```
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

#define MAXPAROLA 30
#define MAXRIGA 80

int main(int argc, char *argv[])
{
  int freq[MAXPAROLA] ; /* vettore di contatori
  delle frequenze delle lunghezze delle parole */
  char riga[MAXRIGA] ;
  int i, inizio, lunghezza ;
  FILE * f ;

  for(i=0; i<MAXPAROLA; i++)
    freq[i]=0 ;

  if(argc != 2)
  {
    fprintf(stderr, "ERRORE, serve un parametro con il nome del file\n");
    exit(1);
  }
  f = fopen(argv[1], "rt") ;
  if(f==NULL)
  {
    fprintf(stderr, "ERRORE, impossibile aprire il file %s\n", argv[1]);
    exit(1);
  }

  while( fgets( riga, MAXRIGA, f ) != NULL )
```

Dynamic Memory Allocation

Dynamic 2-Dimensional Arrays

Stefano Quer

Dipartimento di Automatica e Informatica

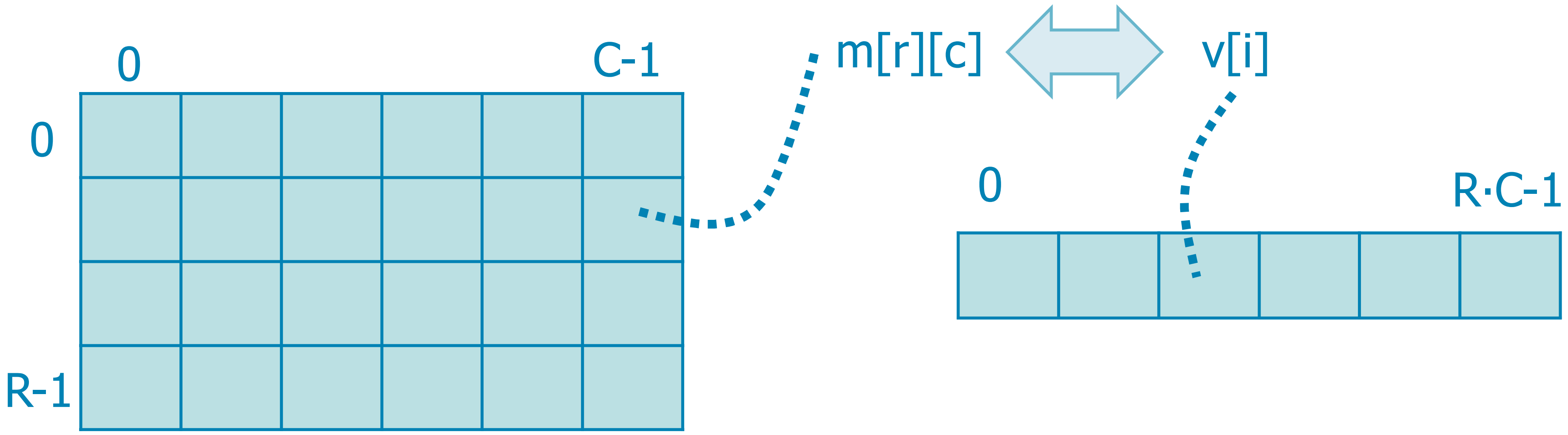
Politecnico di Torino

Problem definition

- Two-dimensional arrays can be allocated in two different ways
  - As a single 1D array including all elements
    - Easy syntax for allocation and manipulation
    - Difficult manipulation logic
  - As an array of pointers to 1D arrays of elements
    - Difficult syntax for allocation and manipulation
    - Standard manipulation logic

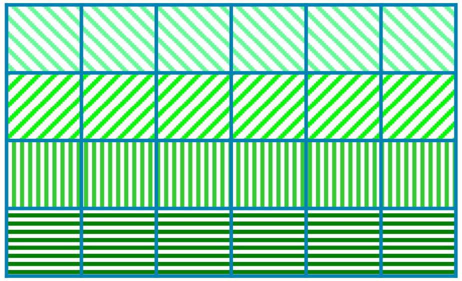
2D as 1D

- To allocate a matrix of R rows and C columns we can allocate a one-dimensional array
  - We must reserve (R · C) contiguous elements
  - Perform on-the-fly conversion 2D→1D and vice-versa

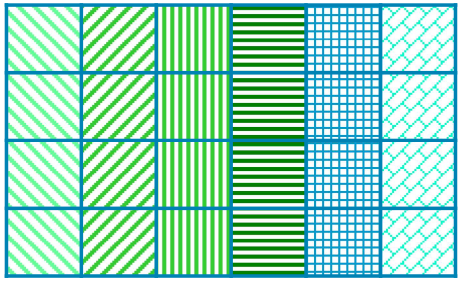


2D as 1D

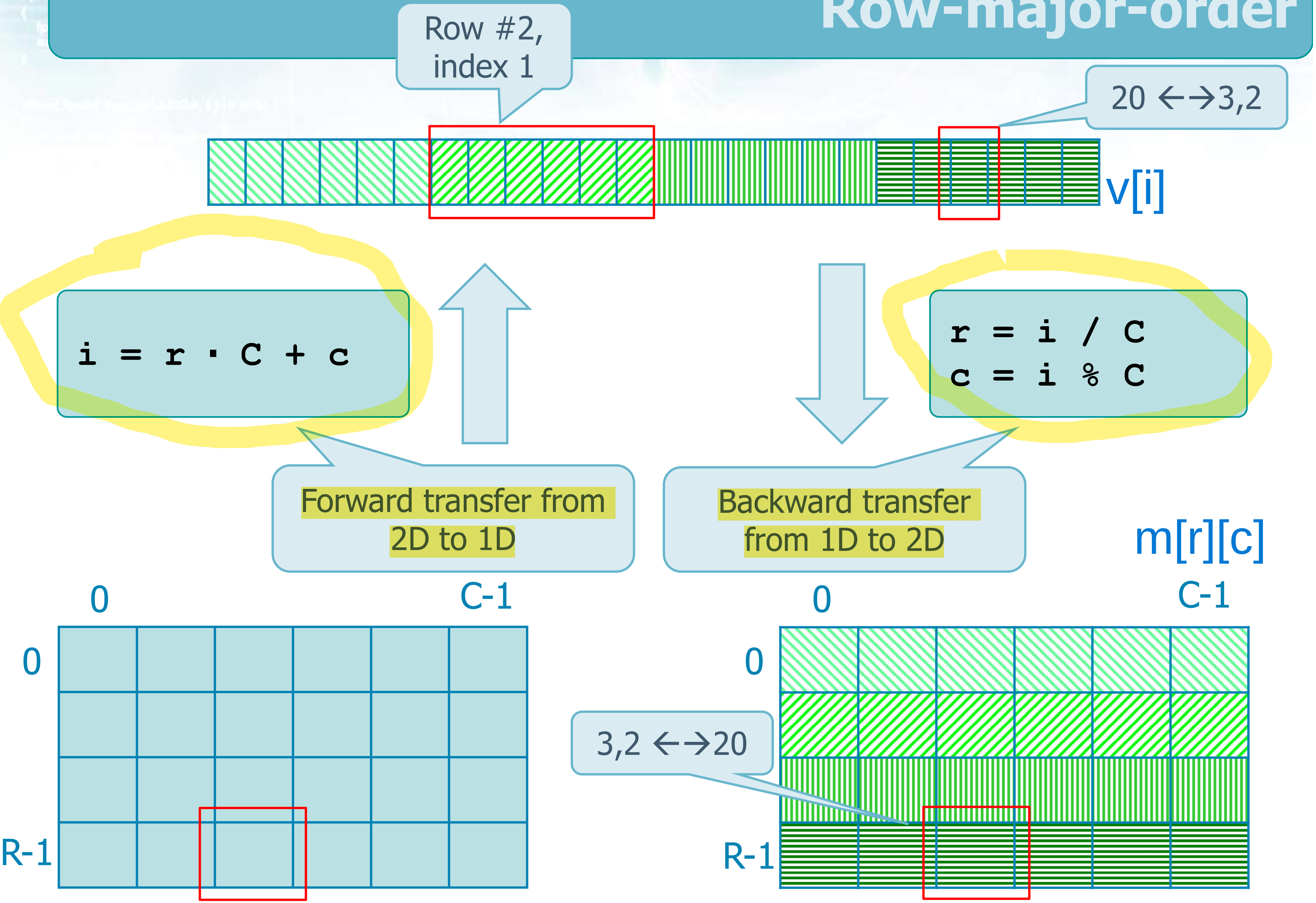
- The linearization is feasible following two different schemes
  - Row-major-order
    - Rows are allocated one after the other, with their elements in contiguous cells
    - Used in Pascal, C, C++, Python, and others



  - Column-major-order
    - Columns are allocated one after the other, with their elements in contiguous cells
    - Used in FORTRAN, OpenGL, Open CL ES, MATLAB, and others



Row-major-order



Row #2, index 1

20 ←→3,2

```
i = r · C + c
```

```
r = i / C
c = i % C
```

Forward transfer from 2D to 1D

Backward transfer from 1D to 2D

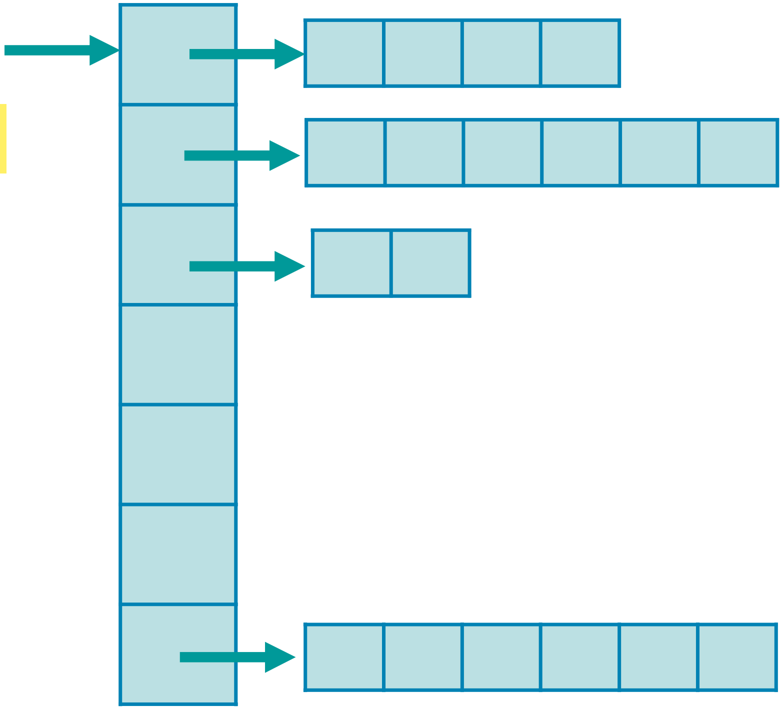
3,2 ←→20

Considerations

- The row-major-order scheme is **automatically** used by any C compiler every time a multi-dimensional array is defined
  - As all other data structures, arrays are stored in the computer memory
  - The computer memory is a **linear** array of cell
  - Thus, all multi-dimensional data structure require **linearization** to be stored internally

2D as 2D

- To allocate an array of R pointers to arrays of C elements, we must
  - Allocate one array of R pointers
    - Each pointer references one entire array of basic elements representing the corresponding row
  - Allocate R arrays of C basic elements
    - One array each row
    - Previous pointers must reference the correct array



2D Allocation

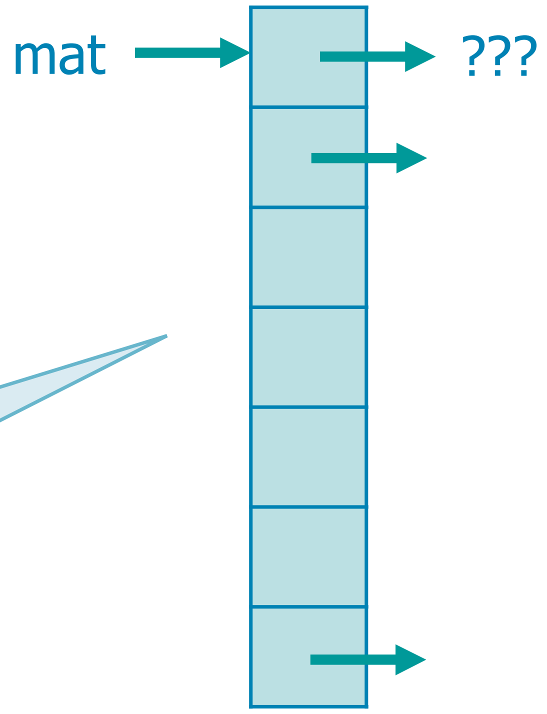
1

We generate a pointer to pointers

Elements are pointers

```
mat = (int **) malloc (r * sizeof (int *));
if (mat == NULL) { ... }
```

We work on integer values. The same reasoning applies on all other **types**



First, we allocate the main array of pointers

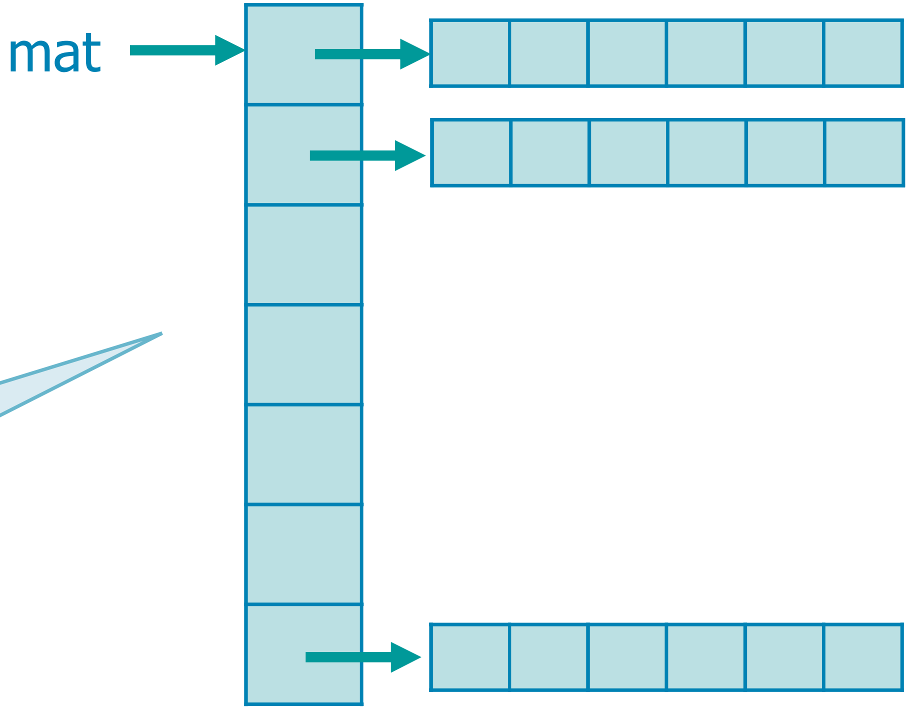
2D Allocation

2

```
for (i=0; i<r; i++) {
  mat[i] = (int *) malloc (c * sizeof (int));
  if (mat[i] == NULL) { ... }
}
```

We work on integer values. The same reasoning applies on all other **types**

If c is fixed it is a rectangular matrix



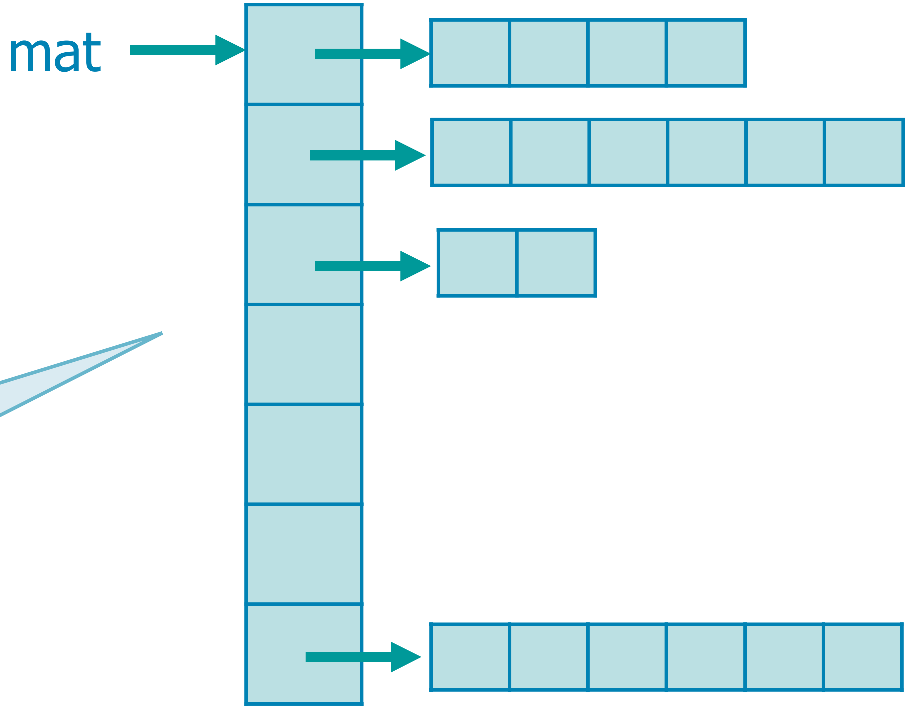
Then, we allocate the set of secondary arrays of elements

2D Allocation

```
for (i=0; i<r; i++) {
  mat[i] = (int *) malloc (c * sizeof (int));
  if (mat[i] == NULL) { ... }
}
```

The numbe of elements may vary in each row

We work on integer values. The same reasoning applies on all other variable types

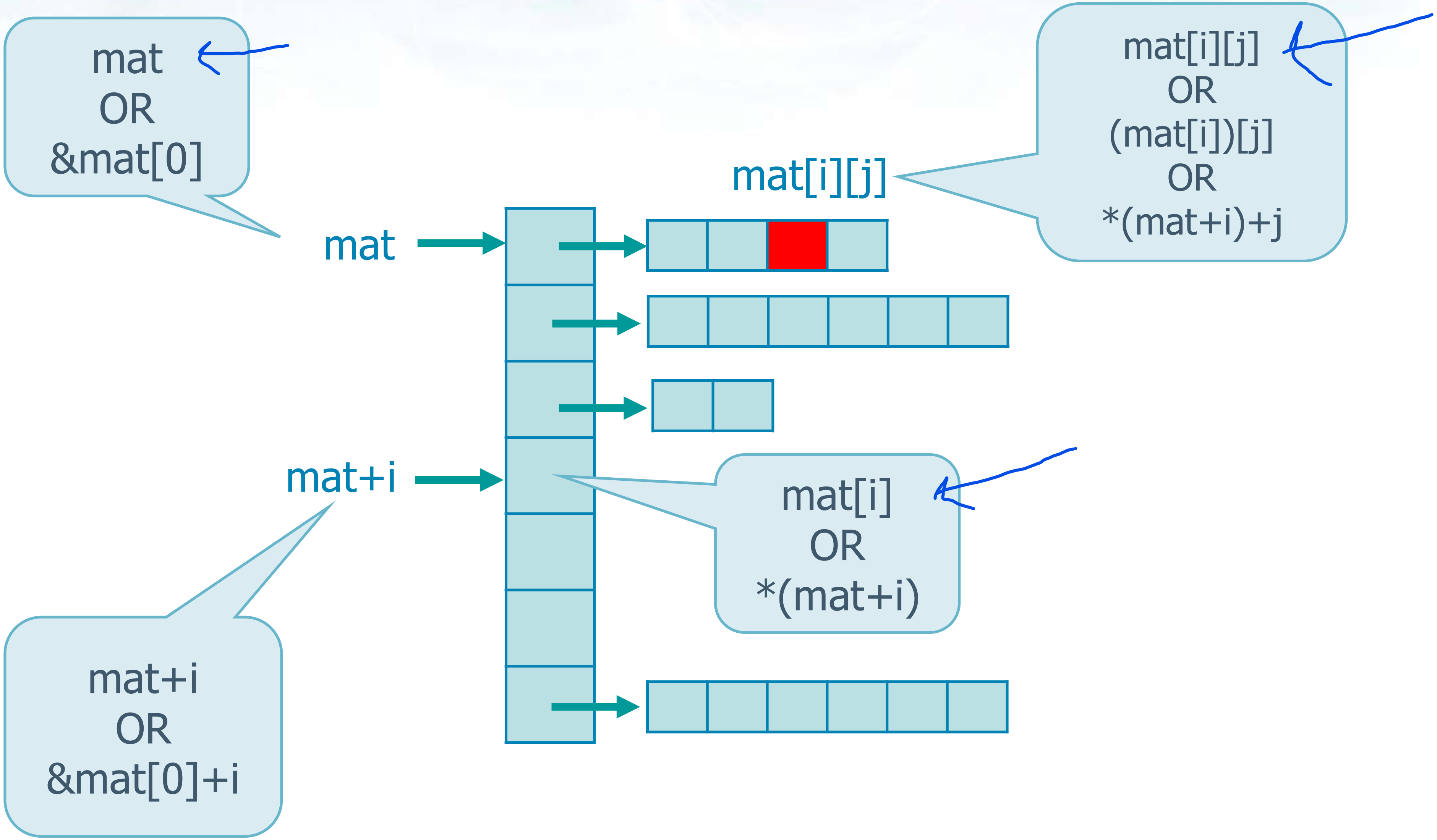


The secondary arrays can have different length

Matrix manipulation

- The easiest way to reach each matrix element is to use the standard matrix notation
  - mat[i][j] or (mat[i])[j]
    - Indicates a single element
    - It is a value
  - mat[i]
    - Indicates an entire row
    - It is a pointer to an array of values
  - mat
    - Indicates the entire matrix
    - It is a pointer to an array of pointers

Matrix manipulation



mat
OR
&mat[0]

mat[i][j]
OR
(mat[i])[j]
OR
*(mat+i)+j

mat[i][j]

mat

mat+i

mat[i]
OR
*(mat+i)

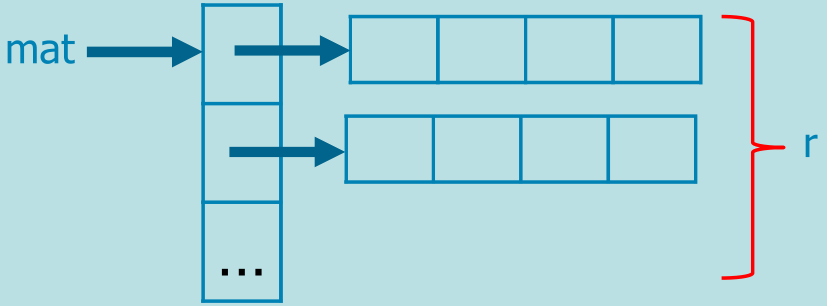
mat+i
OR
&mat[0]+i

=&mat[i]

Dispose the matrix

- As usual, dynamic data structure must be **deallocated**
- To free the data structure we must
  - First, free all secondary arrays (the rows)
  - Then, free the primary array pointers after

```
for (i=0; i<r; i++) {
  free (mat[i]);
}
free (mat);
```

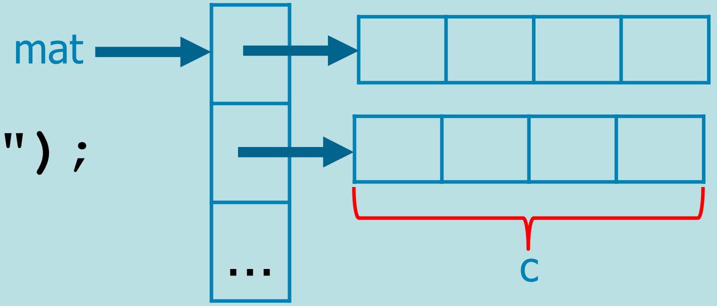


Example

2D matrix of integers

Main array allocation

```
int r, c, i;
int **mat;
printf ("Number of rows: ");
scanf ("%d", &r);
mat = (int **) malloc (r * sizeof (int *));
if (mat == NULL) {
   fprintf (stderr, "Memory allocation error.\n");
   exit (1);
}
printf ("Number of columns: ");
scanf ("%d", &c);
for (i=0; i<r; i++) {
  mat[i] = (int *) malloc (c * sizeof (int));
  if (mat[i] == NULL) {
    fprintf (stderr, "Memory allocation error.\n");
    exit (1);
  }
}
```



if not
rect mat

Secondary array allocation

Example

2D matrix of integers

```
for (i=0; i<r; i++) {
  for (j=0; i<c; j++) {
    printf (“mat[%d][%d]:”);
    scanf (“%d”, &mat[i][j]);
}

...

for (i=0; i<r; i++) {
  free (mat[i]);
}
free (mat);
```

Matrix manipulation

Matrix manipulation goes on

Matrix dispose

Example

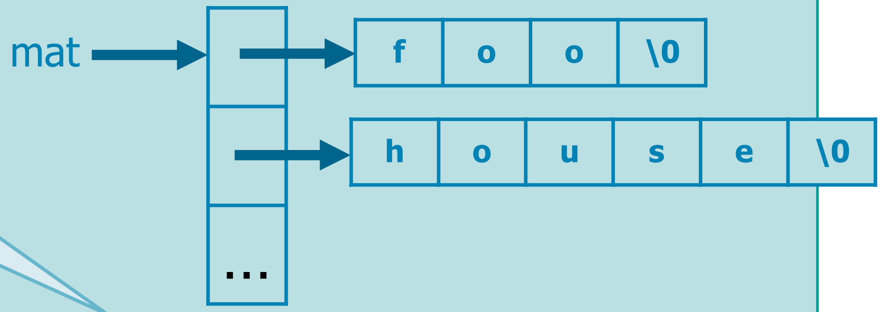
2D matrix of characters

```
int r, c, i;
char str[N], **mat;
printf ("Number of rows: ");
scanf ("%d", &r);
mat = (char **) malloc (r * sizeof (char *));
if (mat == NULL) {
   fprintf (stderr, "Memory allocation error.\n");
   exit (1);
}

for (i=0; i<r; i++) {
  scanf (“%s”, str);
  mat[i] = malloc ((strlen(str)+1) * sizeof (char));
  if (mat[i] == NULL) {
    fprintf (stderr, "Memory allocation error.\n");
    exit (1);
  }
}
```

The matrix can be used to store strings

Do not forget “+1”



The code goes on the previous way …

Common errors

static alloc

```
int mat[6][5];

sizeof (mat) ⇔ 6*5*sizeof(int) = 6*5*4 = 120
sizeof (mat[i]) ⇔ 5*sizeof(int) = 5*4 = 20
sizeof (mat[i][j]) ⇔ sizeof(int) = 4
```

dynamic alloc

```
int r=6, c=5, **mat;
mat = (int **) malloc (r * sizeof (int *));
for (i=0; i<r; i++) {
  mat[i] = (int *) malloc (c * sizeof (int));
}

sizeof (mat) ⇔ size of pointer, 8 (or 4)
sizeof (mat[i]) ⇔ size of a pointer, 8 (or 4)
sizeof (mat[i][j]) ⇔ sizeof(int) = 4
```

2D arrays and modularity

- As for 1D arrays, also 2D arrays may be made visible outside the environment in which they have been allocated
- As for 1D arrays, it is possible to
  - 1 Use **global** variables to contain the matrix pointer never done
  - 2 Adopt the **return** statement to return it easiest way
  - 3 Pass the pointer to the matrix by reference most general
    - Unfortunately, the pointer to the matrix is already a 2-star object (indirect reference)
    - To pass it by reference, we have to use a 3-star object (a reference to a reference of a reference)

Example

Array of characters

```
char **mat;
...
mat = malloc2d (nr, nc);
```

2

We return the pointer

easiest solution

does not work with multiple obj

```
char **malloc2d (int r, int c) {
  int i;
  char **mat;
  mat = (char **) malloc (r * sizeof(char *));
  if (mat == NULL) { ... }
  for (i=0; i<r; i++) {
    mat[i] = (char *) malloc(c * sizeof (char));
    if (mat[i]==NULL) { ... }
  }
  return (mat);
}
```

Example

Array of characters

```
char **mat;
...
malloc2d (&mat, nr, nc);
```

We use a 3-* object with a temporary 2* object as a support

```
void malloc2d (char ***m, int r, int c) {
  int i;
  char **mat;
  mat = (char **) malloc (r * sizeof(char *));
  if (mat == NULL) { ... }
  for (i=0; i<r; i++) {
    mat[i] = (char *) malloc(c * sizeof (char));
    if (mat[i]==NULL) { ... }
  }
  *m = mat;
  return;
}
```

hardest

most general case

dynamically speaking they are equivalent

Example

Array of characters

```
char **mat;
...
malloc2d (&mat, nr, nc);
```

We use a 3-* object without any support

without any support hence pointer is needed

```
void malloc2d (char ***m, int r, int c) {
  int i;
  (*m) = (char **) malloc (r * sizeof(char *));
  if (m == NULL) { ... }
  for (i=0; i<r; i++) {
    (*m)[i] = (char *) malloc(c * sizeof (char));
    if ((*m)[i]==NULL) { ... }
  }
  return;
}
```

The parenthesis are necessary

Example

- Do not forget to free the matrix …

```
void free2d (char **m, int r) {
  int i;
  for (i=0; i<r; i++) {
    free (m[i]);
  }
  free (m);
  return;
}
```

```
void free2d (char ***m, int r) {
  int **mat, i;
  mat = *m;
  for (i=0; i<r; i++) {
    free (mat[i]);
  }
  free (mat);
  m = NULL;
  return;
}
```

Version to set the original pointer to NULL

Observations

- All previous techniques can be applied to any type
  - Integer, float, character, C structures