

## Tipos de datos estructurados: Arrays multidimensionales

Facultad de Informática  
Universidad Complutense

Ana Gil Luezas  
(Adaptadas del original de Luis Hernández Yáñez)



# Índice

---

|   |    |
|---|----|
| Arrays bidimensionales (matrices)           | 2  |
| Recorrido                                   | 7  |
| Matrices con dimensiones variables acotadas | 14 |
| Ejemplo: Imagen                             | 18 |
| Búsqueda                                    | 22 |
| Ejemplo: Imagen en imagen                   | 28 |
| Recorrido de vecinos                        | 30 |
| Diagonales                                  | 31 |
| Arrays con dimensión variable acotada       | 33 |
| Ejemplo: Histograma                         | 37 |
| Ejemplo: Filtrar imagen                     | 42 |
| Arrays multidimensionales                   | 48 |



## Arrays bidimensionales



# Arrays bidimensionales

## *Arrays de dos dimensiones*

```
typedef tipo_base nombre[tamaño1][tamaño2];
```

El array tendrá dos dimensiones: *tamaño1* filas por *tamaño2* columnas.

```
typedef int tMatriz[50][100];
```

```
tMatriz matriz;
```

matriz es una tabla bidimensional de 50 filas por 100 columnas:

|     | 0   | 1   | 2   | 3   | ... | 98  | 99  |
|-----|-----|-----|-----|-----|-----|-----|-----|
| 0   |     |     |     |     | ... |     |     |
| 1   |     |     |     |     | ... |     |     |
| 2   |     |     |     |     | ... |     |     |
| ... | ... | ... | ... | ... | ... | ... | ... |
| 48  |     |     |     |     | ... |     |     |
| 49  |     |     |     |     | ... |     |     |

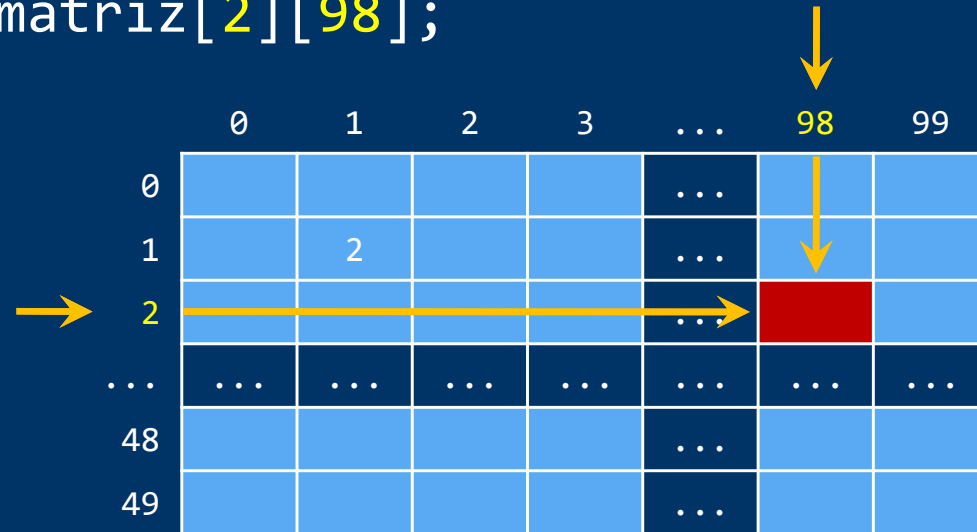


# Arrays bidimensionales

## *Arrays de dos dimensiones, acceso directo*

Ahora cada elemento del array se localiza con dos índices, uno por cada dimensión : *array[filas][columna]*

```
typedef int tMatriz[50][100];  
tMatriz matriz;  
matriz[2][98] = 0;  
cout << matriz[2][98];
```



```
matriz[  
    matriz[1][1]][98] += 1;  
    2
```



# Arrays bidimensionales

*La memoria es de una dimensión: secuencia de celdas*

Los elementos de un array bidimensional se colocan en la memoria por filas: para cada valor del primer índice todos los valores del segundo.

|             |         |   |        |
|-------------|---------|---|--------|
|             | Memoria |   |        |
| cuads[0][0] | 1       | } | fila 0 |
| cuads[0][1] | 1       |   |        |
| cuads[1][0] | 2       | } | fila 1 |
| cuads[1][1] | 4       |   |        |
| cuads[2][0] | 3       | } | fila 2 |
| cuads[2][1] | 9       |   |        |
| cuads[3][0] | 4       |   |        |
| cuads[3][1] | 16      |   |        |
| cuads[4][0] | 5       |   |        |
| cuads[4][1] | 25      |   |        |

`int cuads[5][2];`

5 filas de 2 columnas  
de enteros



# Arrays bidimensionales

---

## *Inicialización*

Para cada valor del primer índice todos los valores del segundo:

Para cada *fila* (de 0 a FILAS – 1):

Para cada *columna* (de 0 a COLUMNAS – 1):

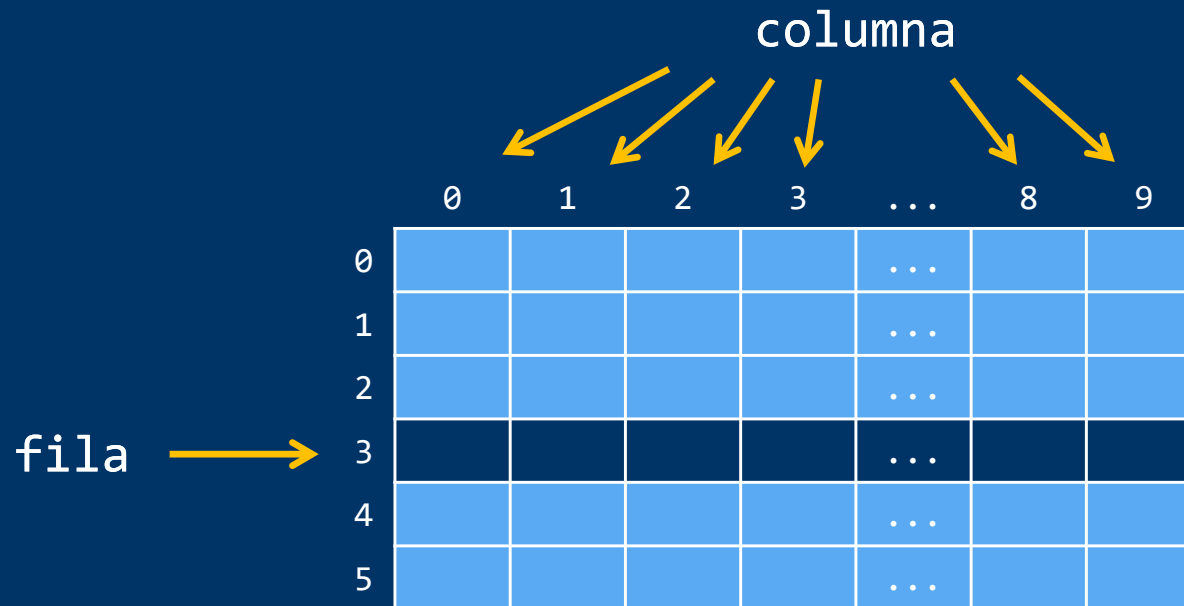
```
const int FILAS = 6;  
const int COLUMNAS = 10;  
typedef int tMatriz[FILAS][COLUMNAS];  
tMatriz matriz;
```

```
for (int fila = 0; fila < FILAS; ++fila)  
    for (int columna = 0; columna < COLUMNAS; ++columna)  
        matriz[fila][columna] = 0;
```



# Arrays bidimensionales

```
void iniciar(tMatriz/*sal*/ matriz, tElem vi){//tElem=int
    for (int fila = 0; fila < FILAS; ++fila)
        for (int columna = 0; columna < COLUMNAS; ++columna)
            matriz[fila][columna] = vi;
}
```





# Arrays bidimensionales

---

## *Recorrido por filas*

```
const int FILAS = ...;
const int COLUMNAS = ...;
typedef tElem tMatriz[FILAS][COLUMNAS];
tMatriz matriz;

... recorrer(tMatriz/*ent/sal?*/ matriz, ...) {
    for (int fila = 0; fila < FILAS; ++fila)
        for (int columna = 0; columna < COLUMNAS; ++columna)
            // Procesar matriz[fila][columna];
}
```



# Arrays bidimensionales

---

*Ejemplo: Suma de matrices*

```
const int DimMat = 30;
typedef double tMat[DimMat][DimMat]; // tElem = double

void suma(tMat const mat1, tMat const mat2,
          tMat/*sal*/ matR) {
    for (int fila = 0; fila < DimMat; ++fila)
        for (int columna = 0; columna < DimMat; ++columna)
            matR[fila][columna] = mat1[fila][columna] +
                                   mat2[fila][columna];
}
```



# Arrays bidimensionales

---

*Ejemplo: Mostrar matriz por filas*

```
typedef int tMatriz[FILAS][COLUMNAS]; // tElem = int

void mostrar(tMatriz const mat) {
    for (int fila = 0; fila < FILAS; ++fila) {
        for (int columna = 0; columna < COLUMNAS; ++columna)
            cout << mat[fila][columna] << ' ';
        cout << endl;
    }
}
```



# Arrays bidimensionales

---

*Ejemplo: Leer matriz por filas*

```
const int DimMat = 30;
typedef double tMat[DimMat][DimMat];

void leer(tMat/*sal*/ mat) {
    for (int fila = 0; fila < DimMat; ++fila)
        for (int columna = 0; columna < DimMat; ++columna)
            cin >> mat[fila][columna];
}
```



# Arrays bidimensionales

---

*Ejemplo: Producto de la diagonal*

```
const int DimMat = 30;
typedef double tMat[DimMat][DimMat];

double productoDiagonal(tMat const mat) {
    double prod = 1.0;
    for (int i = 0; i < DimMat; ++i)
        prod *= mat[i][i];
    return prod;
}

// diagonal principal: fila == col
```



# Arrays bidimensionales

---

*Ejemplo: Intercambiar dos filas*

```
typedef double tMatriz[FILAS][COLUMNAS];

bool interFilas(tMatriz /*ent/sal*/ mat, int f1, int f2){
    if (f1 < 0 || f1 >= FILAS || f2 < 0 || f2 >= FILAS)
        return false;
    else {
        if (f1 != f2)
            for (int c = 0; c < COLUMNAS; ++c)
                intercambiar(mat[f1][c], mat[f2][c]);
        return true;
    }
}
```



# Arrays bidimensionales

## *Matrices con dimensiones variables acotadas*

```
const int MAX_DIM = 10; // Tamaño máximo estimado > 0
typedef struct {
    int numFilas, numCols;
    double elementos[MAX_DIM][MAX_DIM];
} tMatriz;
tMatriz matriz;
```

```
matriz.numFilas = 6; matriz.numCols = 5;
// Parte ocupada: [0..matriz.nFilas)x[0..matriz.numCols)
for (int fila = 0; fila < matriz.numFilas; ++fila)
    for (int col = 0; col < matriz.numCols; ++col)
        matriz.elementos[fila][col] = 0.0;
```

NO MAX\_DIM



# Arrays bidimensionales

```
void iniciar(tMatriz & /*sal*/ mat, int nf, int nc, double vi) {  
    mat.numFilas = nf; mat.numCols = nc;  
    for (int fila = 0; fila < mat.numFilas; ++fila)  
        for (int col = 0; col < mat.numCols; ++col)  
            mat.elementos[fila][col] = vi;  
}  
iniciar(matriz, 6, 5, 0);
```

numFilas es 6 y  
MAX\_DIM es 10

|   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |   |   |   |   |   |
| 1 | 0 | 0 | 0 | 0 | 0 |   |   |   |   |   |
| 2 | 0 | 0 | 0 | 0 | 0 |   |   |   |   |   |
| 3 | 0 | 0 | 0 | 0 | 0 |   |   |   |   |   |
| 4 | 0 | 0 | 0 | 0 | 0 |   |   |   |   |   |
| 5 | 0 | 0 | 0 | 0 | 0 |   |   |   |   |   |
| 6 |   |   |   |   |   |   |   |   |   |   |
| 7 |   |   |   |   |   |   |   |   |   |   |
| 8 |   |   |   |   |   |   |   |   |   |   |
| 9 |   |   |   |   |   |   |   |   |   |   |

numCols es 5 y  
MAX\_DIM es 10

Parte ocupada:  
[0..nF)x[0..nC)





# Arrays bidimensionales

```
... recorrer(tMatriz /*const*/& /*ent/sal?*/ mat ...) {
    for (int fila = 0; fila < mat.numFilas; ++fila)
        for (int col = 0; col < mat.numCols; ++col)
            // Procesar mat.elementos[fila][col];
}

typedef struct {int fila; int col;} tCoor;

bool buscar(tMatriz const& mat, tCoor & /*sal*/ pos) {
    bool enc = false; pos.fila = 0;
    while (pos.fila < mat.numFilas && !enc) {
        pos.col = 0;
        while (pos.col < mat.numCols && !enc)
            if (prop(mat.elementos[pos.fila][pos.col]))
                enc = true;
            else ++pos.col;
        if (!enc) ++pos.fila;
    }
    return enc;
}
```

Parámetro de entrada: const&  
Parámetro de salida o ent/sal: &



# Arrays bidimensionales

*Ejemplo: Mostrar matriz por filas (con una columna singular)*

```
void mostrar(tMatriz const& mat) {  
    for (int fila = 0; fila < mat.numFilas; ++fila) {  
        cout << '\t' << mat.elementos[fila][0];  
        for (int col = 1; col < mat.numCols; ++col)  
            cout << ', ' << mat.elementos[fila][col];  
        cout << '\n';  
    }  
}  
  
void mostrar(tMatriz const& mat) {  
    for (int fila = 0; fila < mat.numFilas; ++fila) {  
        for (int col = 0; col < mat.numCols-1; ++col)  
            cout << mat.elementos[fila][col] << ', ';  
        cout << mat.elementos[fila][mat.numCols-1] << '\n';  
    }  
}
```

Usar var. auxiliar  
`int ultCol = mat.numCols-1`

# Ejemplo: Imagen (bmp)

```
typedef unsigned int uint;    // entero (32/64 bits) sin signo
typedef unsigned short int usint; // entero pequeño sin signo
typedef unsigned char uint8; // entero sin signo de 8 bits (byte)
typedef struct { uint8 rojo, verde, azul; } tRGB;
                                // cantidad de cada componente

const usint Max_Res = 24; // máximo nº de filas y columnas
typedef struct {
    usint numFilas, numCols; // resolución de la imagen
    tRGB bmp[Max_Res][Max_Res]; // Max_Res*Max_Res colores
} tImagen;
```

tRGB color;

|     |   |     |
|-----|---|-----|
| 255 | 0 | 128 |
|-----|---|-----|

tImagen imagen; imagen.numFilas = 9; imagen.numCols = 16;

imagen.bmp

|         |         |           |     |       |       |       |
|---------|---------|-----------|-----|-------|-------|-------|
| 0 0 0   | 255 0 0 | 255 0 128 | ... | - - - | - - - | - - - |
| ...     | ...     | ...       | ... | ...   | ...   | ...   |
| 255 0 0 | - - -   | - - -     | ... | - - - | - - - | - - - |

imagen.bmp[0][2] = color; // los struct se pueden asignar



# Ejemplo: Operadores con colores

---

```
bool operator == (tRGB const& c1, tRGB const& c2) {  
    return c1.azul == c2.azul && c1.rojo == c2.rojo  
        && c1.verde == c2.verde;  
}
```

```
bool operator != (tRGB const& c1, tRGB const& c2) {  
    return !(c1 == c2);  
}
```

```
void mostrar(tRGB const& color) {  
    cout << "Rojo: " << uint(color.rojo) << '\n';  
    cout << "Verde: " << uint(color.verde) << '\n';  
    cout << "Azul: " << uint(color.azul) << endl;  
}
```



# Ejemplo: Girar una imagen

---

```
void girar(tImagen /*ent/sal*/ imagen) {
    tImagen aux = imagen; // los struct se pueden asignar
    imagen.numFilas = aux.numCols;
    imagen.numCols = aux.numFilas;
    for (usint f = 0; f < aux.numFilas; ++f)
        for (usint c = 0; c < aux.numCols; ++c)
            imagen.bmp[c][f] = aux.bmp[f][c]; // ???
    // giro derecha
        // imagen.bmp[c][aux.numFilas-1 - f] = aux.bmp[f][c];
    // giro izquierda
        // imagen.bmp[aux.numCols-1 - c][f] = aux.bmp[f][c];
}
```

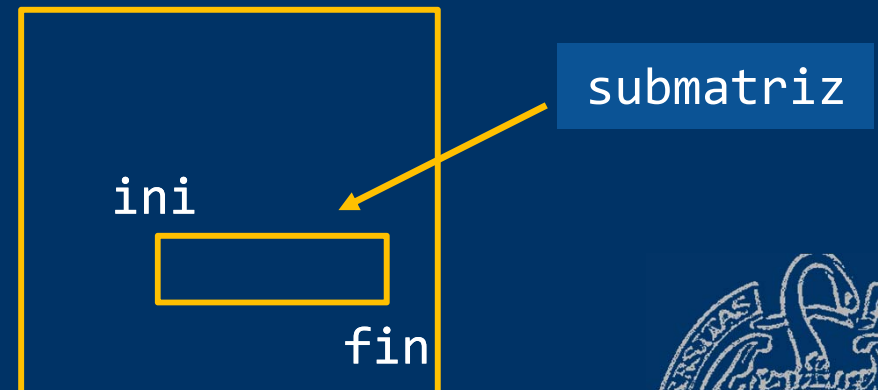


# Ejemplo: Rellenar una imagen

```
typedef struct {uint fila; uint col;} tCoor;
```

```
inline void rellenar(tImagen & /*ent/sal*/ imagen, tRGB const& vi){  
    rellenar(imagen, {0, 0}, {imagen.numFilas, imagen.numCols}, vi);  
}
```

```
void rellenar(tImagen & /*ent/sal*/ imagen, /* rellenar submatriz */  
             tCoor const& ini, tCoor const& fin, tRGB const& vi){  
    for (usint f = ini.fila; f < fin.fila; ++f)  
        for (usint c = ini.col; c < fin.col; ++c)  
            imagen.bmp[f][c] = vi;  
}
```



# Arrays bidimensionales

---

*Búsqueda del primer elemento que cumple una propiedad*

```
const int MAX = ...;
typedef struct { int numFilas, numCols;
                tElem elementos[MAX][MAX]; } tMatriz;
typedef struct {int fila; int col;} tCoor;
```

*Localizar la posición:*

```
bool buscar(tMatriz const& mat, tCoor & /*sal*/ pos) {
    bool enc = false; pos.fila = 0;
    while (pos.fila < mat.numFilas && !enc) {
        pos.col = 0;
        while (pos.col < mat.numCols && !enc)
            if (prop(mat.elementos[pos.fila][pos.col],...)) enc = true;
            else ++pos.col;
        if (!enc) ++pos.fila;
    }
    return enc;
}
```




# Arrays bidimensionales

*Búsqueda del primer elemento que cumple una propiedad*

```
bool buscar(tMatriz const& mat, tCoor & /*sal*/ pos) {  
    bool enc = false; pos.fila = 0;  
    while (pos.fila < mat.numFilas && !enc) {  
        enc = buscar(mat, pos.fila, pos.col); // buscar en una fila  
        if (!enc) ++pos.fila;  
    }  
    return enc;  
}
```

```
bool buscar(tMatriz const& mat, int fila, int & /*sal*/ col) {  
    bool enc = false; col = 0;  
    while (col < mat.numCols && !enc)  
        if (prop(mat.elementos[fila][col],...)) enc = true;  
        else ++col;  
    return enc;  
}
```





# Arrays bidimensionales

---

*Búsqueda del primer elemento que cumple una propiedad*

```
tMatriz matriz; tCoor pos;
```

```
bool buscar(tMatriz const& mat, tCoor & /*sal*/ pos);  
// no es necesario inicializar pos
```

```
bool buscarDesde(tMatriz const& mat,  
                 tCoor & /*ent/sal*/ pos);
```

```
// pos tiene que tener un valor a partir del cual se buscará  
pos = { 0, 0 };  
while (buscarDesde(matriz, pos)) ->
```



# Arrays bidimensionales

---

## *Búsqueda desde una posición*

```
bool buscarDesde(tMatriz const& mat, tCoor& /*ent/sal*/ pos){
    bool enc = false; // pos ya tiene un valor
    while (pos.fila < mat.numFilas && !enc) {
        while (pos.col < mat.numCols && !enc)
            if (prop(mat.elementos[pos.fila][pos.col],...))
                enc = true;
            else ++pos.col;
        if (!enc) {++pos.fila; pos.col = 0; }
    }
    return enc; // pos puede haber cambiado de valor
}
```



# Arrays bidimensionales

---

*Ejemplo: Leer matriz por filas detectando errores*

```
bool leer(tMatriz & /*sal*/ mat, istream & /*ent/sal*/ flujo) {
    flujo >> mat.numFilas >> mat.numCols;
    bool ok = !flujo.fail() && mat.numFilas<= MAX ...&& mat.numCols>= 0;
    int fila = 0;
    while (fila < mat.numFilas && ok) {
        int col = 0;
        while (col < mat.numCols && ok) {
            flujo >> mat.elementos[fila][col];
            if (flujo.fail()) ok = false;
            else ++col;
        }
        if (ok) ++fila;
    }
    return ok;
}
```



# Arrays bidimensionales

---

*Ejemplo: matriz cuadrada triangular inferior (los elementos por encima de su diagonal principal son cero)*

```
bool triangularInf(tMatriz const& mat) {  
    bool tri = mat.numFilas == mat.numCols; // cuadrada?  
    int fila = 0; int col;  
    while (fila < mat.numFilas && tri) { // por filas  
        col = fila + 1;  
        while (col < mat.numCols && tri) {  
            if (mat.elementos[fila][col] != 0.0) tri = false;  
            else ++col;  
        }  
        if (tri) ++fila;  
    }  
    return tri;  
} // diagonal principal: col == fila
```



# Ejemplo: Buscar imagen 3x3 en imagen

```
typedef tRGB tImg3x3[3][3]; // matriz 3x3

bool submatriz(tImagen const& imagen, tImg3x3 const mat,
               tCoor & /*sal*/ pos) {
    bool enc = false;
    pos.fila = 0;
    uint filas = imagen.numFilas-2, cols = imagen.numCols-2;
    while (pos.fila < filas && !enc) {
        pos.col = 0;
        while (pos.col < cols && !enc)
            if (iguales3x3(imagen, pos, mat)) enc = true;
            else ++pos.col;
        if (!enc) ++pos.fila;
    }
    return enc;
}
```



# Ejemplo: Buscar imagen 3x3 en imagen

---

```
bool iguales3x3(tImagen const& imagen, tCoor const& pos,
               tImg3x3 const mat) {
    bool iguales = true;
    uint f = 0, c;
    while (f < 3 && iguales) {
        c = 0;
        while (c < 3 && iguales)
            if (imagen.bmp[pos.fila + f][pos.col + c] != mat[f][c])
                iguales = false;
            else ++c;
        if (iguales) ++f;
    }
    return iguales;
}
```



# Arrays bidimensionales

*Recorrido de los elementos vecinos a uno dado (pos): submatriz 3x3 centrada en ese elemento*

```
typedef struct {int fila; int col;} tCoor;
const int incF[] = {-1, -1, -1, 0, 0, 1, 1, 1}; // 8 vecinos
const int incC[] = {-1, 0, 1, -1, 1, -1, 0, 1};
const int NumDirs = 8; // 8 direcciones

void vecina(tCoor const& pos, int dir, tCoor & /*sal*/ vec){
    vec.fila = pos.fila + incF[dir];
    vec.col = pos.col + incC[dir];
}

void recorrerVecinas(tMatriz /*const*/& mat, tCoor const& pos,...){
    tCoor vec;
    for (int dir = 0; dir < NumDirs; ++dir) {
        vecina(pos, dir, vec);
        if (enRango(mat, vec))
            // Procesar mat[vec.fila][vec.col] //procesar(mat, vec);
    }
}

bool enRango(tMatriz const& mat, tCoor const& pos) { return ... }
```



# Arrays bidimensionales

*Recorrido de diagonales en una matriz cuadrada  $N \times N$ :*

Diagonal principal:  $N$  elementos con  $\text{fila} == \text{col}$  ( $k == 0$ )

$N-1$  diagonales superiores ( $k: 1 \dots N-1$ ):  $N-k$  elementos con  $\text{col} == \text{fila} + k$

$N-1$  diagonales inferiores ( $k: 1 \dots N-1$ ):  $N-k$  elementos con  $\text{fila} == \text{col} + k$

|   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 1 |   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 2 |   |   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 3 |   |   |   | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| 4 |   |   |   |   | 0 | 1 | 2 | 3 | 4 | 5 |
| 5 |   |   |   |   |   | 0 | 1 | 2 | 3 | 4 |
| 6 |   |   |   |   |   |   | 0 | 1 | 2 | 3 |
| 7 |   |   |   |   |   |   |   | 0 | 1 | 2 |
| 8 |   |   |   |   |   |   |   |   | 0 | 1 |
| 9 |   |   |   |   |   |   |   |   |   | 0 |

Diagonal  $k$ :  
 $\text{fila} = \text{col} + k$   
 $\text{elemento}[\text{col}+k][\text{col}]$   
para  $0 \leq \text{col} < N-k$

Diagonal  $k$ :  
 $\text{col} = \text{fila} + k$   
 $\text{elemento}[\text{fila}][\text{fila}+k]$   
para  $0 \leq \text{fila} < N-k$





# Arrays bidimensionales

*Ejemplo: matriz cuadrada triangular inferior (por diagonales)*

*Diagonales superiores ( $k: 1 \dots N-1$ ) todas cero*

```
bool triangularInf(tMatriz const& mat) { // por diagonales
    bool tri = mat.numFilas == mat.numCols; // cuadrada?
    int k = 1, fila;
    while (k < mat.numCols && tri){ // diagonal k: col == fila + k
        fila = 0;
        while (fila < mat.numFilas - k && tri) {
            if (mat.elementos[fila][fila + k] != 0.0) tri = false;
            else ++fila;
        }
        if (tri) ++k;
    }
    return tri;
} // diagonal principal: k==0 (col == fila)
```



# Arrays de dimensión variable acotada

*Array con un máximo de elementos + Contador de elementos*

```
const int TM = ...; // Tamaño máximo estimado: TM > 0  
typedef tDato tArray[TM]; // Por ejemplo tDato= float
```

El array y el contador están mutuamente relacionados ->  
usamos una estructura para encapsularlos:

```
typedef struct {  
    int contador;  
    tArray datos; } tLista;  
tLista lista;
```

Parte ocupada: 0..contador-1

lista

datos

|        |       |        |        |        |        |        |   |   |      |
|--------|-------|--------|--------|--------|--------|--------|---|---|------|
| 125.40 | 76.95 | 328.80 | 254.62 | 435.00 | 164.29 | 316.05 |   |   |      |
| 0      | 1     | 2      | 3      | 4      | 5      | 6      | 7 | 8 | TM-1 |

contador 7

Nº de elementos (primer índice libre)



# Arrays de dimensión variable acotada

```
const int MAX = 100; // Tamaño estimado > 0
typedef struct {
    int cont; // Primer índice libre <-> nº de elementos
    tDato datos[MAX]; // Hasta 100 elementos
} tLista; // Parte ocupada de 0 a (cont-1), resto libre
tLista lista; lista.cont = 0; // lista vacía
```

Recorrido de la lista: De los elementos de la parte ocupada

Parámetro de entrada: const&  
Parámetro de salida o ent/sal: &

```
void recorrer(tLista /*const*/& lista, ...) {
    for (int i = 0; i < lista.cont; ++i)
        // Procesar lista.datos[i]
}
```




# Arrays de dimensión variable acotada

```
const int MAX = 100; // Tamaño estimado > 0
typedef struct {
    int cont; // Primer índice libre <-> nº de elementos
    tDato datos[MAX]; // Hasta 100 elementos
} tLista; // Parte ocupada de 0 a (cont-1), resto libre
tLista lista; lista.cont = 0; // lista vacía
```

Búsqueda en la lista: En la parte ocupada

```
bool buscar(tLista const& lista, ..., int & pos) {
    pos = 0; bool enc = false;
    while (pos < lista.cont && !enc)
        if (... lista.datos[pos]... ) enc = true; // encontrado en pos
        else ++pos;
    return enc;
}
```



Propiedad que se busca



# Arrays de dimensión variable acotada

---

Insertar nuevo último (push\_back)

```
bool push_back(tLista & lista, tDato const& dato) {  
    if (lista.cont == MAX) return false; // lista llena  
    else {  
        lista.datos[lista.cont] = dato; lista.cont += 1;  
        return true;  
    }  
}
```

Eliminar el último (pop\_back)

```
bool pop_back(tLista & lista) {  
    if (lista.cont == 0) return false; // lista vacía  
    else { lista.cont -= 1; return true; }  
}
```



# Ejemplo: Histograma de una imagen

```
const uint MaxColores = Max_Res*Max_Res; // 24*24 = 576 colores
typedef struct {tRGB color; uint cont;} tColorCont;
typedef struct {
    uint numColores; // parte ocupada 0..numColores-1
    tColorCont frecuencias[MaxColores];
} tHistograma;
```

tHistograma histograma;

|              |       |         |           |     |       |       |       |
|--------------|-------|---------|-----------|-----|-------|-------|-------|
| .numColores  | 3     |         |           |     |       |       |       |
| .frecuencias | 0 0 0 | 255 0 0 | 255 0 128 | ... | - - - | - - - | - - - |
|              | 83    | 21      | 40        | ... | -     | -     | -     |
|              | 0     | 1       | 2         |     | 141   | 142   | 143   |



# Ejemplo: Histograma de una imagen

```
bool buscar(tHistograma const& histograma, tRGB const& color, uint & /*sal*/ pos) {  
    pos = 0;  
    while (pos < histograma.numColores && histograma.frecuencias[pos].color != color)  
        ++pos;                                     // != sobrecargado  
    return pos < histograma.numColores;  
}
```

```
void insertar(tHistograma & /*ent/sal*/ histograma, tRGB const& color) {  
    uint pos;  
    if (buscar(histograma, color, pos)) histograma.frecuencias[pos].cont += 1;  
    else {  
        push_back(histograma, { color, 1 }); // sabemos que cabe  
    }  
}
```

```
void histogramaImagen(tImagen const& imagen, tHistograma & /*sal*/ histograma) {  
    histograma.numColores = 0;  
    for (uint f = 0; f < imagen.numFilas; ++f)  
        for (uint c = 0; c < imagen.numCols; ++c)  
            insertar(histograma, imagen.bmp[f][c]);  
}
```



# Ejemplo: Histograma de una imagen

```
bool guardar(tHistograma const& histograma, string const& nombArch) {
    ofstream flujo;
    flujo.open(nombArch);
    if (flujo.is_open()) {
        flujo << histograma.numColores << '\n'; // número de elementos
        for (uint i = 0; i < histograma.numColores; ++i) {
            flujo << histograma.frecuencias[i].color << ' '; // sobrecargado
            flujo << histograma.frecuencias[i].cont << '\n';
        }
        flujo.close();
        return true;
    } else return false;
}

ostream & operator <<(ostream & /*ent/sal*/ flujo, tRGB const& color) {
    flujo << uint(color.rojo) << ' ' << uint(color.verde) << ' '
        << uint(color.azul);
    return flujo;
}
```





# Ejemplo: Histograma de una imagen

```
bool cargar(tHistograma & /*sal*/ histograma, string const& nombArch) {
    ifstream flujo;
    flujo.open(nombArch);
    if (flujo.is_open()) {
        flujo >> histograma.numColores;
        for (uint i = 0; i < histograma.numColores; ++i) {
            flujo >> histograma.frecuencias[i].color; // sobrecargado
            flujo >> histograma.frecuencias[i].cont;
        } flujo.close();
        return true;
    } else return false;
}

istream & operator >>(istream & /*ent/sal*/ flujo, tRGB & /*sal*/ color) {
    uint aux;
    flujo >> aux; color.rojo = uint8(aux);
    flujo >> aux; color.verde = uint8(aux);
    flujo >> aux; color.azul = uint8(aux);
    return flujo;
}
```



# Ejemplo: Histograma de una imagen

...

```
int main() {  
    tImagen imagen = { 9, 16, { // resto 0 !!  
        {{255,0,0},{255,0,0},{255,0,0},{0,255,0},{0,255,0},{0,255,0},{0,0,255},{0,0,255},{0,0,255}},  
        {{255,0,0},{255,0,0},{255,0,0},{0,255,0},{0,255,0},{0,255,0},{0,0,255},{0,0,255},{0,0,255}},  
        {{255,0,0},{255,0,0},{255,0,0},{0,255,0},{0,255,0},{0,255,0},{0,0,255},{0,0,255},{0,0,255}},  
        {{255,0,0},{255,0,0},{255,0,0},{0,255,0},{0,255,0},{0,255,0},{0,0,255},{0,0,255},{0,0,255}},  
        {{255,0,0},{255,0,0},{255,0,0},{0,255,0},{0,255,0},{0,255,0},{0,0,255},{0,0,255},{0,0,255}},  
        {{255,0,0},{255,0,0},{255,0,0},{0,255,0},{0,255,0},{0,255,0},{0,0,255},{0,0,255},{0,0,255}},  
        {{255,0,0},{255,0,0},{255,0,0},{0,255,0},{0,255,0},{0,255,0},{0,0,255},{0,0,255},{0,0,255}},  
        {{255,0,0},{255,0,0},{255,0,0},{0,255,0},{0,255,0},{0,255,0},{0,0,255},{0,0,255},{0,0,255}},  
        {{255,0,0},{255,0,0},{255,0,0},{0,255,0},{0,255,0},{0,255,0},{0,0,255},{0,0,255},{0,0,255}}  
    } };  
    tHistograma histograma;  
    histogramaImagen(imagen, histograma);  
    mostrar(histograma); // histograma.numColores == 4  
    return 0;           // .frecuencias == { {{255,0,0}, 27},  
    }                   {{0,255,0}, 27}, {{0,0,255}, 27}, {{0,0,0}, 63}, ...}
```



# Ejemplo: Aplicar filtro de convolución

---

Se quiere desarrollar un subprograma `convolucion()` que, dada una imagen y una máscara de convolución (matriz de 3x3 enteros), obtenga la imagen resultante de aplicar el filtro.

Aplicar un filtro de convolución consiste en sustituir cada valor de la imagen `imagen[f][c]`, por la media de los 3x3 vecinos, ponderada por los pesos de la máscara centrada en ese elemento.

Vamos a suponer que la suma de la máscara (matriz 3x3) es de suma 1.

```
typedef int8_t tM3x3[3][3]; // matriz 3x3 de enteros (máscara)

void convolucion(tImagen const& imagen, tM3x3 const filtro,
                tImagen & /*sal*/ resultado);
```



# Ejemplo: Aplicar filtro de convolución

Imagen de entrada

|  |  |       |       |       |  |  |  |  |  |
|--|--|-------|-------|-------|--|--|--|--|--|
|  |  |       |       |       |  |  |  |  |  |
|  |  |       |       |       |  |  |  |  |  |
|  |  | $I_0$ | $I_1$ | $I_2$ |  |  |  |  |  |
|  |  | $I_3$ | $I_4$ | $I_5$ |  |  |  |  |  |
|  |  | $I_6$ | $I_7$ | $I_8$ |  |  |  |  |  |
|  |  |       |       |       |  |  |  |  |  |
|  |  |       |       |       |  |  |  |  |  |
|  |  |       |       |       |  |  |  |  |  |
|  |  |       |       |       |  |  |  |  |  |
|  |  |       |       |       |  |  |  |  |  |

Ventana de convolución

|       |       |       |
|-------|-------|-------|
| $I_0$ | $I_1$ | $I_2$ |
| $I_3$ | $I_4$ | $I_5$ |
| $I_6$ | $I_7$ | $I_8$ |

Máscara de convolución

|       |       |       |
|-------|-------|-------|
| $M_0$ | $M_1$ | $M_2$ |
| $M_3$ | $M_4$ | $M_5$ |
| $M_6$ | $M_7$ | $M_8$ |

×

Imagen de salida

|  |  |  |  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|--|--|--|
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |

Nuevo píxel =  $I_0 \times M_0 + I_1 \times M_1 + I_2 \times M_2 +$   
 $I_3 \times M_3 + I_4 \times M_4 + I_5 \times M_5 +$   
 $I_6 \times M_6 + I_7 \times M_7 + I_8 \times M_8$



# Ejemplo: Aplicar filtro de convolución

```
void convolucion(tImagen const& imagen, tM3x3 const filtro,
                tImagen & /*sal*/ resultado) {
    // iniciar dimensiones
    resultado.numFilas = imagen.numFilas;
    resultado.numCols = imagen.numCols;
    // copiar el borde
    ... →
    // rellenar el resto de la imagen calculando la convolución por posición
    uint ultF = imagen.numFilas-1;
    uint ultC = imagen.numCols-1;
    for (uint f = 1; f < ultF; ++f)
        for (uint c = 1; c < ultC; ++c)
            resultado.bmp[f][c] = convolucion(imagen,{f,c},filtro);
}
tRGB convolucion(tImagen const& imagen, tCoor pos, tM3x3 const filtro);
```



# Ejemplo: Aplicar filtro de convolución

```
void convolucion(tImagen const& imagen, tM3x3 const filtro,
                tImagen & /*sal*/ resultado) {
    // iniciar dimensiones
    ...
    // copiar el borde
    uint ultF = imagen.numFilas-1; // primera y última fila
    for (uint c = 0; c < imagen.numCols; ++c) {
        resultado.bmp[0][c] = imagen.bmp[0][c];
        resultado.bmp[ultF][c] = imagen.bmp[ultF][c];
    }
    uint ultC = imagen.numCols-1; // primera y última columna
    for (uint f = 1; f < ultF; ++f) {
        resultado.bmp[f][0] = imagen.bmp[f][0];
        resultado.bmp[f][ultC] = imagen.bmp[f][ultC];
    }
    // rellenar el resto de la imagen calculando la convolución
    // para cada posición
}
```



# Ejemplo: Aplicar filtro de convolución

```
const uint incF[] = {-1, -1, -1, 0, 0, 0, 1, 1, 1}; // vecinos 3x3
const uint incC[] = {-1, 0, 1, -1, 0, 1, -1, 0, 1};

// suponemos que pos no está el borde
tRGB convolucion(tImagen const& imagen, tCoor pos, tM3x3 const filtro) {
    tRGB rgb; int8_t peso; int r,g,b;
    r = g = b = 0;
    tCoor posCF = { 1, 1 }; // centro del filtro
    for (uint d = 0; d < 9; ++d) {
        rgb = imagen.bmp[pos.fila + incF[d]][pos.col + incC[d]];
        peso = filtro[posCF.fila + incF[d]][posCF.col + incC[d]];
        r += rgb.rojo * peso;
        g += rgb.verde * peso;
        b += rgb.azul * peso;
    }
    rgb.rojo = uint8(r); rgb.verde = uint8(b); rgb.azul = uint8(b);
    return rgb;
}
```



# Ejemplo: Aplicar filtro de convolución

```
tRGB operator * (tRGB rgb, int mul) {  
    rgb.rojo *= mul;  
    rgb.verde *= mul;  
    rgb.azul *= mul;  
    return rgb;  
}
```

Parámetro por valor:  
variable local iniciada  
con el argumento de llamada

```
tRGB operator / (tRGB rgb, int div) {  
    rgb.rojo /= div;  
    rgb.verde /= div;  
    rgb.azul /= div;  
    return rgb;  
}
```

```
void operator += (tRGB & /*ent/sal*/ rgb, tRGB const& rgb2) {  
    rgb.rojo += rgb2.rojo;  
    rgb.verde += rgb2.verde;  
    rgb.azul += rgb2.azul;  
}
```





## Arrays multidimensionales



# Arrays multidimensionales

---

## *Arrays de varias dimensiones*

Podemos indicar varios tamaños en la declaración de un array.

Cada uno en su par de corchetes.

```
typedef tipo_base nombre[tamaño1][tamaño2][...][tamañoN];
```

El array tendrá tantas como tamaños se indiquen.

Podemos definir tantas dimensiones como necesitemos.

```
typedef double tMatriz[5][10][20][10];
```

```
tMatriz matriz;
```

Necesitamos tantos índices como dimensiones:

```
cout << matriz[2][9][15][6];
```



# Arrays multidimensionales

## *Inicialización de arrays multidimensionales*

```
typedef double tMatriz[3][4][2][3];  
tMatriz matriz;  
for (int n1 = 0; n1 < 3; ++n1)  
    for (int n2 = 0; n2 < 4; ++n2)  
        for (int n3 = 0; n3 < 2; ++n3)  
            for (int n4 = 0; n4 < 3; ++n4)  
                matriz[n1][n2][n3][n4] = 0;
```

matriz[0][0][0][0]  
matriz[0][0][0][1]  
matriz[0][0][0][2]  
matriz[0][0][1][0]  
matriz[0][0][1][1]  
matriz[0][0][1][2]  
matriz[0][1][0][0]  
matriz[0][1][0][1]  
matriz[0][1][0][2]  
matriz[0][1][1][0]  
matriz[0][1][1][1]  
matriz[0][1][1][2]  
matriz[0][2][0][0]

Memoria

1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
0

...

0



# Arrays multidimensionales

---

## *Recorrido de arrays N-dimensionales*

```
const int DIM1 = 10;
const int DIM2 = 5;
const int DIM3 = 25;
const int DIM4 = 50;
typedef double tMatriz[DIM1][DIM2][DIM3][DIM4];
tMatriz matriz;
```

Anidamiento de bucles desde la primera dimensión hasta la última:

```
for (int n1 = 0; n1 < DIM1; ++n1)
    for (int n2 = 0; n2 < DIM2; ++n2)
        for (int n3 = 0; n3 < DIM3; ++n3)
            for (int n4 = 0; n4 < DIM4; ++n4)
                // Procesar matriz[n1][n2][n3][n4];
```






# Acerca de *Creative Commons*



## Licencia CC (*Creative Commons*)

Este tipo de licencias ofrecen algunos derechos a terceras personas bajo ciertas condiciones.

Este documento tiene establecidas las siguientes:

-  Reconocimiento (*Attribution*):  
En cualquier explotación de la obra autorizada por la licencia hará falta reconocer la autoría.
-  No comercial (*Non commercial*):  
La explotación de la obra queda limitada a usos no comerciales.
-  Compartir igual (*Share alike*):  
La explotación autorizada incluye la creación de obras derivadas siempre que mantengan la misma licencia al ser divulgadas.

Pulsa en la imagen de arriba a la derecha para saber más.

