

# Programación de Sistemas (PrSist)

Departamento de Informática de Sistemas y Computadoras (DISCA)

*Universitat Politècnica de València*

## Práctica 2 Compilando Programas en C

### Contenido

1	Objetivos .....	2
1.1	Entorno necesario .....	2
2	Introducción al GCC.....	2
2.1	El compilador GCC.....	2
2.2	Sintaxis y opciones de GCC .....	2
2.3	Fases de compilación .....	3
3	Ejercicio 1: Fases de compilación y estructuración en módulos .....	4
3.1	Fases de compilación .....	4
3.2	Estructuración en módulos .....	5
3.3	Cuestiones.....	5
4	Ejercicio 2: Ámbito de las variables y gestión de errores. ....	6
4.1	Ámbito de las variables .....	6
4.2	Cuestiones.....	7
5	Ejercicio 3: Parámetros por línea de órdenes .....	7
5.1	Programa listar argumentos .....	8
5.2	Programa opciones .....	8
6	Ejercicio 4: Punteros y estructuras.....	9
6.1	Programa mayúsculas .....	9
6.2	Programa sumafilas .....	9
7	Ejercicio 5: Estructura de datos buffer circular .....	10
7.1	Programa con vector circular .....	12
7.2	Programa definición buffer circular .....	12
7.3	Programa rellenar datos en el buffer circular .....	12
7.4	Funciones para completar operaciones en el buffer circular .....	13

# 1 Objetivos

El objetivo de esta práctica es aprender a programar en C usando las herramientas de programación UNIX.

En concreto los objetivos son:

- Introducir el entorno de programación C en UNIX (shell, editor, gcc, etc.)
- Conocer las fases de compilación de un programa en C.
- Programar en C aprendiendo aquellos aspectos relevantes (ámbito de las variables, cadenas, punteros, gestión de memoria).

Para ello se van a realizar y/o modificar programas sencillos en C donde alumno tendrá que comprobar los resultados obtenidos y responder a pequeñas cuestiones sobre los mismos.

## 1.1 Entorno necesario

El entorno para realizar las prácticas consiste en un sistema Unix (Linux) y el compilador gcc (GNU Compiler). Como entorno de edición se puede utilizar KEdit, KWrite o cualquier otro editor.

Para editar archivos se puede usar cualquier editor de los disponibles, en el laboratorio recomendamos utilizar kate, que se puede iniciar desde el menú o de la línea de órdenes escribiendo para editar circulo.c:

```
$ kate circulo.c &
```

# 2 Introducción al GCC

## 2.1 El compilador GCC

GCC es un conjunto de compiladores desarrollados dentro del proyecto GNU y por lo tanto es software libre. En un principio sólo disponía de un compilador de C, y se denominaba GNU C Compiler. Actualmente incluye compiladores para diversos lenguajes como C, C++, Objective C, Fortran, Ada. Por ello, las siglas GCC ahora significan "GNU Compiler Collection". GCC es capaz de recibir un programa fuente en cualquiera de estos lenguajes y generar un programa ejecutable binario en el lenguaje de la máquina donde ha de ejecutarse. GCC puede, por tanto, generar código para distintas arquitecturas como Intel x86, ARM, Alpha, Power PC y muchas más. Es por esto, que es utilizado como compilador de desarrollo en la mayoría de plataformas. Por ejemplo, Linux, Mac OS X, iOS (iPhone e iPad) están íntegramente compilados con GCC.

## 2.2 Sintaxis y opciones de GCC

La sintaxis de uso del compilador gcc es la siguiente:

```
$ gcc [ opción | archivo ] ...
```

Las opciones van precedidas de un guion, como es habitual en UNIX, cada opción puede constar de varias letras y no pueden agruparse varias opciones tras un mismo guion. Algunas opciones requieren después un nombre de archivo o directorio, otras no. Finalmente, pueden darse varios nombres de archivo a incluir en el proceso de compilación. Un ejemplo de uso sería:

```
$ gcc -o hola hola.c
```

que genera el programa hola compilando el fuente hola.c

GCC tiene infinidad de opciones, a continuación detallamos las que vamos a utilizar a lo largo de las prácticas de fSO.

Opción	Descripción
-c	realiza el preprocesamiento, compilación y ensamblado, obteniendo el archivo en código objeto (.o).
-S	realiza el preprocesamiento y compilación, obteniendo el archivo en ensamblador.
-E	realiza solamente el preprocesamiento, enviando el resultado a la salida estándar
-o archivo	indica el nombre del archivo de salida (el ejecutable).
-Iruta	especifica la ruta hacia el directorio donde se encuentran los archivos marcados para incluir en el programa fuente. No lleva espacio entre la I y la ruta, así: -I/usr/include. Por defecto no es necesario indicar las rutas de las include estándar.
-Lruta	especifica la ruta hacia el directorio donde se encuentran los archivos de biblioteca con el código objeto de las funciones referenciadas en el programa fuente. No lleva espacio entre la L y la ruta, así: -L/usr/lib. Por defecto no es necesario indicar las rutas de las librería estándar.
-lNOMBRE	NOMBRE: librería a utilizar para enlazar junto con nuestro programa. Le dice al compilador qué biblioteca o librerías tiene que incluir con el programa que hemos desarrollado. Se utiliza cuando queremos modificar el conjunto de librerías que el enlazador utiliza por defecto. Por ejemplo: -lm incluiría la librería libm.so (librería matemática)
-v	Verbose on; muestra los comandos ejecutado en cada etapa de compilación y la versión del mismo.

## 2.3 Fases de compilación

El gcc, al igual que otros compiladores, se pueden distinguir 4 etapas en el proceso de compilación:

- **Preprocesado:** En esta etapa se interpretan las directivas al preprocesador. Entre otras cosas, las variables inicializadas con #define son sustituidas en el código por su valor en todos los lugares donde aparece su nombre y se incluye el fuente de los #include.
- **Compilación:** transforma el código C en el lenguaje ensamblador propio del procesador de nuestra máquina. Normalmente la máquina destino será la misma que la de desarrollo, pero en el caso de que la máquina difiera, a esto se denomina compilación cruzado.
- **Ensamblado:** transforma el programa escrito en lenguaje ensamblador a código objeto, un archivo binario en lenguaje de máquina ejecutable por el procesador.
- **Enlazado:** Las funciones de C/C++ incluidas en nuestro código, tal como printf(), se encuentran ya compiladas y ensambladas en bibliotecas existentes en el sistema. Es preciso incorporar de algún modo el código binario de estas funciones a nuestro ejecutable. En esto consiste la etapa de enlace, donde se reúnen uno o más módulos en código objeto con el código existente en las biblioteca

### 3 Ejercicio 1: Fases de compilación y estructuración en módulos

#### 3.1 Fases de compilación

En este ejercicio vamos a probar como compilar un programa en C y las distintas fases de compilación. El programa a compilar "circulo.c" es muy simple y se muestra a continuación:

En primer lugar introducir este código C en el fichero fuente correspondiente mediante un editor.

```
#include <stdio.h>

#define PI 3.1416
main() {
    float area, radio;
    radio = 10;
    area = PI * (radio * radio);
    printf("Area del circulo de radio %f es %f\n", radio, area);
}
```

Figura-1: Código del programa "circulo.c"

Para compilar este programa y generar directamente el ejecutable, realizando todas las fases de compilación, introducir la siguiente orden:

```
$ gcc -o circulo circulo.c
```

Comprobar que se ha generado un fichero ejecutable (con `ls -la`). Para poder probar el programa simplemente escribimos la orden (nota: es necesario poner `./` al nombre del fichero para que encuentre el ejecutable en el directorio actual y no lo busque en el PATH).

```
$ ./circulo
```

Está será la forma normal de trabajar, generar un fichero ejecutable a partir de un fichero con código fuente. Pero vamos a ver los resultados intermedios de cada fase de compilación.

a) PREPOCESAR: Podemos preprocesar el fichero con la opción `-E`. Hay que redirigir la salida a un fichero, que llamaremos `circulo.pp`.

```
$ gcc -E circulo.c > circulo.pp
```

Ver el contenido del fichero `circulo.pp` (con nuestro editor de textos) y podemos comprobar dos cosas, a) que ha desaparecido el `#include` y ha sido sustituido por el contenido de este fichero que contiene una serie de definiciones y prototipos de funciones para entrada y salida (entre ellas la función `printf` que vamos a utilizar en nuestro programa) y b) al final del fichero podréis comprobar que la variable `PI` ha sido sustituida por su valor, 3.1416, tal como había sido fijado en la sentencia `#define`.

b) COMPILAR: Podemos compilar (y preprocesar) el fichero con la opción `-S` lo que nos genera el código ensamblador del programa compilado.

```
$ gcc -S circulo.c
```

podemos comprobar que nos ha generado un fichero con extensión `.s` que contiene el código en ensamblador. Abrir el fichero y ver que efectivamente es así.

c) ENSAMBLAR: al ensamblar se transforma el código en ensamblado en un archivo binario de código máquina que es ejecutable por el procesador:

```
$ gcc -c circulo.c
```

Puede comprobarse el tipo de fichero con el comando file:

```
$ file circulo.o
```

d) ENLAZADO: en nuestro programa se utiliza la función `printf` que está en la librería `stdio`. Para que esta función se incluya dentro del ejecutable a generar, es necesario realizar este enlazado. En esto consiste la etapa de enlace, donde se reúnen uno o más módulos en código objeto con el código existente en las bibliotecas.

```
$ gcc -o circulo circulo.o
```

Si no utilizamos la opción `-o` el nombre que se le asigna por defecto al programa es `a.out`

### 3.2 Estructuración en módulos

En esta sección aprendemos a estructurar un programa en varios módulos. Para ello vamos a crear una función área en un fichero separado y vamos a hacer uso de ella en nuestro programa `circulo.c`

Pasos a realizar:

- Cree un fichero `area.c` y defina una función `float area(float radio)` con la ayuda del código de la figura-1. Recuerde que aquí tiene que ir la implementación de la función.
- Cree un fichero `area.h` e introduzca en él la declaración de la función. Recuerde que aquí únicamente ha de ir la declaración o prototipo de la función.
- Copie el fichero `circulo.c` a `circulo2.c`
- Edite `circulo2.c` y modifíquelo para que incluya el fichero `area.h` y en la función `main` se haga uso de la función `area` implementada en `area.c`.

NOTA: Dado que `area.h` está en el directorio actual se ha de poner `#include "area.h"`

Para compilar podemos hacerlo de la siguiente manera incluyendo los dos ficheros fuentes:

```
$ gcc -o circulo2 circulo2.c area.c
```

esto realiza todas las fases: compilará `circulo2.c` y `area.c` generando sus ficheros objetos (`.o`) y luego enlazará la función `area` para generar un único ejecutable (`circulo2`).

### 3.3 Cuestiones

- Enumere y describa los tipos de ficheros que se han generado en cada una de las fases de la compilación
- En la última compilación los 2 ficheros fuente (`circulo2.c` y `area.c`) son pasados como argumento al compilador `gcc`. Compruebe que ocurre si no pusieramos `area.c` como argumento.

## 4 Ejercicio 2: Ámbito de las variables y gestión de errores.

El objetivo de este ejercicio es doble: primero entender el ámbito de las variables y segundo aprender a corregir los errores surgidos durante la compilación de un programa en C. Para ello se va a proporcionar un programa que contiene una serie de errores (tanto de sintaxis como lógicos) que el alumno tiene que corregir hasta que funcione de forma correcta.

### 4.1 Ámbito de las variables

En C el ámbito de las variables puede ser:

- Global: las variables se declaran fuera de cualquier función y se pueden acceder desde cualquier función del fichero.
- Local: se definen dentro de una función y solo son accesibles dentro de dicha función.
- Estáticas: son variables locales que conservan su valor.

El código fuente del programa a corregir denominado `variables.c`, es el mostrado en la figura-2.

```
#include <stdio.h>
int a = 0; /* variable global */

// Esta función incrementa el valor de la variable global a en 1
void inc_a(void) {
    int a;
    a++;
}

// Esta función devuelve el valor anterior
// y guarda el Nuevo valor v
int valor_anterior(int v) {
    int temp;
    // declarar aquí la variable s estática.

    temp = s;
    s = v;
    return b;
}

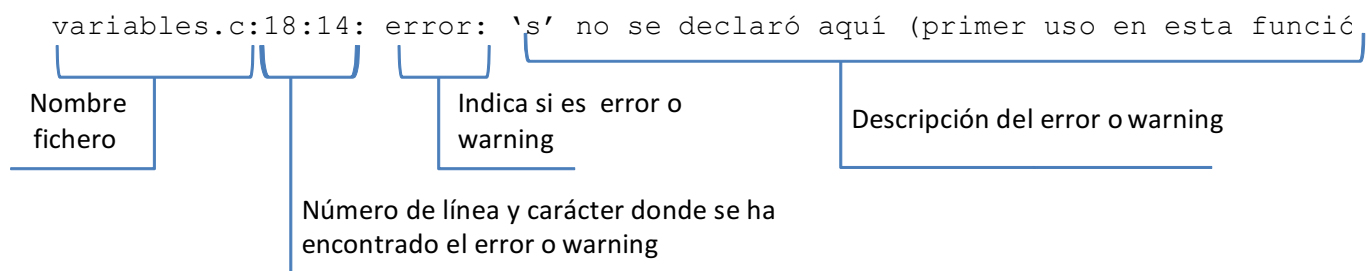
main()
{
    int b = 2; /* variable local */
    inc_a();
    valor_anterior(b);
    printf("a= %d, b= %d\n", a, b);
    a++;
    b++;
    inc_a();
    b = valor_anterior(b);
    printf("a= %d, b= %d\n", a, b);
}
```

Figura-2: Código a corregir proporcionado en el fichero “*variables.c*”

Edite y compile `variables.c`

```
$ gcc -o variables variables.c
```

En pantalla le aparecerán una serie de errores. Los errores suelen tener el siguiente formato:



Hay que distinguir bien entre warnings (avisos) y errores.

- Los **warnings** son avisos que nos proporciona el compilador de código porque 'piensa' que puede ser erróneo o que no cumple algún estándar. Si al compilar se producen warnings y no hay errores, se generará el ejecutable. Los warnings pueden producir errores en ejecución, por ello es recomendable revisar el código e intentar corregirlos
- Los **errores**, han de corregirse todos ya que hasta que no se eliminen no se generará el ejecutable.

El programa también contiene errores de programación que ha de corregir. Al ejecutar `variables.c` debe mostrar en consola:

```
a= 1, b= 2
a= 3, b= 2
```

## 4.2 Cuestiones

a) ¿Qué modificaciones tendríamos que hacer en el programa anterior si declarásemos la variable global `a` como local en `main()`?

## 5 Ejercicio 3: Parámetros por línea de órdenes

Al invocar una orden en UNIX es normal pasarle parámetros. Como hemos visto, el GCC dispone de multitud de opciones y parámetros que se pueden configurar por línea de comandos.

En un programa en C, podemos tratar estos parámetros de forma muy simple con `argc` y `argv`. Para ello hay que definir la función `main` con estos dos argumentos `int main(int argc, char *argv[])` donde:

- `argc` contendrá el número de argumentos pasados, que siempre será mayor que cero, ya que el nombre de la orden es el primer argumento.
- `argv` es un vector de cadenas con los argumentos. El primer elemento de este vector (`argv[0]`) será siempre el nombre de la orden.

En este ejercicio hay que realizar dos programas, partiendo de la base del código siguiente en a editar en un programa de nombre `argumentos.c`.

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    // A completar...
}
```

Figura-3: Contenido del fichero "argumentos.c"

## 5.1 Programa listar argumentos

Realizar un programa denominado `listaArgs`, que muestre por pantalla el número de argumentos y el contenido de cada una de ellos. A continuación se muestra cual debe ser el resultado de su ejecución con distintos argumentos:

```
$ ./listaArgs
Numero de argumentos = 1
Argumento 0 es ./listaArgs
$ ./listaArgs uno dos tres
Numero de argumentos = 4
Argumento 0 es ./listaArgs
Argumento 1 es uno
Argumento 2 es dos
Argumento 3 es tres
```

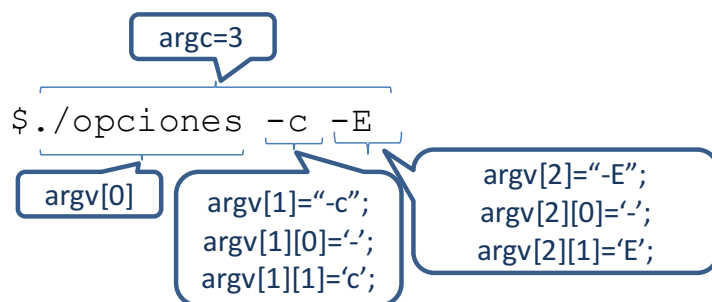
## 5.2 Programa opciones

Realizar un programa denominado `opciones`, que de forma parecida al `gcc` identifique las siguientes opciones y si es el caso muestre la ruta de la opción

```
-c   Mostrará "Compilar"
-E   Mostrará "Preprocesar"
-iruta Mostrará "Incluir + ruta"
```

A continuación se muestra cual debe ser el resultado de su ejecución con distintos argumentos:

```
$ ./opciones -c
Argumento 1 es Compilar
$ ./opciones -c -E -i/includes
Argumento 1 es Compilar
Argumento 2 es Preprocesar
Argumento 3 es Incluir /includes
```





## 6 Ejercicio 4: Punteros y estructuras

Un puntero es una variable que contiene la dirección de otro objeto. Esto nos permite acceder y modificar elementos de cadenas y estructuras de forma simple. En esta parte de la práctica vamos a completar pequeños programas que trabajan con punteros, cadenas y estructuras.

### 6.1 Programa mayúsculas

Completar el programa mayusculas.c (figura 4), que tiene que convertir un texto leído por consola, convertirlo en mayúsculas y luego sacarlo por pantalla. En concreto hay que completar:

- Definir las variables cadena y cadena2 como vectores de cadena con un tamaño TAM\_CADENA.
- Leer un texto de consola y asignarlo a la variable cadena.
- Completar el bucle de conversión a mayúsculas. Para ello se hará uso de dos punteros a cadenas, donde p1 apunta a cadena y p2 apunta a cadena2, y por lo tanto habrá que ir copiando el elemento apuntado por p1 a p2 (restando 32 para convertirlo a mayúscula, sólo si es un carácter en minúscula). Al finalizar del bucle hay que poner el carácter cero de fin de cadena en la cadena2.
- Sacar por consola la cadena2, que tendrá el texto convertido a mayúscula.

```
#include <stdio.h>
#define TAM_CADENA 200
main() {
    // Puntero a caracter para copiar las cadenas
    char *p1, *p2;

    // A) Definir las variables cadena y cadena2
    // B) Leer cadena de consola
    // C) Convertir a mayúsculas
    p1 = cadena;
    p2 = cadena2;
    while (*p1 != '\0') {
        // Copiar p1 a p2 restando 32
    }
    // Acordarse de poner el cero final en cadena2
    // D) Sacar por consola la cadena2.
}
```

Figura-4: Contenido inicial del fichero “mayusculas.c”

### 6.2 Programa sumafilas

Completar el programa sumafilas.c (figura 5), que suma una serie de filas y nos devuelve la suma de cada fila y la suma total. Cada fila es una estructura que contiene dos miembros, un vector con los datos y la suma.

Lo que hay que completar es lo siguiente:

- Definir una variable “filas” que sea un vector de estructuras FILA de tamaño NUM\_FILAS
- Implementar la función suma\_fila. A esta función se le pasa un puntero a la fila a sumar. Tendrá que sumar el vector datos y asignarla al miembro suma.
- Completar el bucle para sumar todas las filas. Hay que llamar a suma\_fila pasándole la fila, completar el printf y actualizar la variable suma\_total.

```
#include <stdio.h>

#define TAM_FILA 100
#define NUM_FILAS 10
struct FILA {
    float datos[TAM_FILA];
    float suma;
```

```

};
// A) Define una variable filas que sea un vector de estructuras FILA de
tamaño NUM_FILAS

void suma_filas(struct FILA *pf) {
// B) Implementar suma_filas
}

// Inicia las filas con el valor i*j
void inicia_filas() {
    int i, j;
    for (i = 0; i < NUM_FILAS; i++) {
        for (j = 0; j < TAM_FILA; j++) {
            filas[i].datos[j] = (float)i*j;
        }
    }
}

main() {
    int i;
    float suma_total;

    inicia_filas();
    // C) Completar bucle
    suma_total = 0;
    for (i = 0; i < NUM_FILAS; i++) {
        // Llamar a suma_filas
        printf("La suma de la fila %u es %f\n", i, /* COMPLETAR */);
        // sumar la fila a suma_total
    }
    printf("La suma final es %f\n", suma_total);
}

```

Figura-5: Contenido inicial del fichero "sumafilas.c"

El resultado final de la ejecución de este programa tiene que ser:

```

$ ./sumafilas
La suma de la fila 0 es 0.000000
La suma de la fila 1 es 4950.000000
La suma de la fila 2 es 9900.000000
La suma de la fila 3 es 14850.000000
La suma de la fila 4 es 19800.000000
La suma de la fila 5 es 24750.000000
La suma de la fila 6 es 29700.000000
La suma de la fila 7 es 34650.000000
La suma de la fila 8 es 39600.000000
La suma de la fila 9 es 44550.000000
La suma final es 222750.000000

```

## 7 Ejercicio 5: Estructura de datos buffer circular

Un buffer circular es una estructura de datos que utiliza un array en el cual existe una posición para insertar un dato (es el dato mas reciente) y otra para sacar un dato (el dato más antiguo). Con este mecanismo de inserción y extracción, adopta una política de FIFO (First In First Out), es decir el primer dato que se inserto es el primero que sale. Esta característica los hace idóneos para estructuras de datos que almacena los mismos ordenados por antigüedad. El efecto que tenemos es una estructura con capacidad siempre de albergar los N datos mas antiguos.

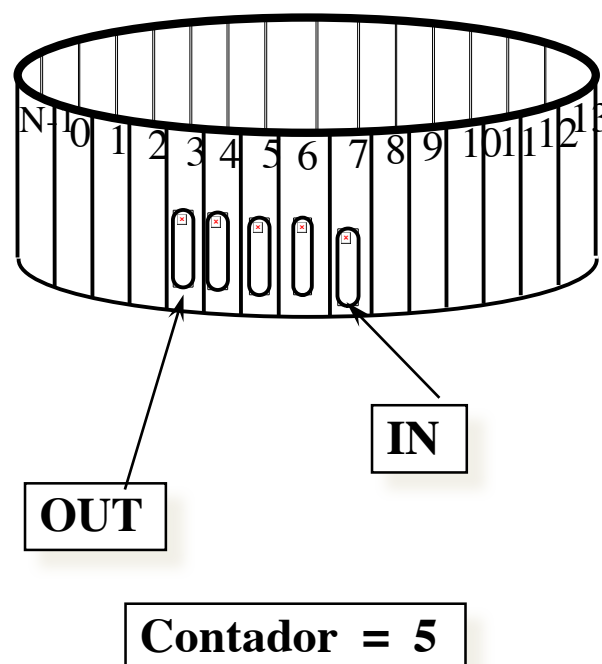
Esta forma de operar los hace interesantes para buffers de datos. Cuando el buffer se llena las operaciones de inserción no se pueden realizar y deben ser notificada dicha eventualidad de alguna forma. De igual manera se debe notificar cuando se va a sacar un dato y no existe ninguna disponible en el mismo. Las operaciones habituales con un buffer circular son:

- Insertar dato
- Sacar dato
- Ver si esta lleno
- Ver si esta vacío
- Listar el contenido
- Conocer la cantidad de datos que alberga

Un buffer circular trabaja básicamente con dos índices para acceder a los elementos del buffer, que aquí llamaremos IN y OUT. Ambos índices tienen avance incremental y cíclico, es decir, se incrementan de uno en uno y luego de apuntar al último elemento del buffer vuelven a apuntar al primero. Esto se realiza mediante una aritmética en modulo del índice del vector. En el caso de implementar la estructura en C, dado que los vectores empiezan en 0 y para un tamaño N, la ultima posición es la N-1, cuando se llega a dicha posición su siguiente es la 0. De esta forma se maneja la condición de circular.

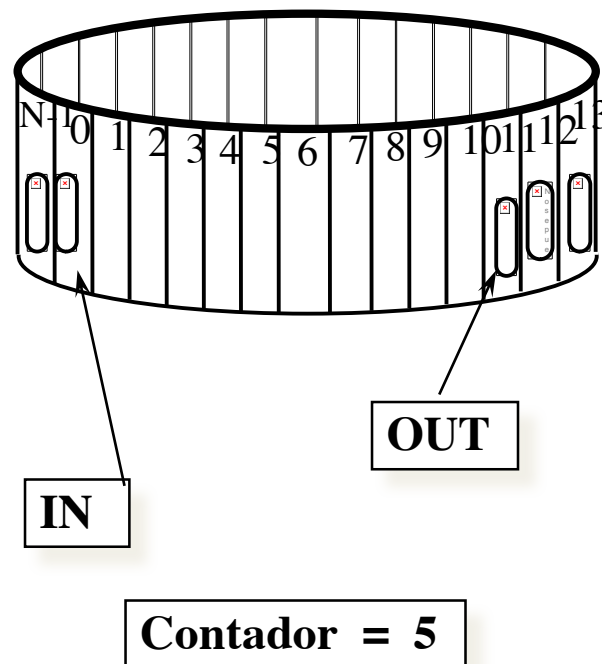
Al inicio los dos índices apuntan al primer elemento del buffer. Veamos cómo y cuándo se incrementan:

- Cada nuevo dato a guardar en el buffer será depositado en la casilla actualmente apuntada por IN. A continuación IN se incrementa en uno.
- Por otro lado, cada dato que salga del buffer será el de la casilla actualmente apuntada por OUT. A continuación OUT se incrementa en uno.



Como podemos observar en la figura, los valores índice de las posiciones por donde extraer o insertar elementos pueden encontrarse desplazados respecto a la primera posición del vector, debido las operaciones realizadas. En el caso de la figura anterior, una situación valida sería haber insertado 8 elementos (del 0 al 7) y extraído 3 (del 0 al 2) quedando en la estructura 5 datos, la siguiente posición a extraer la del índice 3, y la posición con el último elemento insertado la del índice 7.

Incluso podría darse el caso en el que la posición de IN fuera menor en valor de índice del vector que la de OUT.



También es interesante destacar que dada su condición de circularidad, es conveniente mantener un contador de elementos en la estructura.

### 7.1 Programa con vector circular

Implementar un programa en lenguaje C en el que se declare un vector de 20 posiciones. En cada una de las posiciones se guardará el índice de la siguiente posición consecutiva, según un esquema circular de vector. Listar por pantalla el contenido del vector. Implementa el programa con una declaración de constante de tamaño máximo de vector BUFSIZE, de forma que el programa funcione para cualquier valor de BUFSIZE que se declare, sin cambiar ninguna línea de código del mismo.

### 7.2 Programa definición buffer circular

Implementar una estructura de datos en lenguaje C mediante la definición de un struct C Buffer\_Circ, que contenga los campos:

- Vector de enteros de tamaño BUFSIZE (inicialmente con valor 10)
- Índices bufIN y bufOUT
- Variable contador

Así mismo declararse las funciones

- `int get_item(X: *int, buff * Buffer_Circ)` para extraer un dato del buffer circular
- `int put_item(X: int, buff * Buffer_Circ)` para insertar un dato en el buffer circular

Ambas operaciones devuelven 0 si se ha podido realizar la operación, y -1 en caso contrario.

### 7.3 Programa rellenar datos en el buffer circular

- Implementar un programa en lenguaje C que realice la siguiente secuencia de operaciones sobre una variable de tipo Buffer\_Circ.
- 5 inserciones de datos (son 5 invocaciones seguidas a la función `put_item`)
- Mostrar el valor de la variable contador
- 5 extracciones de datos (son 5 invocaciones seguidas a la función `get_item`)

- Mostrar el valor de la variable contador
- 5 inserciones de datos
- Mostrar el valor de la variable contador
- 2 extracciones de datos
- Mostrar el valor de la variable contador
- inserciones de datos
- Mostrar el valor de la variable contador
- extracciones de datos
- Mostrar el valor de la variable contador

#### **7.4 Funciones para completar operaciones en el buffer circular**

Implementar para la estructura Buffer\_Circ las operaciones de:

- Comprobar si una variable Buffer\_Circ esta vacia, devolviendo True si lo esta, False en caso contrario
- Comprobar si una variable Buffer\_Circ esta llena, devolviendo True si lo esta, False en caso contrario
- Listar el contenido de una variable Buffer\_Circ
- Devolver el número de elementos de una variable Buffer\_Circ

Se deja a criterio del alumno los nombres y parámetros de las funciones correspondientes. Una vez implementadas, invocar en el ejercicio anterior la función de listar después de cada secuencia de insertar o sacar datos.