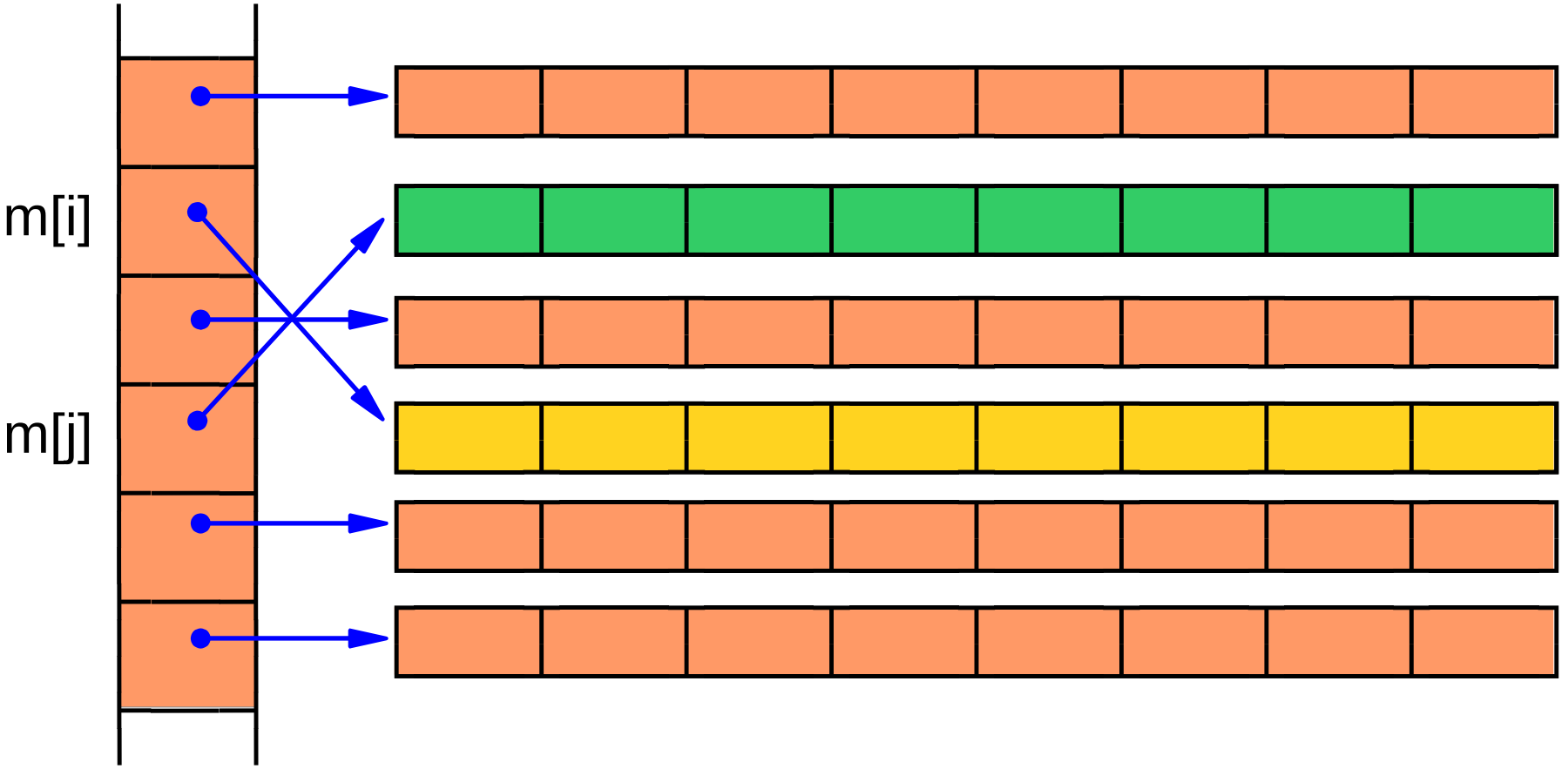
Trocar m[i] com m[j]:

```
tipo *aux = m[i];
m[i] = m[j];
m[j] = aux;
```



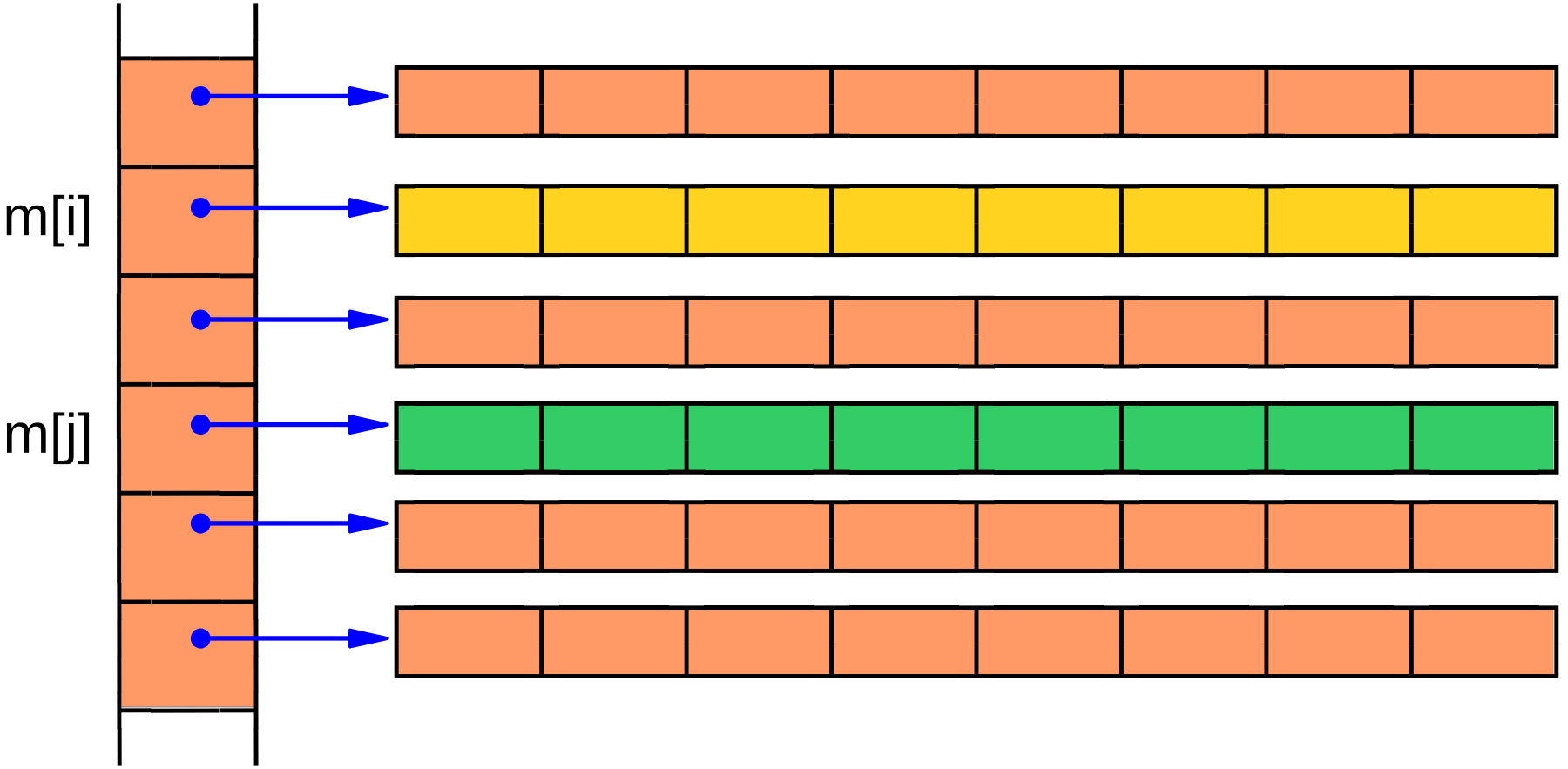
Trocar m[i] com m[j]:

```
tipo *aux = m[i];
m[i] = m[j];
m[j] = aux;
```



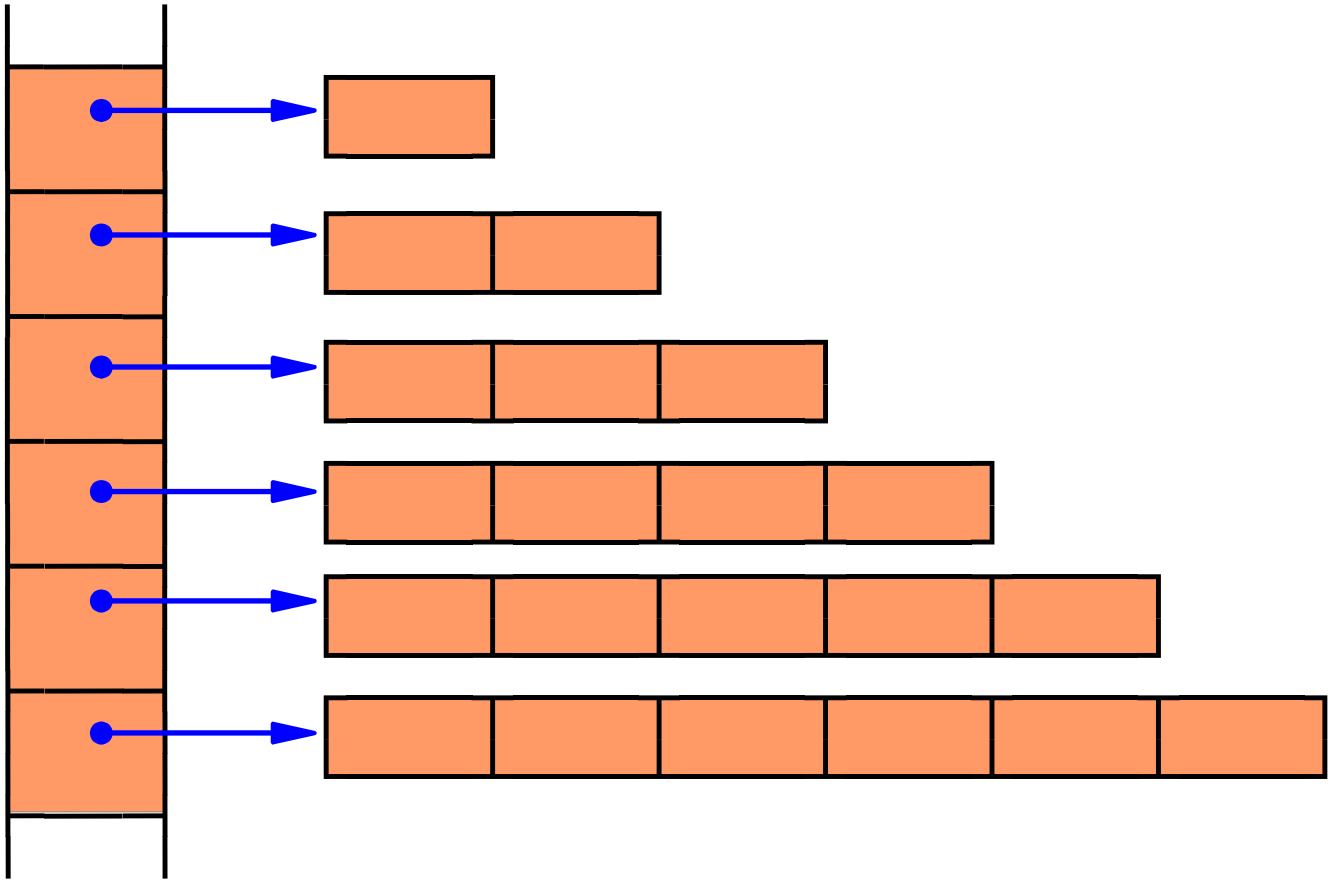
Trocar m[i] com m[j]:

```
tipo *aux = m[i];
m[i] = m[j];
m[j] = aux;
```



VANTAGENS

Se a matriz não possui todos os elementos, podemos economizar memória. Ex: matriz triangular inferior.



DESVANTAGENS DE PONTEIROS DUPLOS P/ MATRIZ

- Gerenciamento manual de memória. Isto inclui alocação e desalocação do vetor de ponteiros para as linhas, além da alocação e desalocação de cada linha.

- Representação padrão de matriz em C é totalmente diferente da forma como usamos ponteiros duplos para representar matrizes. Isto requer que o programador faça manualmente as conversões entre as duas representações caso elas sejam usadas em um mesmo programa.

- Treinaremos ponteiros duplos e matrizes na próxima aula de laboratório

- A PROVA ESTÁ CHEGANDO.
  Você está estudando?

- Mais exercícios em breve no site.