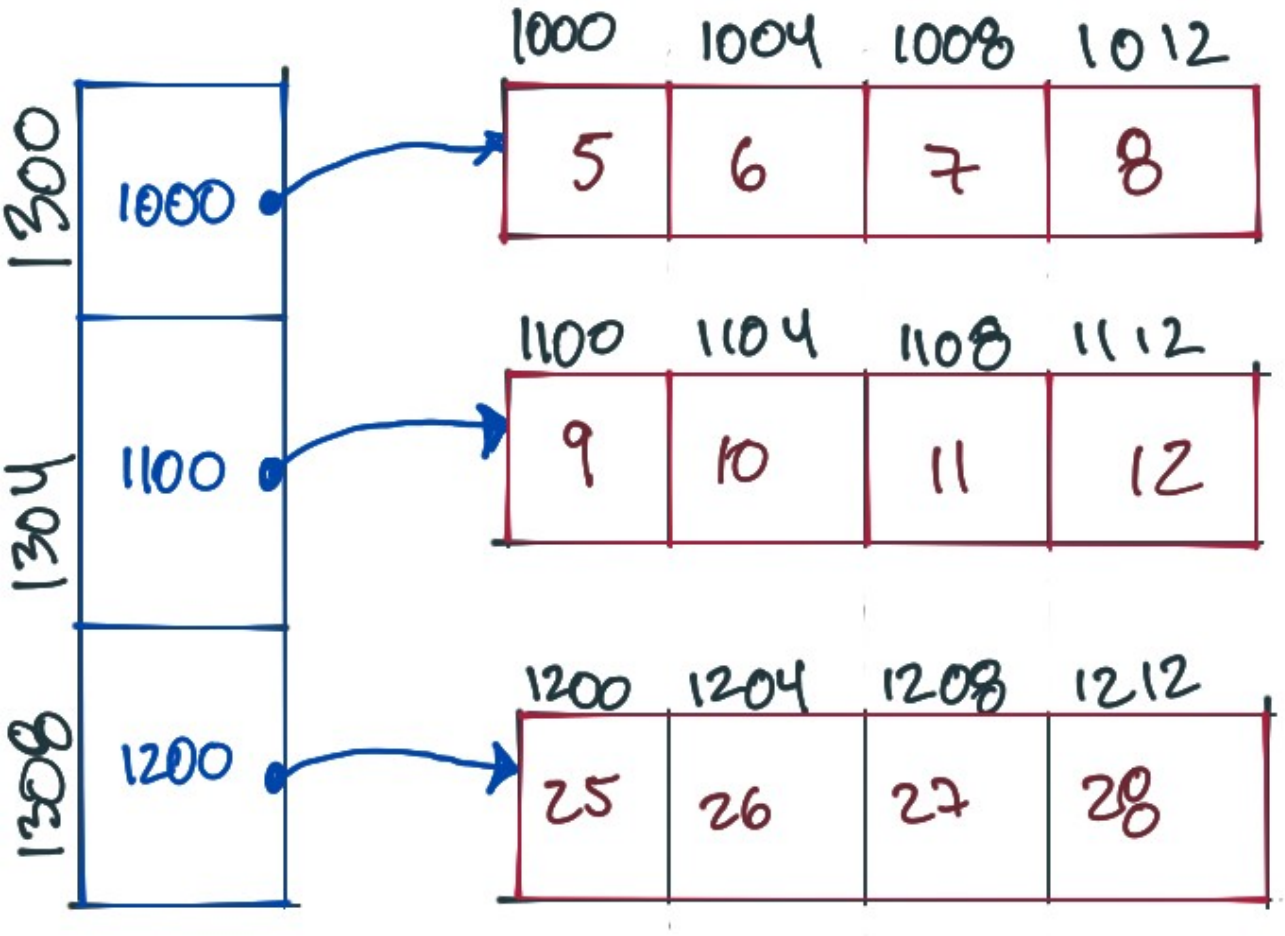
Matrices dinámicas en C

Profesor: M. en I. Fco. Javier Rodríguez García

Programación Avanzada y Métodos Numéricos, F.I. UNAM,
Octubre 2015

1. Introducción

Una matriz dinámica es en realidad un arreglo de arreglos. Tendremos tantos arreglos horizontales como filas requiera la matriz, y un arreglo vertical que mantendrá una referencia a cada uno de los arreglos horizontales. La siguiente figura muestra desde el punto de vista de un arreglo de arreglos la representación de una matriz de tres filas por cuatro columnas que almacenará números enteros:



El tamaño del arreglo vertical será el mismo que el número de filas de la matriz. El tamaño de cada arreglo horizontal será igual al número de columnas que se desee. En este caso tenemos tres arreglos con cuatro columnas cada uno. En color rojo podemos ver valores enteros arbitrarios, mientras que en negro tenemos direcciones hipotéticas de cada arreglo. El valor de estas direcciones no es importante, solamente basta con entender el concepto de arreglo de arreglos.

También es fundamental que nos quede claro que mientras los arreglos horizontales guardarán datos del tipo que nosotros necesitemos (en este caso enteros), el arreglo vertical guardará direcciones de arreglos de enteros (valores en color azul).

Una vez que tenemos en mente la imagen del arreglo de arreglos lo siguiente es codificarlo en C.

2. Matriz con número fijo de filas y columnas

El siguiente ejemplo nos muestra la forma general (para una matriz de tamaño fijo) de declarar los arreglos horizontales y el arreglo vertical, utilizando en forma intensiva los conceptos de memoria dinámica, aunque la idea de una matriz dinámica es que sea el usuario o la propia aplicación sean quien decida el tamaño de la matriz en tiempo de ejecución (es decir, cuando el programa ya está ejecutándose).

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, const char *argv[])
{
 int **mat; // (1)
 // mat almacena la dirección de arreglos que guardan direcciones de enteros

 int filas = 3;
 mat = (int **) malloc(filas * sizeof(int **)); // (2)
 // se reservó la memoria para mat, pero ésta aún no apunta a nada

 int *fila1, *fila2, *fila3;

 int cols = 3;
 fila1 = (int *) malloc(cols * sizeof(int)); // (3)
 fila2 = (int *) malloc(cols * sizeof(int)); // (3)
 fila3 = (int *) malloc(cols * sizeof(int)); // (3)
 // se crean 3 filas
 // fila1 ya se puede considerar como un arreglo de enteros; igual para las
 // otras

 mat[0] = fila1; // (4)
 mat[1] = fila2; // (4)
 mat[2] = fila3; // (4)
 // hacemos que la primer entrada de mat apunte a la primera fila; igual para
 // las otras filas
 // este paso se puede hacer antes de inicializar a las filas
```

```
 mat[0][0] = 100; // (5)
 mat[0][1] = 200; // (5)
 mat[0][2] = 300; // (5)
 // ya podemos utilizar a mat con notación de matrices

 mat[1][0] = 1000; // (5)
 mat[1][1] = 2000; // (5)
 mat[1][2] = 3000; // (5)
 // ahora estamos llenando la segunda fila con notación de matrices

 for (int i = 0; i < filas; i++) {
  for (int j = 0; j < cols; j++) {
   printf("%d\t", mat[i][j]);
  }
  printf("\n");
 }
 // imprimimos mat completa, otra vez utilizando notación de matrices

 Print(mat, filas, cols);

 free(fila1); // (6)
 free(fila2); // (6)
 free(fila3); // (6)
 // se devuelve la memoria que cada arreglo pidió

 free(mat);  // (7)
 // se devuelve la memoria que el arreglo mat pidió. No olvidar que mat es un
 // arreglo cuyo cada elemento guarda la dirección de un apuntador.

 return 0;
}
```

(1) `mat` es el nombre con el que nos vamos a referir a la matriz en nuestro programa. Es una variable que guardará direcciones de vectores de enteros, por eso aparece el doble asterisco.
(2) Pide memoria para crear el arreglo vertical.
(3) Pide memoria para crear cada una de las filas.

Hasta este punto el arreglo vertical no tiene nada que ver con los arreglos horizontales. Entonces lo que sigue es ligar ambos arreglos, como lo muestra (4).

(4) En cada entrada del arreglo vertical guardamos la dirección de inicio de cada arreglo horizontal.

(5) Ya podemos utilizar a nuestra matriz como si la hubiéramos declarado en la manera tradicional.
(6) Debemos devolver la memoria utilizada por cada uno de los arreglos cuando ya no necesitemos más a nuestra matriz. Empezamos por devolver la memoria de cada arreglo horizontal.
(7) Devolvemos la memoria del arreglo vertical. ¡El órden de (6) y (7) sí importa!

3. Matriz con número variable de filas y columnas

El siguiente ejemplo es similar al que acabamos de ver, pero utiliza ciclos para pedir y devolver memoria para los arreglos horizontales. Este es el procedimiento normal para trabajar con matrices dinámicas.

```c
#include <stdio.h>
#include <stdlib.h>

void Print(int **m, int filas, int cols)
{
 int i, j;

 for (i = 0; i < filas; i++) {
  for (j = 0; j < cols; j++) {
   printf("%d", m[i][j]);
  }
  printf("\n");
 }
}

int main(int argc, const char *argv[])
{
 int **mat;
 // mat almacena la dirección de arreglos que guardan direcciones de enteros

 int filas = 3;
 int cols = 3;
 // límites para la matriz mat

 int i, j;
 // contadores de propósito general

 mat = (int **) malloc(filas * sizeof(int **)); // (1)
 // se reservó la memoria para mat, pero ésta aún no apunta a nada

 for (i = 0; i < filas; i++) { // (2)
```

```
  mat[i] = (int *) malloc(cols * sizeof(int));
 }
 // cada slot de mat almacena la dirección de cada fila. Cada fila tiene
'cols' slots

 // utilizamos a la matriz recién creada
 for (i = 0; i < filas; i++) {  // (3)
  for (j = 0; j < cols; j++) {
   mat[i][j] = 100 + 100 * j + 1000 * i;
   // esta operación es para obtener los mismos resultados que en el
   // ejemplo ej1.c. Lo importante es que estamos guardando valores enteros
  }
 }

 // imprimimos mat completa, otra vez utilizando notación de matrices
 for (i = 0; i < filas; i++) {
  for (j = 0; j < cols; j++) {
   printf("%d\t", mat[i][j]);
  }
  printf("\n");
 }


 Print(mat, filas, cols);

 for (i = 0; i < filas; i++) { // (4)
  free(mat[i]);
 }
 // se devuelve la memoria que cada arreglo pidió

 free(mat); // (5)
 // se devuelve la memoria que el arreglo mat pidió. No olvidar que mat es un
 // arreglo cuyo cada elemento guarda la dirección de un apuntador.

 return 0;
}
```

(1) `mat` es el nombre con el que nos vamos a referir a la matriz en nuestro programa. Es una variable que guardará direcciones de vectores de enteros, por eso aparece el doble asterisco.
(2) Pide memoria para crear el arreglo vertical.
(3) Pide memoria para crear cada una de las filas. Cada vuelta del ciclo se corresponde con cada uno de los arreglos horizontales; al mismo tiempo enlaza el arreglo horizontal con el arreglo vertical. (En cada

entrada del arreglo vertical guardamos la dirección de inicio de cada arreglo horizontal.)
(4) Ya podemos utilizar a nuestra matriz como si la hubiéramos declarado en la manera tradicional.
(5) Debemos devolver la memoria utilizada por cada uno de los arreglos cuando ya no necesitemos más a nuestra matriz. Empezamos por devolver la memoria de cada arreglo horizontal.
(6) Devolvemos la memoria del arreglo vertical. ¡El órden de (5) y (6) sí importa!

4. Matrices y ADTs (Avanzado)

El siguiente ejemplo muestra el uso de un ADT para matrices. Se deja al alumno o lector el estudio de éste.

```
/* Copyright (C)
 * 2014 - p.avanzada@yahoo.com.mx
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; either version 2
 * of the License, or (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA.
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

typedef struct
{
 int **matrix;
 ///< La matriz en sí misma

 int rows;
 ///< El número de filas de la matriz

 int cols;
```

```
 ///< El número de columnas de la matriz

} Matrix_Type;


/**
 * @brief Crea una nueva matriz
 *
 * @param cols El número de columnas de la nueva matriz
 * @param rows El número de filas de la nueva matriz
 *
 * @return Una referencia a la nueva matriz
 *
 * @pre: Ninguna
 * @post: Si la memoria alcanzó, entonces se creó la nueva matriz. Esta función
 * debe ser llamada antes que cualquier otra del ADT.
 */
Matrix_Type * Ctor(int cols, int rows)
{
 int i, j;

 Matrix_Type * this = NULL;

 this = (Matrix_Type *) malloc(sizeof(Matrix_Type));
 // crea al objeto matriz

 if (this == NULL) {
  return this;
 }

 this->rows = rows;
 this->cols = cols;
 // guarda los límites de la matriz

 this->matrix = (int **) malloc(rows * sizeof(int **));
 // crea el vector que guardará la dirección de inicio de cada una de las
 // filas

 for (i = 0; i < cols; i++) {
  for (j = 0; j < rows; j++) {
   this->matrix[i] = (int *) malloc(cols * sizeof(int));
   // crea cada una de las filas
  }
```

```
 }

 return this;
}

/**
 * @brief Destruye una matriz creada con Ctor
 *
 * @param this Referencia a la matriz que se va a destruir
 *
 * @pre: La matriz debe existir
 */
void Dtor(Matrix_Type * this)
{
 int i;

 for (i = 0; i < this->rows; i++) {
  free(this->matrix[i]);
  // devuelve la memoria que cada fila pidió
 }

 free(this->matrix);
 // devuelve la memoria que pidió

 free(this);
 // devuelve la memoria pedida por el objeto de trabajo
}

/**
 * @brief Imprime en la pantalla una matriz completa
 *
 * @param this La referencia a la matriz
 *
 * @pre: La matriz debe existir
 * @post: Ninguna
 */
void Print(Matrix_Type * this)
{
 int i, j;

 for (i = 0; i < this->rows; i++) {
  for (j = 0; j < this->cols; j++) {
   printf("%d\t", this->matrix[i][j]);
```

```
   }
   printf("\n");
  }
}

/**
 * @brief Escribe un nuevo valor en la celda especificada
 *
 * @param this La matriz
 * @param row La fila
 * @param col La columna
 * @param val El nuevo valor
 *
 * @pre: La matriz debe existir
 * @pre: La fila y columna no deben sobrepasar los límites de la matriz
 * @post: La celda [row][col] queda afectada con el nuevo valor
 */
void SetSingleVal(Matrix_Type * this, int row, int col, int val)
{
 this->matrix[row][col] = val;
}

/**
 * @brief Escribe una fila completa
 *
 * @param this La matriz
 * @param row El número de fila que se va a escribir
 * @param vector El arreglo que se va a copiar a la matriz en la fila 'row'
 *
 * @pre 'vector' debe tener el mismo número de columnas que la matriz
 */
void SetRow(Matrix_Type * this, int row, double vector[])
{
 int i = 0;

 assert(row < this->rows);

 for (i = 0; i < cols; i++) {
  this->matrix[row][i] = vector[i];
 }
}

/**
```

```
 * @brief Driver program
 *
 * @param argc NA
 * @param argv[] NA
 *
 * @return NA
 */
int main(int argc, const char *argv[])
{
 Matrix_Type * matriz = Ctor(3,3);
 assert(matriz);
 // si no se puede crear al objeto, entonces el programa se termina

 SetSingleVal(matriz, 0,0,100);
 SetSingleVal(matriz, 1,1,200);
 SetSingleVal(matriz, 2,2,300);
 // asignamos algunos valores

 Print(matriz);

 Dtor(matriz);
 // se devuelve toda la memoria asignada al objeto de trabajo

 return 0;
}
```