Taller de programación: C - Parte 3

Autor: Luis A. Muñoz - UPC- 2020

Los arreglos son objetos que permiten la organización de la información en memoria y sus manipulación es la entrada a los algoritmos útiles en C.

Arrays: una colección de datos

Los tipos de datos en lenguaje C permiten registrar tanto letras como caracteres en memoria. Pero estos siempre estan reducidos a valores *atómicos*, es decir valores puntuales. Sin embargo, cuando se trata de procesar datos lo que se tienen son secuencias o colecciones de números; por ejemplo, todos los pesos y alturas de una población para obtener el IMC de cada una de las personas. No tiene sentido tener diferentes variables para los diferentes datos de la población cuando, a fin de cuentas, todos los datos almacenados en la memoria. Lo que se necesita no es una etiqueta para cada posición de memoria (lo que en términos prácticos resulta ser una variable), sino una etiqueta para especificar un bloque completo de memoria que tenga una secuencia consecutiva de datos homogéneos.

Así, podemos definir un arreglo como una colección de datos, todos del mismo tipo, organizados en un bloque de memoria en donde los datos estan almacenados en orden consecutivo. Para su gestión eficiente necesitamos algunos criterios de diseño:

- Una colección de datos ocupa un bloque de memoria
- Para la manipulación de los datos se requiere conocer dos propiedades de este bloque: donde empieza y cuanto ocupa.

Como se reviso en un documento anterior, un `string` es un arreglo de `chars` que tenía un control de *fin de arreglo* con el caracter *\0*. En el caso de los arreglos que almacenan números (tanto `int` como `float`) no existe este control de fin de bloque, por lo que es necesario tener esta información a la mano.

Una array se define de la siguiente manera:

```
int enteros[] = {1, 2, 3, 4, 5};
float reales[] = {1.0, 2.0, 3.0, 4.0, 5.0};
```

En ambos casos, se tendran bloques de 4 bytes x 5 = 20 bytes en memoria. El compilador busca un bloque en donde pueda colocar los datos de forma consecutiva.

Considere el siguiente script:

In [ ]:
```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

int main()
{
    // Arrays en una dimension
    int numeros[]= {1, 2, 3, 4, 5};
    int *ptr;

    ptr = numeros;

    // La variable "array" contiene la direccion del primer elemento
    printf("numeros = %d\n\n", numeros);

    // La direcciones de los elementos son consecutivas
    for (int i=0; i<5; i++) {
        printf("numeros[%d] = %d (Direccion: %d)\n", i, numeros[i], &numeros[i
]);
    }

    // Los punteros saltan en direcciones por el tipo de dato
    printf("\n");
    for (int i=0; i<5; i++) {
        printf("numeros[dir:%d] = %d\n", ptr, *ptr);
        ptr++;
    }

    return 0;
}
```

```
numeros = 6487536

numeros[0] = 1 (Direccion: 6487536)
numeros[1] = 2 (Direccion: 6487540)
numeros[2] = 3 (Direccion: 6487544)
numeros[3] = 4 (Direccion: 6487548)
numeros[4] = 5 (Direccion: 6487552)

numeros[dir:6487536] = 1
numeros[dir:6487540] = 2
numeros[dir:6487544] = 3
numeros[dir:6487548] = 4
numeros[dir:6487552] = 5
```

Observe que el arreglo `numeros` almacena una colección de 5 valores enteros, pero cuando se imprime el valor de la etiqueta `numeros` (es decir, el nombre del arreglo) lo que se muestra es un número que no esta en el secuencia. Esto es importante porque es la noción capital para entender los arreglos y como manipularlos. Vea que hay tres valores que devuelven el mismo dato:

```
printf("numeros = %d\n", numeros);
printf("numeros[%d] = %d (Direccion: %d)\n", 0, numeros[0], &numeros[0]);
```

Es decir, `numeros` y `&numeros` tienen el mismo valor, lo que es equivalente a decir que cuando se define un arreglo se esta creando un puntero (recuerde, un puntero es un valor que almacena una dirección de memoria) que almacena la dirección de memoria del primer valor en el bloque).

Por eso se puede escribir la operación:

```
ptr = numeros;
```

En lugar de:

```
ptr = &numeros;
```

Ya que `numeros` ya tiene la dirección. Entonces *tiene que entenderse que el operador & se encuentra implicito al momento de llamar a un arreglo*. Esto significa que cuando se pasa un arreglo a una función se pasa siempre *por referencia*.

Este código también indica algo importante:

```
for (int i=0; i<5; i++) {
    printf("numeros[dir:%d] = %d\n", ptr, *ptr);
    ptr++;
}
```

Esto muestra la dirección de memoria y cada uno de los valores en el bloque. Esto por la manipulación de punteros. Al incrementar el valor del puntero se obtienen los siguientes resultados:

```
numeros[dir:6487536] = 1
numeros[dir:6487540] = 2
numeros[dir:6487544] = 3
numeros[dir:6487548] = 4
numeros[dir:6487552] = 5
```

Lo primero que debe de notar es que el primer datos esta en 6487536 y el siguiente en 6487540, es decir 4 dirección más adelante y no una (lo que podría esperarse después de incrementar el puntero co la operación `ptr++` . Esto es porque al definir el puntero se especifico:

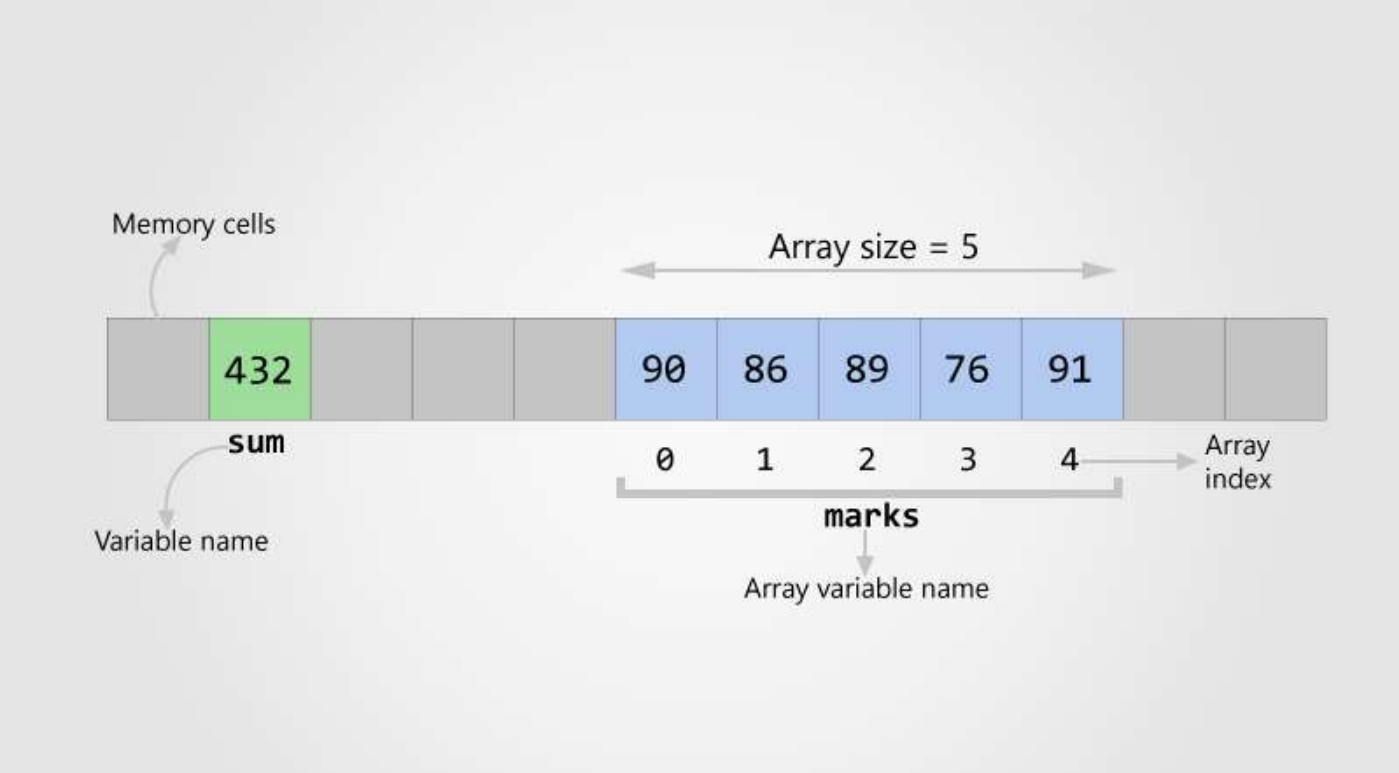
```
int *ptr;
```

Al hacer esto, se le indica que hará seguimiento de objetos de 4 bytes, por lo que cuando se incremente su valor, *el siguiente valor* esta en una dirección 4 posiciones más adelante. Por eso los punteros deben inicializarse con el tipo de dato al que van a apuntar.

Pero hay una forma más inmediata de acceder a los elementos de un arreglo y es por medio de un índice. Estas operaciones son equivalentes:

```
numeros[0] -> *ptr
numeros[1] -> *(ptr+1)
numeros[2] -> *(ptr+2)
```

Es decir, el índice de un arreglo no es más que el *offset* o desviación en la posición de un dato respecto a la posición del primer valor del bloque.



Así que al definir un arreglo ya tenemos un puntero con la dirección del primer elemento. Al momento de definir un arreglo también sabemos de antemano su tamaño. Sin embargo, este dato debemos de compartirlo con todas las secciones del código que vayan a manipular un arreglo. El número de elementos de un arreglo se puede calcular con la operación:

```
sizeof(array) / sizeof(array_type)
```

Esto divide todo el espacio de memoria ocupado por un arreglo por el tamaño de cada celda de datos por el tipo almacenado. Sin embargo, la mejor forma de tener el dato del numero de elementos de un arreglo de forma global es definir este dato como un *macro* o una etiqeta global:

In [ ]:
```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#define NDATA 5

int main()
{
    // Arrays en una dimension
    int numeros[NDATA]= {1, 2, 3, 4, 5};

    // Numero de elementos: sizeof(array) / sizeof(array_type)
    for (int i=0; i<NDATA; i++) {
        printf("numeros[%d] = %d (Direccion: %d)\n", i, numeros[i], &numeros[i
]);
    }
}
```

```
numeros[0] = 1 (Direccion: 6487552)
numeros[1] = 2 (Direccion: 6487556)
numeros[2] = 3 (Direccion: 6487560)
numeros[3] = 4 (Direccion: 6487564)
numeros[4] = 5 (Direccion: 6487568)
```

Como se observa, la gestión de los índices en un lazo `for` permite tener acceso a cada uno de los elementos de un arreglo en los límites del bloque de memoria. Esto es importante ya que puede generse un error por desbordamiento, es decir salirse del bloque de datos.

Arreglos como parametros de función

Pasar un arreglo como parametro de función forzosamente significa que se pasará por referencia. La definición de este procedimiento en C asume esto por lo que una nomenclatura ya no utiliza la especificación de un puntero.

In [6]:
```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#define NDATA 10

void array_al_cuadrado(int[]);

int main()
{
    int numeros[NDATA];

    for (int i=0; i<NDATA; i++) {
        numeros[i] = 1 + rand() % 25;
    }

    for (int i=0; i<NDATA; i++) {
        printf("%d ", numeros[i]);
    }

    // Funcion que eleva todos los elementos de un array al cuadrado
    array_al_cuadrado(numeros);

    printf("\n");
    for (int i=0; i<NDATA; i++) {
        printf("%d ", numeros[i]);
    }

    printf("\n");

    return 0;
}

void array_al_cuadrado(int arr[]) {
    for (int i=0; i<NDATA; i++) {
        arr[i] = arr[i] * arr[i];
    }
}
```

```
9 12 3 16 19 11 12 18 25 22
81 144 9 256 361 121 144 324 625 484
```

En el script anterior de prototipa la función de la forma:

```
void array_al_cuadrado(int[]);
```

El `[]` significa que se pasará un arreglo, pero realmente es un puntero con la dirección de memoria del primer elemento del arreglo.

```
void array_al_cuadrado(int arr[]) {
    for (int i=0; i<NDATA; i++) {
        arr[i] = arr[i] * arr[i];
    }
}
```

Se observa que la función no retorna ningún valor, ya que su objetivo es elevar al cuadrado los elementos del arreglo y lo qye hace es afectar los valores que estan en el bloque de memoria. Para esto necesita información sobre donde está la información para alterarla enla memoria. Entonces, como es un arreglo, el lugar de utilizar la operación `*arr + i` para encontrar el dato a modificar, se utiliza la nomenclatura basada en índices `arr[i]` para leer los datos del arreglo y guardar las modificaciones en la misma posición de memoria:

```
arr[i] = arr[i] * arr[i];
```

Los límites del lazo `for` son marcados por el macro `NDATA` disponible a lo largo del script. En caso no sea asi, se le deberá pasar a la función un parámetro adicional con la indicación del número de elementos del arreglo, con lo que el prototipo de la función tendría que considerar esta información:

```
void array_al_cuadrado(int[], int);
```

y al llamar a la función se incluiría este dato:

```
array_al_cuadrado(numeros, NDATA);
```

Ejemplo de aplicación: lanzamiento de datos

Apliquemos los arreglos a resolver algunos problemas. Por ejemplo, a analizar la frecuencia en la ocurrencia del lanzamiento de un dado. Para esto debemos definir un arreglo que contenga la simulación del lanzamiento de un número de dados. Para esto tendremos dos funciones:

```
void genera_dados(int[], int);
int freq_array(int[], int ,int);
```

La función `genera_dados()` retornará un arreglo de tamaño `n` de valores entre 1 y 6. Por otro lado, la función `freq_array()` retornará cual es la frecuencia de ocurrencia de un número `n` en un arreglo. Separemos los códigos para entender el desarollo.

Revisemos las funciones:

In [ ]:
```
void genera_dados(int arr[], int size) {

    for (int i=0; i<size; i++) {
        arr[i] = 1 + rand() % 6;
    }
}
```

La función `genera_dados()` toma como parametros el arreglo (formalmente, la dirección del primer elemento) y el tamaño de este arreglo. Con esta información, carga en la direcciones de memoria los números aletorios en el rango 1 a 6.

In [ ]:
```
int freq_array(int arr[], int size, int n) {
    int cont = 0;

    for (int i=0; i<size; i++) {
        if (arr[i] == n) {
            cont++;
        }
    }
    return cont;
}
```

La función `freq_array` requiere más parametros de entrada: el arreglo, el tamaño de este y el número del que se quiere contar cuantas veces esta presente en el arreglo. Se inicializa un contador en 0 y se barren todas las direcciones de memoria y se va contando cuantas direcciones tienen almacenado el valor `n` para retornar dicha cuenta.

Estas funciones se prototipan y se define `main()` :

In [ ]:
```c
#include <stdio.h>
#include <stdlib.h>

void genera_dados(int[], int);
int freq_array(int[], int, int);

int main()
{
    int n_veces;
    float freq;

    printf("Cuantas veces desea lanzar un dado: "); scanf("%d", &n_veces);

    // Generamos un arreglo con 100 valores del lanzamento de un dado
    int dados[n_veces];
    genera_dados(dados, n_veces);

    // Se muestra la probabilidad de ocurrencia
    // de cada resultado
    for (int i=1; i<7; i++) {
        freq = (float)freq_array(dados, n_veces, i)/n_veces;
        printf("freq(%d): %.4f\n", i, freq);
    }

    return 0;
}
```

La función `main()` define la variable `n_veces` que se cargará con el numero de lanzamientos requeridos. Esto es equivalente al tamaño del arreglo. Asi que se utiliza el valor de esta variable para llamar a la función `genera_dados()` para que retorne el arreglo `dados` (definido previemante con el tamaño `n_veces`), para contar en el rango de 1 a 7 cual es la frecuencia de ocurencia de cada resultado. Para hallar la probailidad se debe dividir entre el número de lanzamientos, por loq que hay que convertir el resultado de la frecuencia en un valor tipo `float` .

Agregemos un indicador gráfico de la densidad de probabilidad. Vamos a definir una función `put_starts()` que imprima un número de asteriscos como sucede con `puts()` que agrega un string y un '\n' al final:

In [ ]:
```c
void put_stars(int n) {
    for (int i=0; i<n; i++) {
        printf("*");
    }
    printf("\n");
}
```

Ahora, nuestra función `main()` queda de la siguiente forma:

In [ ]:
```c
#include <stdio.h>
#include <stdlib.h>

void genera_dados(int[], int);
int freq_array(int[], int, int);
void put_stars(int);

int main()
{
    int n_veces;
    float freq;

    printf("Cuantas veces desea lanzar un dado: "); scanf("%d", &n_veces);

    // Generamos un arreglo con 100 valores del lanzamento de un dado
    int dados[n_veces];
    genera_dados(dados, n_veces);

    // Se muestra la probabilidad de ocurrencia
    // de cada resultado
    for (int i=1; i<7; i++) {
        freq = (float)freq_array(dados, n_veces, i)/n_veces;
        printf("freq(%d)= %.4f : ", i, freq);
        put_stars((int)(freq*100*n_veces));
    }

    return 0;
}
```

Generemos 100 lanzamietos:

```
Cuantas veces desea lanzar un dado: 100
freq(1)= 0.1400 : **************
freq(2)= 0.0900 : *********
freq(3)= 0.1800 : ******************
freq(4)= 0.2000 : ********************
freq(5)= 0.2000 : ********************
freq(6)= 0.1900 : *******************
```

Puede ver como la distribución de probabilidad de detalla de forma numérica y como una barras horizontales. En un dado justo, la probabilidad de ocurrencia de cada lado es igual, asi que si lanzamos un gran número de veces, todas las probabilidades deben ser iguales (1/6 = 0.16666):

```
Cuantas veces desea lanzar un dado: 100000
freq(1)= 0.1663 : ****************
freq(2)= 0.1672 : ****************
freq(3)= 0.1646 : ****************
freq(4)= 0.1675 : ****************
freq(5)= 0.1677 : ****************
freq(6)= 0.1667 : ****************
```

Arreglos en una nueva dimension

Asi como los datos se pueden organizar en memoria como un bloque, se puede definir un *bloque de bloques*, donde cada bloque será una parte del bloque principal. Así como los datos en un arreglo son homogeneos, los bloques en una colección de bloques tambiénlo serán, es decir que tendrán los mismo tipos de datos y cada bloque tendrá el mismo tamaño.

Esto es equivalente a guardar una colección de arreglos bajo un arreglo más grande. Se puede conceptualiar esto como una *tabla de datos*, donde los bloques internos son las filas de la tabla. Observe la siguiente figura:

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Cada uno de los datos tiene dos índices, uno para fila y otro para columna. Recuerde que cada uno de estas posiciones son *offsets* de la posicion inicial. Asi que:

```
a[0][0] -> *a
a[0][1] -> *(a+1)
a[0][2] -> *(a+2)
a[0][3] -> *(a+3)

a[1][0] -> *(a+4)
a[1][1] -> *(a+5)
a[1][2] -> *(a+6)
a[1][3] -> *(a+7)

.
.
```

El compilador necesita la información del número de elementos del arreglo y el número de bloques internos, lo que es equivalente a especificar el número de filas y columnas:

```
int tabla[num_elementos][num_bloques] -> int tabla[n_filas][n_columnas]
```

Para la tabla anterior la definición sería:

```
int tabla[3][4] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
```

o de forma más expliicita:

```
int tabla[3][4] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}, {10, 11, 12}};
```

Otra forma más reducida de indicar lo mismo sería:

```
int tabla[][4] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}, {10, 11, 12}}
```

¿Por que se puede prescindir del primer índice? Porque es el número de elementos del arreglo que se calcula de los datos a asignar (como sucede en el caso de los arreglos de una dimensión). La explicación de que "el primer indice es el número de filas y el sigueinte en número de columnas" no es correcto ya que esta es una aproximación a que el primer índice es el número de elementos y el segundo el número de bloques en los que están organizados.

Una buena práctica de programación es organizar los datos al momento de definir un arreglo como una tabla:

```
int tabla[3][4] = {{1, 2, 3, 4},
                   {5, 6, 7, 8},
                   {9, 10, 11, 12}};
```

C no toma en consideración los espacios en blanco ni los saltos de línea en el código asi que esto es equivalente a las definiciónes anteriores, pero se gana claridad en el código.

In [21]:
```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#define FILA 3
#define COL 4

int main()
{
    // Arrays en dos dimensiones
    int numeros[FILA][COL]= {{1, 2, 3, 4},
                             {5, 6, 7, 8},
                             {9, 10, 11, 12}};


    // 1ra fila
    printf("%3d", numeros[0][0]);
    printf("%3d", numeros[0][1]);
    printf("%3d", numeros[0][2]);
    printf("%3d", numeros[0][3]);
    printf("\n");

    // 2da fila
    printf("%3d", numeros[1][0]);
    printf("%3d", numeros[1][1]);
    printf("%3d", numeros[1][2]);
    printf("%3d", numeros[1][3]);
    printf("\n");

    // 3ra fila
    printf("%3d", numeros[2][0]);
    printf("%3d", numeros[2][1]);
    printf("%3d", numeros[2][2]);
    printf("%3d", numeros[2][3]);
    printf("\n");

    return 0;
}
```

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

Observe que los diferentes `printf` van imprimiendo cada uno de los elementos especificado todas las combinaciones de índices. Todo esto es un proceso repetivo que se puede reducir con un lazo `for` :

In [22]:
```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#define FILA 3
#define COL 4

int main()
{
    // Arrays en dos dimensiones
    int numeros[FILA][COL]= {{1, 2, 3, 4},
                             {5, 6, 7, 8},
                             {9, 10, 11, 12}};


    for (int j=0; j<COL; j++) {
        printf("%3d", numeros[0][j]);
    }
    printf("\n");

    for (int j=0; j<COL; j++) {
        printf("%3d", numeros[1][j]);
    }
    printf("\n");

    for (int j=0; j<COL; j++) {
        printf("%3d", numeros[2][j]);
    }
    printf("\n");

    return 0;
}
```

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

En el código anterior cada `for` extrae los valores de cada columna por fila. Esto sigue siendo repetitivo, así que podemos incluir un `for` externo para que vaya barriendo cada fila:

In [23]:
```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#define FILA 3
#define COL 4

int main()
{
    // Arrays en dos dimensiones
    int numeros[FILA][COL]= {{1, 2, 3, 4},
                             {5, 6, 7, 8},
                             {9, 10, 11, 12}};


    for (int i=0; i<FILA; i++) {
        for (int j=0; j<COL; j++) {
            printf("%3d", numeros[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```

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

Un lazo `for` anidado nos permite barrer todos los elementos de un arreglo bidimensonal en una secuencia de barrido. Esta construcción es muy común en un código C cuando se manipulan arreglos.

Y antes de culminar con los arreglos de este tipo: ¿se pueden tener arreglos de más de 2 dimensiones? Si. Por ejemplo, una colección de 100 cartones de bingo (con 5 columnas y 5 filas) tiene la forma:

```c
int bingo[100][5][5];
```

Y podriamos generar todos los cartones con números aleatorios entre 1 y 100 de la forma:

```c
for (int h=0; h<5; h++) {
    for (int i=0; i<5; i++) {
        for (int j=0; j<5; j++) {
            bingo[h][i][j] = 1 + rand() % 101;
        }
    }
}
```

Pero no se asuste. No vamos a entrar a esos terrenos.

Ejemplo de aplicación: lanzamiento de *n* dados

Ahora que tenemos una estructura bidimensional, podemos lanzar *n* dados. Si lanzamos 10 veces un dados teníamos lo siguiente:

```
dados[10];
```

Ahora, si lanzamos 10 veces dos dados, tendremos:

```
dados[10][2];
```

Así que, en términos generales, lanzar n_dados un número n_veces será equivalente a codificar:

```
dados[n_veces][n_dados];
```

Solo que, para obtener los resultados de la propabilidad de ocurencia de los resultados, deberemos sumar las caras de los dados. Es decir, si el lanzamiento 3 del dado tiene los resultados (el 3er lanzamiento esta en el índice 2 ya que inicia la cuenta con 0):

```
dados[2][] = {3, 5}
```

Esa jugada será equivalente a haber lanzado 8, o sea:

```
resultado = dados[2][0] + dados[2][1];
```

Habra que adaptar las funciones. En particular, debemos indicarle ya no solo el número de elementos, sino el número de bloques internos, o lo que es lo mismo, el número de filas y columnas de arreglo. Estas especificaciones son más complejas que en el casode una sola dimension ya que es necesario incluír en el prototipo las dimensiones del arrego (por lo menos, el tamaño de los bloques). Así que lo mejor será definir las variables `n_veces` y `n_dados` como globales. Por otro lado, hay que invertír el orden de los parametros, ya que el número de filas y columnas debe de ir antes del arreglo. Para el caso de `genera_dados()` :

In [ ]:
```
void genera_dados(int fil, int col, int arr[fil][col]) {
    for (int i=0; i<fil; i++) {
        for (int j=0; j<col; j++) {
            arr[i][j] = 1 + rand() % 6;
        }
    }
}
```

Para el caso de `freq_array()` hay que sumar los resultados (suma por filas) y contar estos resultados:

In [ ]:
```c
int freq_array(int fil, int col, int arr[fil][col], int n) {
    int resultado, cont = 0;

    for (int i=0; i<fil; i++) {
        resultado = 0;
        for (int j=0; j<col; j++) {
            resultado += arr[i][j];
        }

        if (resultado == n) cont++;
    }
    return cont;
}
```

Ahora,. nuestra función `main()` contempla un arreglo en dos dimensiones:

In [ ]:
```c
#include <stdio.h>
#include <stdlib.h>

int n_veces, n_dados;

void genera_dados(int, int, int[][n_dados]);
int freq_array(int, int, int[][n_dados], int);
void put_stars(int);

int main()
{
    float freq;

    printf("Cuantas veces desea lanzar un dado: "); scanf("%d", &n_veces);
    printf("Cuantos dados quiere lanzar: "); scanf("%d", &n_dados);

    // Generamos un arreglo con el lanzamiento de los dados
    int dados[n_veces][n_dados];
    genera_dados(n_veces, n_dados, dados);

    // Se muestra la probabilidad de ocurrencia
    // de cada resultado
    for (int i=n_dados; i<=(6*n_dados); i++) {
        freq = (float)freq_array(n_veces, n_dados, dados, i)/n_veces;
        printf("freq(%2d)= %.4f : ", i, freq);
        put_stars((int)(freq*100*n_dados));
    }

    return 0;
}
```

La función `put_stars()` no cambia. Ahora veamos los resultaodos de lanzar dos dados 100,000 veces:

```
Cuantas veces desea lanzar un dado: 100000
Cuantos dados quiere lanzar: 2
freq( 2)= 0.0274 : **
freq( 3)= 0.0550 : *****
freq( 4)= 0.0829 : ********
freq( 5)= 0.1118 : ***********
freq( 6)= 0.1401 : **************
freq( 7)= 0.1679 : ****************
freq( 8)= 0.1379 : *************
freq( 9)= 0.1101 : ***********
freq(10)= 0.0838 : ********
freq(11)= 0.0556 : *****
freq(12)= 0.0276 : **
```

El numero más probable al lanzar dos dados será el número 7 (el "lucky number") y la distribución del lanzamiento de dos dados tiene la forma de una distribución normal. Esto es más notable para el lanzamiento de un mayor número de dados:

```
Cuantas veces desea lanzar un dado: 10000
Cuantos dados quiere lanzar: 12
freq(12)= 0.0000 :
freq(13)= 0.0000 :
freq(14)= 0.0000 :
freq(15)= 0.0000 :
freq(16)= 0.0000 :
freq(17)= 0.0000 :
freq(18)= 0.0000 :
freq(19)= 0.0000 :
freq(20)= 0.0002 :
freq(21)= 0.0001 :
freq(22)= 0.0002 :
freq(23)= 0.0003 :
freq(24)= 0.0005 :
freq(25)= 0.0012 : *
freq(26)= 0.0020 : **
freq(27)= 0.0027 : **
freq(28)= 0.0036 : ***
freq(29)= 0.0043 : ****
freq(30)= 0.0080 : ********
freq(31)= 0.0123 : ************
freq(32)= 0.0170 : *****************
freq(33)= 0.0193 : *******************
freq(34)= 0.0281 : ****************************
freq(35)= 0.0321 : ********************************
freq(36)= 0.0418 : ****************************************
freq(37)= 0.0484 : ************************************************
freq(38)= 0.0507 : *************************************************
freq(39)= 0.0618 : ************************************************************
freq(40)= 0.0613 : ************************************************************
freq(41)= 0.0682 : ********************************************************************
****
freq(42)= 0.0675 : ********************************************************************
***
freq(43)= 0.0719 : ********************************************************************
*******
freq(44)= 0.0620 : ************************************************************
freq(45)= 0.0587 : ********************************************************
freq(46)= 0.0528 : ****************************************************
freq(47)= 0.0486 : ********************************************
freq(48)= 0.0404 : ****************************************
freq(49)= 0.0308 : ******************************
freq(50)= 0.0255 : *************************
freq(51)= 0.0197 : ********************
freq(52)= 0.0183 : ******************
freq(53)= 0.0149 : **************
```

```
freq(54)= 0.0078 : *******
freq(55)= 0.0061 : ******
freq(56)= 0.0050 : *****
freq(57)= 0.0028 : **
freq(58)= 0.0011 : *
freq(59)= 0.0007 :
freq(60)= 0.0008 :
freq(61)= 0.0002 :
freq(62)= 0.0000 :
freq(63)= 0.0001 :
freq(64)= 0.0001 :
freq(65)= 0.0001 :
freq(66)= 0.0000 :
freq(67)= 0.0000 :
freq(68)= 0.0000 :
freq(69)= 0.0000 :
freq(70)= 0.0000 :
freq(71)= 0.0000 :
freq(72)= 0.0000 :
```

Proyecto: Adivina un número

Vamos a hacerun script que le adivine un número al usuario. Esto esta basado en el truco de cartas de adivinar una carta de un mazo de 21. El detalle de este truco de muestra aqui: (https://www.youtube.com/watch?v=KpF2KsH7yWM)

El proceso del truco lo vamos a simular en la manipulación de arreglos. El truco tiene la siguiente mecánica:

Se repite tres veces:

- Distribuir los números en tres columnas, fila a fila
- El usuario debe de indicar en que columna esta el numero seleccionado
- Volver a recolectar los numeros por columnas, donde la columna indicada este en el medio

Luego de 3 repeticiones, por la forma de recolectar los números de forma iterativa (en donde la columna seleccionada va al medio) fuerza a colocar el número seleccionado hacia el medio de la colección de números y en una secuencia de 21 valores, esta es la posición 11. Como vamos a tener un arreglo de valores que inicia la cuenta en 0, el valor buscado estará en la posición 10.

Entonces: generamos dos arreglos, uno de una dimension (el mazo de cartas) y otra de dos dimensiones (las cartas puestas sobre la mesa). En nuestro caso serán números aleatorios entre 100 y 999. El arreglo `numeros` es cargado los valores aleatorios que luego son repartidos en el arreglo bidimensonal `tablero`. La función `recoje_numeros` toma como parametros de entrada el orden en el que se recojeran los números por orden de columnas para volver a formar el arreglo `numeros` y repetir todo este proceso tres veces. Al finalizar se le muestra al usuario el elemento de la posición 11. Ese será el número seleccionado. ¡Nunca falla!

In [ ]:
```c
#include <stdio.h>
#include <stdlib.h>

#define ROW 7
#define COL 3

int numeros[ROW * COL];
int tablero[ROW][COL];

void show_tablero();
void recoje_numeros(int, int, int);

int main()
{
    int idx, opc;

    // Se generan numeros aleatorios para el tablero
    for(int i=0; i<(ROW * COL); i++) {
        numeros[i] = 100 + rand() % (999 - 100 + 1);
    }

    // Lazo de control (3 iteraciones)
    for (int k=0; k<3; k++) {
        // Se reparten los numeros en el tablero
        idx = 0;
        for (int i=0; i<ROW; i++) {
            for (int j=0; j<COL; j++) {
                tablero[i][j] = numeros[idx];
                idx++;
            }
        }

        // Se muestra el tablero
        system("cls");
        show_tablero();

        // El usuario ingresa la columna donde esta el numero escogido
        if (k==0) printf("\nEscoja un numero...\nEn que columna esta?: ");
        if (k==1) printf("\nAhora en que columna esta?: ");
        if (k==2) printf("\nY ahora en que columna esta?: ");
        if (k==3) printf("\nY Ahora...: ");

        scanf("%d", &opc);

        // Se recogen los de arr a aba y de izq a der de forma tal que
        // se recoja en el medio a la columna escogida
        if (opc == 1) recoje_numeros(3, 1, 2);
        if (opc == 2) recoje_numeros(1, 2, 3);
        if (opc == 3) recoje_numeros(2, 3, 1);
    }

    // Se reparten los numeros en el tablero
    system("cls");
    printf("\a\n\tEl numero escogido es %d!\n", numeros[10]);

    return 0;
```

```
}

void show_tablero() {
    printf("\t\tPODER MENTAL\n");
    printf("\t\t============\n");
    for (int i=0; i<ROW; i++) {
        printf("\t\t");
        for (int j=0; j<COL; j++) {
            printf("%3d ", tablero[i][j]);
        }
        printf("\n");
    }
    printf("\n");
}

void recoje_numeros(int c1, int c2, int c3) {
    int idx = 0;
    for (int i=0; i<ROW; i++) {
        numeros[idx] = tablero[i][c1-1];
        idx++;
    }

    for (int i=0; i<ROW; i++) {
        numeros[idx] = tablero[i][c2-1];
        idx++;
    }

    for (int i=0; i<ROW; i++) {
        numeros[idx] = tablero[i][c3-1];
        idx++;
    }
}
```