Matrices estáticas y dinámicas, mas sobre clases y sobrecarga de operadores

Juan F. Huete

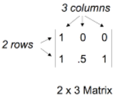
Departamento de Ciencias de la Computación e I.A.
Universidad de Granada

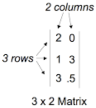
Metodología de la Programación, 2018

Índice

Matriz ...

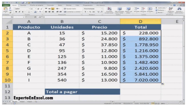
- Una matriz es un array bidimensional de elementos del mismo tipo cuyos elementos se acceden utilizando un doble índice, fila y columna





$$\begin{aligned} a_{1,1}x_1 + a_{1,2}x_2 + \cdots + a_{1,n}x_n &= b_1 \\ a_{2,1}x_1 + a_{2,2}x_2 + \cdots + a_{2,n}x_n &= b_2 \\ \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots \\ a_{m,1}x_1 + a_{m,2}x_2 + \cdots + a_{m,n}x_n &= b_m \end{aligned}$$

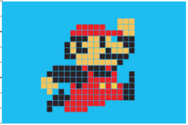
$$\equiv \quad A \cdot X = b$$

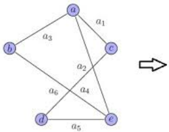


Producto	Unidades	Precio	Total
A	15	$ 15.200	$ 228.000
B	36	$ 24.800	$ 892.800
C	47	$ 37.850	$ 1.778.950
D	95	$ 12.800	$ 1.216.000
E	125	$ 11.000	$ 1.375.000
F	136	$ 10.900	$ 1.482.400
G	247	$ 9.800	$ 2.420.600
H	354	$ 16.500	$ 5.841.000
I	540	$ 13.000	$ 7.020.000
		Total a pagar	

ExpertoEnExcel.com

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9





$$\begin{pmatrix} 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \end{pmatrix}$$

Matriz ...

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & a_{m3} & \cdots & a_{mn} \end{pmatrix}$$

- En una matriz matemática los elementos son números, normalmente se denotan como $a_{i,j}$ donde i representa la fila y j la columna.
- Si n representa el número de filas y n el número de columnas, entonces la matriz tendrá $n \times m$ elementos.
- Una matriz matemática tiene definidas múltiples operaciones, por lo que será un tipo idóneo para profundizar en la sobrecarga de Operadores

- Aunque es cómodo utilizar la matriz matemática como ejemplo, en uso en informática va mucho mas allá.

Esquema

Arrays estáticos

- Declaración: Para declarar un array bidimensional estático de elementos tenemos que indicar tanto las filas como las columnas

```
1 char pagina[30][100];
2 int  array2[2][3];
```

- Acceso elementos: Cada elemento individual del array A se obtiene de la forma A[i][j], donde $0 \leq i < filas$ y $0 \leq j < columnas$.

```
1  pagina[2][4]='x';
2  array2[0][1]=150;
3
4 for (int f =0;f<30;f++){
5   for (int c=0; c<100; c++)
6     cout << pagina[f][c];
7   cout << endl;
8   }
```

- Inicializar: Podemos utilizar paréntesis para inicializar un array bidimensional

```
1 int  array2[2][3]  = { {1,2,3}, {4,5,6} };
2 int  array2[2][3]  = { 1,2,3,4,5,6 }; // Equivalente
```

Paso arrays bidimensionales a funciones

- Podemos necesitar pasar un array bidimensional a una función

void funcion(char [ ][N], int filas, int cols);
N es una constante entera que indica el número de columnas, difícil de parametrizar

```
1  void pintaPagina(char pag[][50], int filas, int cols){
2  for (int f =0;f<filas;f++){
3    for (int c=0; c<cols; c++)
4      cout << pag[f][c];
5    cout << endl;
6    }
7  }
8  int main(){
9  char pagina[30][50]={ ...};
10 char pagCorta[10][30]= {  };
11
12 pintaPagina(pagina,30,50); // ok
13 pintaPagina(pagCorta,10,30); // Error de compilacion
```

```
1  cannot convert 'int (*)[30]' to 'int (*)[50]'
```

Arrays N-dimensionales

Se pueden declarar arrays con tantas dimensiones como se desee

```
1  const int ANIOS = 10, MESES = 12, DIAS = 31;
2  double cuantos[ANIOS][MESES][DIAS];
3
4  cuantos[1][2][21] = 3.2; // asingamos valor
5
6  void pinta(double ndim[][MESES][DIAS], int A, int M, int D){
7    for (int a = 0; a < A; a++)
8      for (int m = 0; m < M; m++)
9        for (int d = 0; d < D ; d++)
10           cout << ndim[a][m][d];
11 }
```

Problemas

Difíciles de parametrizar, necesitamos conocer el tamaño apriori, todas las columnas deben tener el mismo tamaño...

Solución

Uso de arrays dinámicos, encapsulados dentro de clases bien definidas

Índice

Esquema

Arrays Dinámicos

- Veremos como poder utilizar arrays donde se pueden reservar el tamaño del array en tiempo de ejecución y no de compilación.
- Estudiares distintas representaciones para el array bidimensional, que son fácilmente extensibles a arrays n-dimensionales
- Presentaremos distintas soluciones para arrays bidimensionales con distinto número de elementos por filas
- Utilizaremos una clase para encapsular las estructuras resultantes.

Matriz matemática

Será el modelo que utilizaremos como ejemplo

Matrices: Operaciones básicas

Especificación de las operaciones

- Acceso a los elementos $\rightarrow a_{i,j}$
- Asignación $C = A$, con $A_{n\times m}, C_{n\times m}$: $\rightarrow c_{i,j} = a_{i,j}$
- Suma(Resta) de matrices $C = A + B$, con $A_{n\times m}, B_{n\times m}, C_{n\times m}$: $\rightarrow c_{i,j} = a_{i,j} + b_{i,j}$
- Producto por un escalar, $C = A * e$:$\rightarrow c_{i,j} = a_{i,j} * e$
- Producto Matricial $C = A * B$, con $A_{n\times m}, B_{m\times r}, C_{n\times r}$: $\rightarrow c_{i,j} = \sum_{k=1}^{m} a_{i,k} * b_{k,j}$
- Trasposición $C = \sim A$, con $A_{n\times m}, C_{m\times n}$: $\rightarrow c_{j,i} = a_{i,j}$
- Operadores relacionales (igualdad, distinto)
- Operadores de entrada/salida

Especificación

```
1  class matriz {
2  public:
3      matriz ();
4      matriz (int f, int c);
5      ...
6      int & operator()(int f,int c);    // tambien version  const
7      int & at(int f, int c);    //tambien version const
8  // consultores
9      size_t getfilas()const;
10     size_t getcolumnas(int f)const;
11     string toString()const;  // para imprimir
12 // operadores
13     matriz operator+(const matriz & drcha) const ;
14     matriz operator*(int k) const ;
15     matriz operator*(const matriz & drcha) const ;
16     matriz operator~()const;
17     matriz & operator+=(const matriz & drcha);
18     ...
19      void clear();
20     ~matriz ();
21 private:
```

Representación

Representación

Implica definir las estructuras para almacenar la matriz en el ordenador.

Podemos considerar distintas alternativas, la selección de una u otra dependerá en gran medida de criterios que dependen de la aplicación a utilizar:

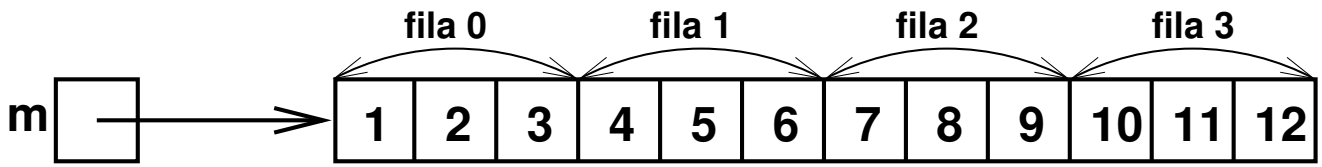
- Eficiencia de las operaciones críticas
- Memoria utilizada
- Si la matriz debe poder modificar sus dimensiones
  - Aumentar/Disminuir filas
  - Aumentar/Disminuir columnas

Consideramos 2 posibles alternativas, que llamaremos Matriz1D y Matriz2D

Índice

Matriz1D: Matriz utilizando un array

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




```
class matriz{   // Version bloque continuo
public:
     .....
private:
     int * data;
     int filas;
     int columnas;
}
```

Esquema

matriz1D: Acceso a los elementos

```
1
2  int & matriz::at(int f, int c){//acceso seguro
3      assert(0<=f && f<filas);
4      assert(0<=c && c<columnas);
5      return data[columnas*f+c];
6  }
7
8  const int & matriz::at(int f, int c)const {//acceso seguro
9      assert(0<=f && f<filas);
10     assert(0<=c && c<columnas);
11     return data[columnas*f+c];
12 }
```

matriz1D: Constructores

```
1  matriz::matriz() {
2      filas = columnas = 0;
3      data = nullptr;
4  }
5  matriz::matriz(int f, int c){
6      filas = f; columnas = c;
7      data = new int[filas*columnas];
8  }
9  matriz::matriz(int f, int c, int valor){
10     filas = f; columnas = c;
11     data = new int[filas*columnas];
12     for (int i=0; i< f*c; i++)
13        data[i]=valor;   //fill_n(data,f*c,valor); //<algorithm>
14 }
15 matriz::matriz(const matriz& orig) {
16     filas = orig.filas;  columnas = orig.columnas;
17     data = new int[filas*columnas];
18     for (int i=0; i< orig.filas*orig.columnas; i++)
19        data[i] = orig.data[i];
20        //copy_n(orig.data,filas*columnas,data);  //<algorithm>
21
```

matriz1D: Destructores y liberadores

```
1
2  void matriz::clear() {
3       if (data!=nullptr)
4         delete [] data;
5       filas = columnas = 0;
6       data = nullptr;
7    }
8   matriz::~matriz() {
9        if (data!=nullptr) delete [] data;
10    }
```

Esquema

Introducción a la sobrecarga de operadores

- C++ permite usar un conjunto de operadores con los tipos predefinidos que hace que el código sea muy **legible y fácil de entender**.
- Por ejemplo, la expresión:

$$a + \frac{b \cdot c}{d \cdot (e + f)}$$

  se calcularía en C++ con `a+(b*c)/(c*(e+f))`
- Si usamos un tipo que no dispone de esos operadores escribiríamos:
  `Suma(a,Divide(Producto(b,c),Producto(c,Suma(e,f))))`

  que es más engorroso de escribir y entender.

Recordatorio sobre sobrecarga de operadores

- C++ permite **sobrecargar** casi todos sus operadores en nuestras propias clases, para que podamos usarlos con los objetos de tales clases.
- Para ello, definiremos un método o una función cuyo nombre estará compuesto de la palabra `operator` junto con el operador correspondiente. Ejemplo: `operator+()`.
- Esto permitirá usar la siguiente sintaxis para hacer cálculos con objetos de nuestras propias clases:

```
matriz A(2,3), B(2,3), C;

C = A+B;
```

- No puede modificarse la sintaxis de los operadores (número de operandos, precedencia y asociatividad).
- No deberíamos tampoco modificar la semántica de los operadores.

Operadores que pueden sobrecargarse

+	-	*	/	%	^	&	|	~	«	»
=	+=	-=	*=	/=	%=	^=	&=	|=	»=	«=
==	!=	<	>	<=	>=	!	&&	||	++	--
->*	,	->	[]	()	new	new[]	delete		delete[]	

- Los operadores que no pueden sobrecargarse son:

.	.*	::	?:	sizeof

- Al sobrecargar un operador no se sobrecargan automáticamente operadores relacionados.

  Por ejemplo, al sobrecargar + no se sobrecarga automáticamente +=, ni al sobrecargar == lo hace automáticamente !=.

Esquema

matriz1D: Sobrecarga como función miembro

Sobrecarga como función miembro

Consiste en añadir un método a la clase, que recibirá un objeto (o ninguno para operadores unarios) de la clase y devolverá el resultado de la operación.

```
class matriz{
   public:
    matriz operator+(const matriz & drch)const;
    matriz operator*(int k) const;
    matriz operator~( ) const;
};
```

- Cuando el compilador encuentre una expresión tal como `p+q` la interpretará como una llamada al método `p.operator+(q)`
  - Sin embargo no es posible definir así el operador para usarlo con expresiones del tipo: `int k = 3; C = k*M`
    llamaría al método `int::operator(const matriz & )`, esto es `k.operator*(M)`, de ïnt", que no existe

Matriz1D: Sobrecarga como función miembro

Tenemos acceso a los campos privados de la clase.

Miembro de la clase

```
1  matriz matriz::operator+( const mathMatriz & drcha) const{
2        size_t f = filas, c = columnas ;
3
4        matriz  suma(f,c);
5        for (int i=0; i<f;i++)
6          for (int j=0; j<c;j++){
7              suma.data[c*i+j]= data[c*i+j]+ drcha.data[c*i+j];
8           }
9        return suma;
10 }
```

Sobrecarga como función externa

Sobrecarga como función externa

Consiste en añadir una función externa a la clase, que recibirá dos objetos (o uno para operadores unarios) de la clase y devolverá el resultado de la operación.

```
matriz operator+(const matriz & A, const matriz & B);
```

- Cuando el compilador encuentre una expresión tal como `p+q` la interpretará como una llamada a la función `operator+(p,q)`
- Incluso podríamos sobrecargar el operador aunque los dos operandos sean de tipos distintos:

```
matriz operator*(const matriz & A, int k);
matriz operator*(int k, const matriz & A);
...
C=A*2;
C=2*A;
```

Sobrecarga como función externa

Tenemos que utilizar la interfaz de la clase matriz, por lo que la implementación de estos métodos no depende de la representación escogida.

Externo a la clase

```
1  matriz operator+(const matriz & izq, const mathMatriz & drcha)
       {
2        size_t f = izq.getfilas(), c = izq.getcolumnas();
3
4        matriz suma(f,c);
5        for (int i=0; i<f;i++)
6            for (int j=0; j<c;j++){
7                suma.at(i,j)=izq.at(i,j) + drcha.at(i,j);
8            }
9        return suma;
10 }
```

Esquema

Sobrecarga como función miembro o externa

- La sobrecarga de un operador con una **función miembro** puede hacerse si tenemos acceso al código fuente de la clase y el primer operando es del tipo de la clase.
- El lenguaje C++ obliga a que los operadores `()`, `[]`, `->` y los operadores de asignación, sean implementados como **funciones miembro**.

```
matriz & matriz::operator=(const matriz& orig) {
    if (this!=&orig){
        clear();
        data = new int[orig.filas*orig.columnas];
        copy_n(orig.data,orig.filas*orig.columnas,data);
        filas = orig.filas; columnas = orig.columnas;
    }
    return *this;
}
int & matriz::operator ()(int f, int c){
    return data[columnas*f+c];
}
const int & matriz::operator ()(int f, int c)const{
    return data[columnas*f+c];
}
```

Esquema

Sobrecarga de operador de llamada a función

Operador de llamada a función: operator()

Obligatoriamente se implementará como función miembro.
Puede implementarse con cualquier número de parámetros (podemos tener varias versiones de este operador).
La semántica del operador, lo que hace, depende del programador

- En el tipo matriz loo utilizamos para simular el acceso directo, esto es, indexar los elementos del array considerando las filas y las columnas.

```
int & matriz::operator ()(int f, int c){
    return data[columnas*f+c];
}
const int & matriz::operator ()(int f, int c)const{
    return data[columnas*f+c];
}
int main(){
   matriz matriz(20,20);
   for (int i =0; i<20; i++)
     cout << matriz(i,i) << endl; // diagonal principal
```

Sobrecarga como función miembro o externa

- Si el primer operando debe ser un objeto de una clase diferente, debemos sobrecargarlo como **función externa**.
- También, si el primer operando debe ser un dato de un tipo primitivo, debemos sobrecargarlo como **función externa**.
  Ejemplo: El operador `*` para multiplicar entero con `matriz` lo implementaremos con una función externa.

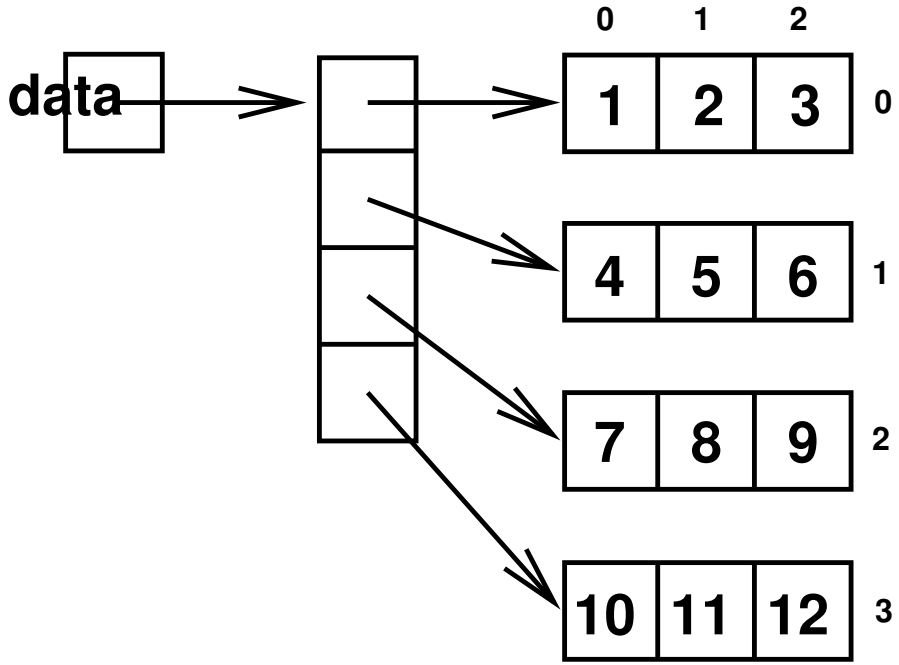
```
1  matriz  operator*(int k, const matriz & m)  {
2       size_t f = m.getfilas(), c = m.getcolumnas();
3
4       matriz prod(f,c);
5       for (int i=0; i<f;i++)
6            for (int j=0; j<c;j++){
7                 prod(i,j)=k*m(i,j);
8            }
9       return prod;
10 }
```

Índice

Esquema

Matriz2D: Matriz utilizando un array de arrays

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




```
class matriz{   // Version 1D
public:
     .....
private:
     int ** data;
     int filas;
     int columnas;
}
```

Esquema

matriz2D: Acceso a los elementos

```
1  int & matriz::operator()(int f, int c){
2      return data[f][c];
3  }
4  int & matriz2D::at(int f, int c){
5      assert(0<=f && f<filas);
6      assert(0<=c && c<columnas);
7      return data[f][c];
8
9  }
10
11 const int & matriz::operator()(int f, int c)const{
12     return data[f][c];
13 }
14 const int & matriz::at(int f, int c)const {
15     assert(0<=f && f<filas);
16     assert(0<=c && c<columnas);
17     return data[f][c];
18
19 }
```

matriz2D: Constructores

```
1  matriz::matriz() {
2      filas = columnas = 0;
3      data = nullptr;
4  }
5  matriz::matriz(int f, int c, int val){
6      filas = f; columnas = c;
7      data = new int*[filas];
8      for (int i=0;i<filas;i++){
9          data[i] =new int[columnas];
10         fill_n(data[i],columnas,val);
11     }
12 }
13 matriz::matriz(const matriz& orig) {
14     filas = orig.filas; columnas = orig.columnas;
15     data = new int*[filas];
16     for (int i=0;i<filas;i++){
17         data[i] =new int[columnas];
18         copy_n(orig.data[i],orig.columnas,data[i]);
19     }
20 }
```

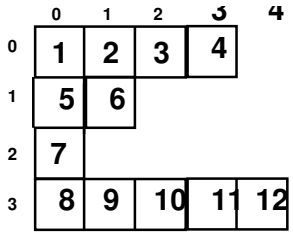
matriz2D: Destructores y liberadores

```
1  void matriz::clear(){
2      filas = columnas = 0;
3      if (data!=nullptr){
4        for (int i=0;i<filas;i++)
5          delete [] data[i];
6      delete [] data;
7      }
8      data = nullptr;
9  }
10 matriz::~matriz() {
11     if (data!=nullptr){
12     for (int i=0;i<filas;i++)
13         delete [] data[i];
14     delete [] data;
15     }
16 }
```

Índice

Esquema

raggMatriz

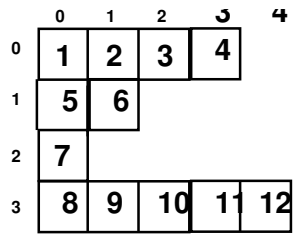


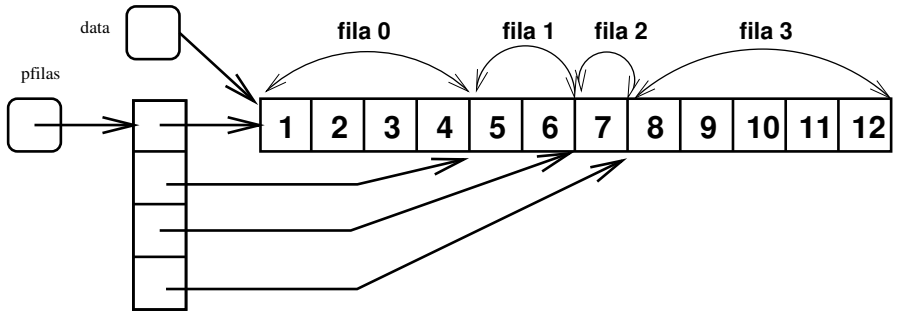
- Son muchas las aplicaciones en las que necesitamos tener un array bidimensional con un número de elementos distintos por filas
- Un ejemplo sencillo podría ser un editor de texto
- Una alternativa, utilizar un array bidimensional con un número de columnas igual al máximo posible, pero en este caso, se desperdiciaría mucha memoria.

En esta sección, veremos dos alternativas para gestionar este tipo de estructuras, que dependen de si almacenamos los datos en un bloque continuo o en bloques independientes

Esquema

raggMatriz1D: Bloque continuo de datos






```
class raggMatriz{   // Version 1D
public:
     .....
private:
      int * data;     // bloque de datos
      int ** pfilas; // un puntero al comienzo de cada fila
      int filas;      // numero de filas
      int total;      // tamano bloque de datos
}
```

Esquema

raggMatriz1D: Constructores

```
1  raggMatriz1D::raggMatriz() {
2      filas = total = 0;
3      data = nullptr;
4      pfilas = nullptr;
5  }
6
7  raggMatriz::raggMatriz(const raggMatriz& orig) {
8      filas = orig.filas;
9      total = orig.total;
10     data = new int[total];
11     pfilas = new int*[filas];
12     copy_n(orig.data,total,data);
13     for (int i=0;i<filas;i++){
14         pfilas[i]= &data[orig.pfilas[i]-orig.data];
15
16     }
17 }
```

raggMatriz1D: Constructores

```
1  raggMatriz::raggMatriz(int f, int c){
2   // todas las filas  tenen igual tamao
3       filas = f; total  = f*c;
4       data = new int[total];
5       pfilas = new int*[filas];
6       for (int i=0;i<filas;i++){
7           pfilas[i]= &data[i*c];
8       }
9  }
10 raggMatriz::raggMatriz(int f, int *c){
11 //cada fila con tamao diferente
12      filas = f; total = 0;
13      for (int i=0;i<f;i++) total+=c[i];
14      data = new int[total];
15      pfilas = new int*[filas];
16      int pos = 0;
17      for (int i=0;i<filas;i++){
18          pfilas[i]= &data[pos];
19          pos+=c[i];
20      }
```

raggMatriz1D: Acceso a los elementos

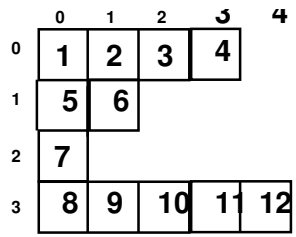
```
1  int & raggMatriz::operator ()(int f, int c){
2      return pfilas[f][c];
3  }
4  int & raggMatriz::at(int f, int c){
5      assert(0<=f && f<filas);
6      if (f!=filas-1)assert(0<=c && c< pfilas[f+1]- pfilas[f]);
7      else assert(0<=c && c<(data+total)-pfilas[f]);
8     return pfilas[f][c];
9  }
10
11 const int & raggMatriz::operator ()(int f, int c)const{
12     return pfilas[f][c];
13 }
14 const int & raggMatriz::at(int f, int c)const {
15     assert(0<=f && f<filas);
16     if (f!=filas-1)assert(0<=c && c< pfilas[f+1]- pfilas[f]);
17     else assert(0<=c && c<(data+total)-pfilas[f]);
18    return pfilas[f][c];
19 }
```

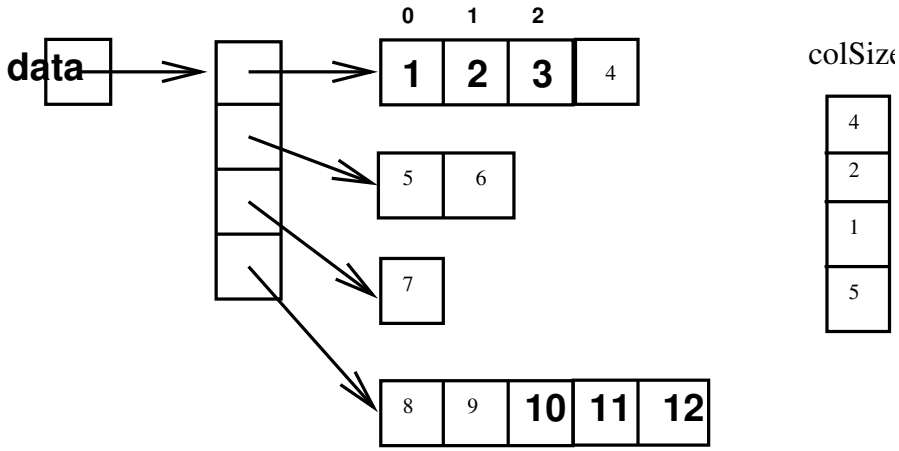
raggMatriz1D: Destructores y liberadores

```
1  raggMatriz::~raggMatriz() {
2      if (data!=nullptr) {
3        delete [] data;
4        delete [] pfilas;
5        }
6  }
7
8  void raggMatriz::clear() {
9      if (data!=nullptr) {
10       delete [] data;
11       delete [] pfilas;
12     }
13     filas = total = 0;
14     data = nullptr;
15     pfilas = nullptr;
16 }
```

Esquema

raggMatriz2D: Bloques independientes de datos






```
class raggMatriz{   // Version 2D
public:
     .....
private:
     int ** data;    // puntero a bloques de datos
     int * colSize; // array con el tamano de las columnas,
     int filas;      // numero de filas
     int total;      // tamano total
}
```

Esquema

raggMatriz2D: Constructores

```
1
2  raggMatriz::raggMatriz() {
3      filas = total = 0;
4      data = nullptr;
5      colSize = nullptr;
6  }
7
8  raggMatriz::raggMatriz(const raggMatriz& orig) {
9      filas = orig.filas;
10     total = orig.total;
11     colSize = new int[filas];
12     copy_n(orig.colSize,filas,colSize);
13     data = new int*[filas];
14     for (int i=0;i<filas;i++){
15         data[i]=new int[colSize[i]];
16         copy_n(orig.data[i],colSize[i],data[i]);
17     }
18 }
```

raggMatriz2D: Constructores

```
1  raggMatriz::raggMatriz(int f, int c){
2  // todas las filas iguales
3      filas = f;  total  = f*c;
4      colSize = new int[f];
5      for (int i=0;i<f;i++)colSize[i]=c;
6      data = new int*[filas];
7      for (int i=0;i<filas;i++){
8          data[i] =new int[colSize[i]];
9      }
10 }
11 raggMatriz::raggMatriz(int f, int *c){
12 //cada fila con numero diferente de elementos
13     filas = f;  total = 0;
14     colSize = new int[f];
15     for (int i=0;i<f;i++) total+= colSize[i]= c[i];
16     data = new int*[filas];
17     for (int i=0,; i<filas;i++){
18         data[i] =new int[colSize[i]];
19     }
20 }
```

raggMatriz2D: Acceso a los elementos

```
1  int & raggMatriz::operator ()(int f, int c){
2      return data[f][c];
3  }
4  int & raggMatriz::at(int f, int c){
5      assert(0<=f && f<filas);
6      assert(0<=c && c<colSize[f]);
7      return data[f][c];
8  }
9  const int & raggMatriz::operator ()(int f, int c)const {
10     return data[f][c];
11 }
12 const int & raggMatriz::at(int f, int c)const {
13     assert(0<=f && f<filas);
14     assert(0<=c && c<colSize[f]);
15     return data[f][c];
16 }
```

raggMatriz2D: Destructores y liberadores

```
1  raggMatriz::~raggMatriz() {
2    if (data!=nullptr){
3      for (int i=0;i<filas;i++)
4          delete [] data[i];
5      delete [] data;
6      delete [] colSize;
7     }
8  }
9
10 void raggMatriz::clear(){
11   if (data!=nullptr) {
12     for (int i=0;i<filas;i++)
13         delete []data[i];
14     delete [] data;
15     delete [] colSize;
16   }
17   filas = total = 0;
18   data = nullptr;
19   colSize = nullptr;
20 }
```