Entrada Salida

- Se entiende por operaciones de entrada y salida a aquellas que permiten intercambiar datos entre la memoria ram sobre la que trabaja mi programa y cualquier otra fuente datos externa
- Estas fuentes externas se las denomina genéricamente dispositivos, aunque no siempre lo son en el sentido literal de la palabra.
  Ejemplos:
  - Archivos: en discos rígidos, pendrive y otros soportes físicos.
  - Conexión física: por ejemplo por un puerto serie
  - Conexión de red: sockets
  - Tuberías entre procesos

Niveles de acceso

- Como es habitual se puede encarar el tema en distintos niveles
- Bajo nivel:
  - Depende más del sistema operativo
  - Permite un control más fino del entorno con que se opera, por ejemplo: buffers, concurrencia, modalidad a nivel sistema operativo
  - Menos portable
  - Permite el control de parámetros específicos del dispositivo, por ejemplo: velocidad del puerto serial
- Alto nivel:
  - Fácilmente portable
  - Abstrae detalles y se maneja lo que se como un “flujo de datos” (stream)
  - Se basan en las funciones de bajo nivel
- Casos Particulares:
  - Sockets
  - Mapeos de memoria

Flujos de datos

- Si bien no se restringen a archivos, es su uso habitual, de ahí que la estructura que lo representa se llama FILE (razones históricas)
- FILE es una estructura definida en stdio.h que contiene los datos necesarios para el manejo del flujo:
  - Buffers
  - Posición dentro del archivo para la próxima operación
  - Estado, por ejemplo EOF o errores varios
  - Administración de concurrencia (lock)

Declaración

- Habiendo incluido stdio.h se declara
  `FILE *archivo;`
- Notar que defino un puntero, ya que no quiero acceder a los campos de `FILE` sino referenciarlo y que `stdio` lo maneje por mí.
- Además de los flujos que podamos abrir nosotros hay 3 flujos estándar, por así decirlo, ya declarados y abiertos
  - `stdin` (lo habitual es el teclado)
  - `stdout` (lo habitual es la consola)
  - `stderr` (lo habitual es la consola, es común redireccionarlo a un archivo)

Apertura

- Para abrir un archivo usamos la función fopen

```
FILE * fopen (const char *nombre_archivo,
              const char *tipo_apertura);
```

Ejemplo

```
archivo = fopen("datos.dat", "r");
```

- Donde nombre_archivo es el “pathname” al archivo (con todas las particularidades del sistema operativo subyacente)
- Y tipo_apertura es la modalidad con que se abre el archivo

Cierre

- Cerramos un archivo con

```
int fclose(FILE *flujo);
```

Ejemplo

```
fclose(archivo);
```

- Donde flujo es el puntero a FILE donde guardamos el resultado devuelto por fopen

Tipos de apertura

Tipo de apertura	Descripción
r	Solo lectura, si el archivo no existe da error
w	Solo escritura, si el archivo existe borra su contenido anterior, si no existe lo crea
a	Para agregado, es decir escribir al final de archivo. Si el archivo existe su contenido anterior se mantiene, si el archivo no existe se crea
r+	Lectura y escritura, debe existir el archivo, se posiciona al inicio y se mantiene el contenido anterior
w+	Lectura y escritura, si no existe el archivo se crea, si existe se trunca su contenido.
a+	Lectura y escritura, si el archivo no existe se crea, si existe el contenido se mantiene y se agrega al final del mismo.

Nota: En algunos sitemas también se usa b para indicar un flujo binario. Así tenemos rb,wb, ab. Y en los casos con + puede ir antes o después del mismo, o sea, r+b o rb+ En windows, al menos con mingw hay un bug, a veces anda mal si no se usa b en un archivo binario.

Nota 2: A partir de C11 a cualquier opción con w se le puede agregar x como carácter final. Esto indica que el archivo NO debe existir y lo va a crear (con acceso exclusivo)

Texto o binario

- Básicamente se puede acceder un archivo en modo texto o en modo binario.
- Por modo binario nos referimos a no hacer ninguna interpretación particular del contenido del archivo. En general es mover la memoria ram al archivo **sin ningún** tipo de conversión. Así los nros se guardan en el formato binario que se use (complemento a dos, punto flotante)
- Por modo texto entendemos una entrada salida que espera encontrar texto, y dento del mismo separadores de líneas. Hay conversiones, por ejemplo guardamos un nro con una representación, por ejemplo en base 10 o en otra base
- Nota: es común en archivos binarios cambiar la posición desde donde se lee a donde se escribe. Se puede hacer con archivos de texto pero no es habitual

Escritura en modo binario

- Para escribir usamos

```
size_t fwrite(const void *datos,
              size_t tamaño_individual,
              size_t cuantos,
              FILE *flujo);
```

Ejemplo (escribo un vector de 5 elementos de tipo int)

```
escritos = fwrite(vector, sizeof(int), 5, archivo);
```

- Donde
  - Datos es un puntero al dato o al arreglo de datos que quiero escribir
  - tamaño_individual es el tamaño en bytes de cada uno de los elementos
  - Cuantos es la cantidad de elementos a escribir
  - Flujo es el archivo al que vamos a escribir
- fwrite devuelve la cantidad de elementos efectivamente escritos. Si no hubo error debería ser igual al valor de cuantos

Lectura en modo binario

- Para leer usamos

```
size_t fread(void *datos,
             size_t tamaño_individual,
             size_t cuantos,
             FILE *flujo);
```

Ejemplo

```
leidos = fread(&velocidad, sizeof(double), 1, archivo);
```

- Donde
  - Datos es un puntero al dato o arreglo de datos donde quiero cargar lo leído
  - tamaño_individual es el tamaño en bytes de cada uno de los elementos
  - Cuantos es la cantidad de elementos a leer
  - Flujo es el archivo del cual vamos a leer
- fread devuelve la cantidad efectivamente leída de datos (completos)

Lectura escritura en modo texto

- En modo texto hay varias funciones. Si bien no es una regla, es común que si hay una función llamada **fxx** exista otra llamada simplemente **xx** que no tiene el parámetro flujo, y en su lugar se utilice stdin o stdout según sea lectura o escritura
- Las dos funciones genéricas son
  - `fprintf` : igual a printf pero agrega un primer parámetro de tipo `FILE*`
  - `fscanf` : igual a scanf pero agrega un primer parámetro de tipo `FILE*`

Lectura por línea

```
char *fgets (char *string, int cuantos,
             FILE *flujo);
```

- Lee hasta encontrar un \n.
- Incluye el \n y agrega un \0 para terminar la cadena correctamente.
- Guarda en lo leído en lo apuntado por string
- Si no encuentra en \n frena al leer cuantos – 1 caracteres (reserva lugar para el \0).
- Siempre agrega \0 al final (eventualmente sobre el \n)
- Si hay un \0 en medio del flujo fgets falla.
- Si falla devuelve NULL, sino, devuelve string.

Lectura por línea

```
char *gets (char *string);
```

- Similar a fgets pero lee desde stdin. Deprecado en C99, elminado en C11
- Lee hasta `\n` pero **NO** lo incluye en lo leído (a diferencia de fgets). Agrega un `\0` para terminar correctamente la cadena
- No hay control de máxima cantidad de caracteres (por eso es peligoroso y se suele usar getline o uno similar si no es un compilador gnu)

```
char *gets_s (char *string, rsize_t n);
```

- Disponible en forma optativa a partir de C11 (suele no estar implementada)
- Lee hasta `\n` o `EOF` o error, a lo sumo n-1 caracteres, para poder agregar `\0` al final.

Escritura por línea

- `int fputs (const char *string, FILE *flujo);`
  - Escribe la cadena string en el archivo flujo. No escribe en el archivo el `\0` final del string ni agrega `\n`, solo copia el contenido propiamente dicho del string .
  - Devuelve `EOF` si falló y un valor mayor a cero si tuvo éxito.

`int puts (const char *string);`

  - Similar a fputs pero escribe a stdout
  - A diferencia de fputs **SI** agrega un `\n` al final de string al copiarlo en stdout. No copia el `\0` final.

Lectura por caracteres

```
int fgetc(FILE *flujo);
```

- Lee desde flujo un carácter. Devuelve un int para poder incluir EOF.

```
int getc(FILE *flujo);
```

- Lee desde flujo un carácter.
- Llamativamente la función **getc** no lee de stdin, sino que es un **versión optimizada** de fgetc. En general se prefiere usar getc en lugar de fgetc

```
int getchar(void);
```

- Lee desde stdin un carácter.
- Esta si es la versión de fgetc que lee de stdin

Escritura por caracteres

```
int fputc(int c, FILE *flujo);
```

- Convierte c al tipo unsigned char y lo escribe en flujo. Si no hubo error devuelve c, sino EOF.

```
int putc(int c, FILE *flujo);
```

- Versión optimizada de fputc. (En general, al igual que getc, depende del sistema donde esté implementada)

```
int putchar(int c);
```

- Escribe c a stdout

```
int ungetc(int c, FILE *flujo);
```

- No es propiamente de escritura. Se utiliza para “devolver” un carácter leído al flujo. Es habitual en algoritmos que deben reconocer patrones

Posicionamiento

- Nos referimos a la posición dentro del archivo en la cual haremos nuestra próxima operación, ya sea de lectura o escritura
- Es mirar el archivo como un arreglo de bytes
  - El primer byte esta desplazado (offset) 0 bytes respecto al inicio.
  - El último está desplazado en el tamaño del archivo menos uno.
  - Si nos desplazamos el tamaño del archivo y leemos, dará como resultado `EOF`
- Tenemos dos tipos de funciones, las “tradicionales” siguen el estándar POSIX y usan int para indicar desplazamientos. Es conveniente usar las que agregan al nombre una **o** al final, indicando que usan parámetros de tipo `off_t`

Funciones de posicionamiento

- Para averiguar en que posición se está ahora

```
long int ftell(FILE *flujo);
off_t ftello(FILE *flujo);
```

  - Devuelven -1 si hubo error

- Para cambiar la posición

```
int fseek(FILE *flujo, long int desp, int desde_donde);
int fseeko(FILE *flujo, off_t desp, int desde_donde);
```

  - Si hubo éxito devuelven cero
  - El parámetro desde_donde puede ser
    - SEEK_SET (desde el inicio)
    - SEEK_CUR (desde la posición actual)
    - SEEK_END (desde el final)
  - Implica bajar buffers pendientes (flush buffers), limpiar señales de `eof` (end of file) y descartar operaciones de ungetc pendientes

Otras Operaciones

- Para cerrar todos los archivos abiertos
```
int fcloseall(void);
```
  - Devuelve 0 si puedo cerrarlos todos, EOF si hubo algún error
- Para averiguar si se llegó al EOF en un archivo
```
int feof(FILE *flujo);
```
  - Devuelve algo distinto de cero, o sea verdadero, si se llegó al final de archivo
- Para bajar buffers a disco
```
int fflush(FILE *flujo);
```
  - Devuelve EOF si hubo error
  - Si el parámetro flujo es NULL lo hace para todos los archivos abiertos

Licencia