Práctica 2

Tratamiento de ficheros

2.1 Ficheros en C. El tipo `FILE`

C es un lenguaje muy antiguo, lo que hace que se aleje mucho, en algunos aspectos, de los lenguajes algorítmicos. En concreto, el tratamiento de ficheros de C es muy diferente al explicado en teoría. Para empezar, en cualquier libro que lea de C le dirán que existen dos tipos de ficheros: los ficheros de texto y los ficheros binarios. En realidad no hay gran diferencia entre ellos, pero para los de texto hay librerías muy potentes[1].

Todo lo que le explico a continuación está implementado en la librería de C, `libc.a`. Para poder emplearlo debemos incluir el *header* estándar `stdio.h`.

Un fichero de texto es, como hemos visto en teoría, una estructura de datos de acceso secuencial que contiene cadenas de caracteres. Obviamente, un fichero de texto se puede recorrer sin que se altere la secuencia de datos que almacena[2]. Los ficheros binarios, por su parte, son una secuencia de datos de un tamaño dado (el del tipo de dato contenido). Lo más habitual es que los ficheros binarios sólo se abran para leer o para escribir (de modo exclusivo)[3]. A pesar de las diferencias, C emplea el nombre `FILE` para ambos. La librería `libc.a` proporciona operaciones que permiten trabajar con los `FILE` de modo similar a como se ha explicado en teoría (apertura en lectura, lectura, apertura en escritura, escritura, etc.), aunque son distintas para los ficheros de texto que para los binarios.

El lenguaje de programación C dispone una implementación en memoria secundaria del tipo `FILE`. Con esto se consigue que los datos depositados en un fichero queden almacenados en un dispositivo auxiliar de almacenamiento permanente de forma que perduren aún después de haber terminado la ejecución del programa que los creó.

2.1.1 Definición y declaración de ficheros

El constructor de tipos `FILE` se puede usar tanto en la definición de nuevos tipos como en la declaración de variables. El tipo de datos que va a almacenar y el tipo concreto de fichero se fijan al abrirlo.

Una cuestión importante es que todas las funciones que trabajan con ficheros necesitan un apuntador al fichero, por lo que no suelen declararse variables de tipo fichero. Se usan

[1] También los podemos tratar como binarios cuyo tipo es `unsigned char`.

[2] Aunque también podemos abrirlos para modificar su contenido.

[3] Aunque las librerías de C tienen funciones que permiten trabajar con ellos de otras maneras, incluso como si fuesen estructuras de acceso directo.

apuntadores a fichero. Por ejemplo, si se quiere declarar una variable para trabajar con un fichero de enteros grandes debemos escribir:

```
FILE * f;
```

`f` es un apuntador a fichero, pero hasta que no trabajemos con él no podremos indicar que es un fichero de enteros grandes. Hay veces que resulta más cómodo definir un alias para el tipo, la forma de hacerlo sería:

```
typedef FILE FicheroDeEnterosGrandes;
FicheroDeEnterosGrandes * f;
```

Aunque también podríamos poner:

```
typedef FILE * FicheroDeEnterosGrandes;
FicheroDeEnterosGrandes f;
```

Esta segunda forma se acerca más a la representación algorítmica, pero se aleja de las costumbres más habituales entre los programadores de C.

Falta explicar que en cualquier programa C hay siempre predefinidas tres variables de tipo `FILE *` que apuntan a tres ficheros de texto. Son `stdin`, `stdout` y `stderror`. La primera es el fichero estándar de entrada del programa, es decir, la vía por la que el programa espera recibir datos (hablando informalmente, suele ser el teclado); el segundo es el fichero estándar de sálida (o sea, la pantalla) y el tercero el fichero estándar de error (nuevamente la pantalla). El primer fichero está abierto en modo lectura y los dos últimos en modo escritura. En los sistemas unix/linux estas variables pueden redirigirse a través de la ventana de comandos o incluso mediante programa. En otros sistemas esto no siempre es cierto.

2.1.2 Conexión con el sistema de ficheros

Como los objetos de tipo `FILE` se emplean para alterar la información del disco, antes de poder usar un fichero es necesario establecer un vínculo entre la variable de tipo `FILE *` del programa con un determinado archivo almacenado en disco. Esta vinculación se realiza al abrir el fichero.

Para ello podemos hacer dos cosas. Bien relacionar el `FILE *` con una cadena que represente la ruta del fichero en el sistema de archivos o bien solicitar al usuario la ruta del fichero. Obviamente, esta segunda forma es mucho más flexible, pero antes de usar cadenas de caracteres, es preciso que sepamos que son.

Las cadenas de caracteres en C

C no tiene definido el tipo cadena de caracteres, pero si tiene preparada una gran infraestructura para que las usemos. Una cadena de caracteres en C no es más que un array de caracteres terminado con el caracter `NULL`[4], pero C nos permite trabajar con ellos obviando ese detalle. De este modo, podemos inicializar una cadena de caracteres de cualquiera de las dos siguientes formas:

[4]El caracter `NULL es, en realidad '\0'`

```
char saludo1[10]={'H', 'o', 'l', 'a', '\0'};
char saludo2[] = ``Hola'';
```

Obviamente, la segunda forma es más cómoda. Es el compilador el que se encarga de poner el caracter nulo al final y de calcular el tamaño adecuado para el array. Lo más habitual es que no manipulemos directamente las cadenas de caracteres. En lugar de ello recurrimos a las funciones declaradas en el *header* `<string.h>`[5].

C nos permite utilizar variables de tipo cadena de caracteres en las funciones `printf` y `scanf`. Para ello debemos indicar en la cadena de formato que la variable se lee/escribe como una "%s".

Con todo esto, para pedir al usuario el nombre de un fichero a abrir lo que debemos hacer es algo similar a:

```
#define TAMANIO_NOMBRE 101
...
char nombre[TAMANIO_NOMBRE]
...
printf("Escriba el nombre del fichero (menos de 100 caracteres):  ");
scanf("%s",nombre);
```

El tamaño del array debe ser suficiente para que quepa la cadena que escriba el usuario y el caracter `NULL`

2.1.3 Operaciones sobre ficheros en C

Como se ha dicho, en el lenguaje C, hasta que no se abre el fichero no se define el modo en el que se va a trabajar con él. Para abrir un fichero se emplea la función `fopen`. La cabecera de esta función es:

```
FILE *fopen(const char * filename, const char * mode );
```

donde `filename` es una cadena de caracteres que contiene la ruta, relativa o absoluta, del fichero que se desea vincular con la variable a la que se asigna el resultado y `mode` es otra cadena que indica el modo de apertura. Al igual que en la teoría, en C, hay dos modos *excluyentes* de proceso de ficheros: proceso *en modo lectura* y proceso *en modo escritura*[6]:

- "r": Abre un fichero de texto en modo lectura (es la implementación en C de abrir_lec para ficheros de texto). Si el fichero no existe, devuelve `NULL`.
- "rb": Abre un fichero binario en modo lectura (es la implementación en C de abrir_lec para ficheros binarios). Si el fichero no existe, devuelve `NULL`.
- "w": Abre un fichero de texto en modo escritura. Si el fichero existía elimina su contenido. Si el fichero no existía crea uno en la ruta indicada. Es la implementación en C de abrir_esc para ficheros de texto.

[5] https://www.tutorialspoint.com/c_standard_library/string_h.htm

[6] C permite otros modos de apertura más potentes, pero más peligrosos. Se usa "a" (texto) o "ab" (binario) para el modo append, es decir, para añadir cosas al final, "r+", "w+" o "rb+", "wb+" para leer y escribir y, finalmente,"a+" o "ab+" para leer y añadir al final.

- "wb": Abre un fichero binario en modo escritura. Si el fichero existía elimina su contenido. Si el fichero no existía crea uno en la ruta indicada. Es la implementación en C de abrir_esc para ficheros binarios.

Esta función nos devuelve un puntero a un *stream*[7] asociado al fichero. Si no es posible abrir el fichero por cualquier motivo (por ejemplo, falta de permisos) devuelve `NULL`.

En función del modo en que hallamos abierto el fichero podemos después utilizar el resto de operaciones. En `stdio.h` se definen muchas operaciones, pero cabe destacar:

- La operación equivalente al `cerrar` algorítmico es `int fclose(FILE *fp );`. El entero devuelto es cero si todo ha ido bien o `EOF` (la marca de fin de fichero) si ha habido un error. Los errores son posibles porque C no escribe las cosas directamente en los ficheros de disco. Utiliza un `stream` intermedio que el sistema operativo gestiona. En el momento de cierre se fuerza la escritura en el disco de todo lo que esté aún el `stream`.

- La operación equivalente al `leer` algorítmico depende del tipo de fichero con el que estemos trabajando. Para ficheros de texto es `int fgetc(FILE *fp);`. Esta función devuelve el código ASCII del carácter leido o `EOF` si ha habido un error. La función tiene como efecto colateral modificar la posición de la cabeza de lectura, como vimos en teoría. Para ficheros binarios tenemos que utilizar la función[8] `size_t fread(void *ptr, size_t sizeOfElements, size_t numberOfElemts, FILE *a_file);`. Esta función copia a partir de la dirección indicada por `ptr` (que puede ser un array) la cantidad `numberOfElemts` de elementos de tamaño `sizeOfElements` leídos del fichero apuntado por `a_file` y avanza en el fichero todo lo leído. Devuelve el número de elementos leídos. Por lo tanto si todo ha ido bien devuelve `numberOfElemts`. Por lo tanto, para leer un valor `long int` en un fichero de enteros grandes debemos decir `fread(&x, sizeof(long int), 1, f)`.

  Cabe destacar que la librería `stdio.h` contiene otras funciones que permiten leer datos de un fichero como: `char *fgets(char *buf, int n, FILE *fp );`, que lee de `fp` hasta que encuentra el final del fichero o un salto de línea o ha leído `n-1` caracteres y escribe en `buf` lo leído. Obviamente avanza la cabeza de lectura de modo que apunte al primer caracter no leído. También está `int fscanf(FILE *fp, const char *format, ...)`, que es muy similar a la ya familiar `scanf`, pero el uso de estas funciones es más complicado de lo que parece.

  Hay también otras funciones para leer del fichero estandard de entrada (`stdin`), como `int getchar(void)` que lee un caracter y devuelve un `unsigned int` que es el caracter leído o `EOF` si no ha leído nada.

- La operación equivalente al `escribir` algorítmico para ficheros de texto es `int fputc(int c, FILE *fp );`. Esta función escribe en el fichero apuntado por `fp` el caracter de código ASCII `c`. El entero devuelto es el carácter escrito si toda ha ido bien o `EOF` si ha habido un error. Para ficheros binarios la operación equivalente al `escribir` algorítmico es `size_t fwrite(const void *ptr, size_t`

[7] Este término es el que se emplea para hacer referencia al tipo de datos que genera C en memoria para gestionar el fichero

[8] No se asuste, `void *` le permite pasar la dirección de cualquier cosa y `size_t` solo indica que es un tipo particular de entero válido para un tamaño

`sizeOfElements, size_t numberOfElements, FILE *a_file);`. Esta función escribe al final del fichero apuntado por `a_file` la cantidad de `numberOfElements` de elementos de tamaño `sizeOfElements`. El puntero `ptr` (que puede ser un array) indica dónde están los valores a escribir. Devuelve el número de elementos escritos. Por lo tanto si todo ha ido bien devuelve `numberOfElements`.

La librería `stdio.h` contiene otras funciones que permiten escribir datos a un fichero como: `int fputs(const char *str, FILE *stream);`, que escribe en `fp` el contenido de `str` y devuelve el número de caracteres escritos o `EOF` en caso de error. También está `int fprintf(FILE *stream, const char *format, ...)`, que es muy similar a la ya familiar `printf`. Nuevamente el uso de estas funciones es más complicado de lo que parece.

- La operación más parecida al `fdf` algorítmico es `int feof(FILE *fp );`. Esta función devuelve `EOF` si se ha acabado de leer el fichero y 0 en caso contrario. Ahora bien, funciona de un modo muy diferente a como funciona el `fdf` algorítmico. Lo que realmente devuelve es un *flag* del sistema operativo que indica si se ha intentado leer cuando ya se había acabado de leer el fichero, no si aún quedan cosas o no por leer.

En https://es.wikibooks.org/wiki/Programación_en_C/Manejo_de_archivos se le muestran otras funciones de la librería incluida con el *header* <stdio.h>. Si quiere conocer aún más, puede ir a https://www.tutorialspoint.com/c_standard_library/stdio_h.htm.

2.1.4 Ejemplos

Vamos a fijar conceptos con un ejemplo. El siguiente programa reparte en los ficheros binarios (de valores `int`) `pares.dat` e `impares.dat` los valores de un fichero binario que contenga valores `int` cuyo nombre se pide al usuario.

```
// Autor:  Federico Fariña Figueredo .
// version:  1.0.0 (14/3/16)
// Entrada:  (por teclado) nombre, una cadena de caracteres que es el nombre
//      de un fichero binario que contiene valores de tipo int
// Salida:  Crea dos ficheros binarios de int (pares.dat e impares.dat) en el
//      directorio actual.  Escribe los valores pares del fichero indicado por
//      nombre en pares.dat y los impares en impares.dat (cero se considera par)
//
#include <stdio.h>
//
void main() {
    FILE * f;
    FILE * fp;
    FILE * fi;
    int x;
    char nombre[30];
    printf("Federico Fariña Figueredo\n");
    printf("Ejemplo de uso de fichers\n");
    printf("14 de Marzo de 2016\");
    printf("Introduzca el nombre del fichero a tratar\n");
    scanf("%s", nombre);
    f=fopen(nombre,"rb");
    fp= fopen ("pares.dat", "wb");
```

```
    fi= fopen("impares.dat", "wb");
    while (!feof(f)) {
        fread(&x, sizeof(int), 1, f);
        if (x % 2 == 0)
            fwrite(&x, sizeof(int), 1, fp);
        else
            fwrite(&x, sizeof(int), 1, fi);
    }
    fclose(f);
    fclose(fi);
    fclose(fp);
}
```

Tiene buena pinta, ¿verdad?. Pues no funciona. Para comprobarlo cree un fichero vacio (por ejemplo mediante `touch nombre`). Compruebe con `ls -l` que está realmente vacío. Posteriormente ejecute el programa de ejemplo que le he dado diciéndole que separe los valores del fichero que ha creado. Si ejecuta verá (con `ls -l`) que uno de los ficheros no está vacío (además es posible que el suyo no vacío sea uno y el de su compañero otro). ¿Que ha pasado?.

La respuesta es sencilla. Como le había dicho, la función `feof` nos dice sólo si el fichero se había acabado cuando se intento leer por última vez, que no es lo mismo que decirnos si está acabado antes de leer. Dicho de otro modo, en C no hay que tratar la última lectura (la primera que se hace sobre un fichero ya leído). Esto obliga a cambiar algo los esquemas teóricos a la hora de implementarlos en C. Hay diversas alternativas posibles y, además dependen del tipo de esquema[9]:

- Para recorridos (utilizo la lectura de un entero del ejemplo anterior y `tratar(x)` para representar el tratamiento)

  1. Comprobar el éxito de cada lectura y no hacer nada en caso de que no sea un éxito. Es decir, cambiar `tratar(x)` por:

     ```
     if (fread(&x, sizeof(int), 1, f) ==1)
         tratar(x);
     ```

  2. Convertir los recorridos en búsquedas con tratamiento en las que buscamos haber realizado una lectura errónea. Es decir, después de abrir el archivo hacer

     ```
     fread(&x, sizeof(int), 1, f);
     while(!feof(f)) {
         tratar(x)
         fread(&x, sizeof(int), 1, f);
     }
     ```

     A diferencia de lo visto en teoría, este bucle asegura a la salida que se han tratado todos los elementos del fichero.

  3. Mezclar ambas opciones utilizando, además las salidas de las funciones para comprobar que la lectura ha sido correcta. Es decir, tras abrir el fichero hacer:

[9] Y en la comunidad de programadores de C no hay un acuerdo totalmente aceptado, aunque es mayoritario el grupo de los que utilizan la ultima opción de las expuestas en cada caso. Si busca por internet observará que hay otras opciones utilizando la instrucción `break`, pero se las desaconsejo

```
while(fread(&x, sizeof(int), 1, f) == 1) {
    tratar(x)
}
```

Nuevamente, hemos tratado todos los elementos del fichero a la salida del bucle.

- Para búsquedas (Utilizo `A(x)` representando la condición buscada):

  1. Mantener el mismo esquema que en teoría:

     ```
     fread(&x, sizeof(int), 1, f);
     while(!feof(f) && !A(x))
         fread(&x, sizeof(int), 1, f);
     ```

     La forma en que analiza C las expresiones booleanas hace que tengamos que cambiar la comprobación del éxito. Ahora `feof(f)` sí significa que no hay exito en la búsqueda. El orden de las comprobaciones en el while tiene que ser el arriba indicado. Cabe destacar que desaparece la precondición de que el fichero no esté vacío.

  2. Utilizar las propiedades de la evaluación de booleanas en C junto con la comprobación de errores en lectura:

     ```
     fread(&x, sizeof(int), 1, f);
     while(!A(x) && fread(&x, sizeof(int), 1, f) == 1);
     ```

     En este caso, el éxito se comprueba analizando el último valor leído. Recuerde, si se hace verdadera `A(x)`, no se lee el siguiente valor. Obviamente, el orden de las condiciones importa y mucho.

  3. Mezclar ambas opciones

     ```
     while((fread(&x, sizeof(int), 1, f) == 1) && !A(x));
     }
     ```

     En este caso el éxito puede comprobarse analizando la condición o el final del fichero.

- Para búsquedas con tratamiento:

  1. Mantener el mismo esquema que en teoría:

     ```
     fread(&x, sizeof(int), 1, f);
     while(!feof(f) && !A(x)) {
         tratar(x);
         fread(&x, sizeof(int), 1, f) == 1) ;
     }
     ```

     Nuevamente el orden de las comprobaciones en el while tiene que ser el arriba indicado y desaparece la precondición de que el fichero no esté vacío.

  2. Utilizar las propiedades de la evaluación de booleanas en C junto con la comprobación de errores en lectura:

     ```
     while((fread(&x, sizeof(int), 1, f) == 1) && !A(x)) {
         tratar(x);
     }
     ```

Nuevamente el orden de las condiciones importa y mucho.

- Para recorridos asíncronos. El único esquema válido es el de teoría. Eso sí, hay un cambio muy importante. Cuando salimos del primer bucle hemos tratado el último elemento de los ficheros que se han acabado, lo que hace que la solución sea mucho más corta.

En los ejemplos aparecen las funciones de acceso a ficheros binarios. Obviamente la idea es la misma en los de texto.

2.2 Programación robusta con ficheros en C

Pruebe ahora a pasarle a cualquiera de las modificaciones del programa de ejemplo un fichero que no existe. Verá que el programa le devuelve simplemente una indicación de error `Segmentation fault (core dumped)`. Para evitar esto, es recomendable, antes de utilizar una función que trabaje con un fichero, que se puede realmente usar. Por ejemplo, antes de leer sobre un fichero, comprobar que realmente se ha abierto. Lo mejor para hacer esto es chequear que no se han dado las condiciones de error en todas aquellas operaciones que puedan fallar. Por ejemplo, para abrir los ficheros utilizar secuencias de instrucciones como la que le indico ahora:

```
printf("Introduzca el nombre del fichero a tratar\n");
scanf("%s", nombre);
while ((f=fopen(nombre,"rb"))==NULL) {
    printf("Nombre incorrecto.  Introduzca oto nombre\n");
    scanf("%s", nombre);
}
```

Si lo piensa, muchas de las opciones anteriores vienen de esta misma idea. Las funciones de C devuelven muchas veces códigos de error que se pueden aprovechar para mejorar la usabilidad del código.

2.3 Trabajo con ficheros de texto

Como ya hemos visto, los ficheros de texto son un tipo particular de ficheros. Contienen caracteres pero permite interpretar dichos caracteres de muchas formas utilizando `fprintf` y `fscanf`. No ocurre lo mismo en un fichero binario en el que hayamos guardado caracteres. Nunca confunda ficheros de texto con el resto.

Una característica de los ficheros de texto es que permiten escrituras *más visuales*. Ahora bien, esa ventaja puede ser una desventaja cuando los estamos leyendo. Las lecturas tienen que coincidir perfectamente con la estructura del fichero. El problema se complica porque los ficheros standard de entrada, sálida y error (teclado y pantalla por defecto) están declarados internamente como ficheros de texto (el sistema se encarga de abrir y cerrar por su cuenta estos ficheros).

Tenga en cuenta, cuando esté leyendo caracteres, que el *enter* que usted pulsa para finalizar la introducción de un dato es también un caracter que va a parar al mismo fichero.

Siempre debe tener en cuenta que los editores de textos habituales trabajan sólo con ficheros de texto. Es obvio, por tanto, que para ver por pantalla o escribir un fichero

binario deberá implementar algoritmos que lo hagan. Nunca puede emplear un editor habitual.

2.4 Actividades a realizar

1. **Fichero de capicúas2** (`capicuas.c`).

   Cree el programa `capicuas.c` que construye el fichero binario `capicuas.int` de valores de tipo entero con todos los números capicúas menores que 1.008.002 (quizá pueda utilizar la función de la sesión anterior).

2. **Fichero de primos** (`primos.c`).

   Cree el programa `primos.c` que construye el fichero binario `primos.int` de valores de tipo entero con todos los números primos menores que 1.008.002.

3. **Fichero de primos de tipo 4n+1** (`primo4n1.c`).

   Cree el programa `primos4n1.c` que construye el fichero binario `primo4n1.int` con todos los enteros de `primos.int` que sean de la forma $4n + 1$, siendo n un número natural.

4. **Intersección ordenada** (`intersec.c`).

   Cree el programa `intersec.c` que solicita al usuario las rutas de dos ficheros binarios de enteros ordenados de menor a mayor (orden estrictamente creciente) y crea un nuevo fichero binario de enteros ordenados con los valores que aparezcan en ambos ficheros. Utilice este programa para crear a partir de `capicuas.int` y de `primos.int` un nuevo fichero denominado `priycap.int` que contenga todos los valores menores que 1.008.002 que sean a la vez primos y capicuas.

2.4.1 Función para chequear si un número es primo

```
Entrada:  n= N, un numero entero
Requisitos:  n > 1
Salida:  el booleano b sera verdadero si N es primo
funcion esPrimo (n:  entero) dev b:  booleano;
    var
        k:  entero;
    fvar
    si n = 2 ⟶
        b:= verdadero
    [] n = 3 ⟶
        b:= verdadero
    [] n > 3 ⟶
        si n mod 2 = 0 ⟶
            b:= falso
        [] n mod 2 ≠ 0 ⟶
            k := 3;
            mientras (n > k * k) y (n mod k ≠ 0) hacer
                k:= k + 2
```

```
            fmientras;
            b:= n mod k ≠ 0
        fsi
    fsi;
    dev b
ffuncion
```